University of Glasgow

Adam Smith Business School

WORKING PAPER SERIES

Markov-Switching DSGE Modeling in RISE

Junior Maih, Nigar Hashimzade, Oleg Kirsanov and Tatiana Kirsanova

Paper No. 2026-01
January 2026

Markov-Switching DSGE Modeling in RISE[1]

Junior Maih[2]
Nigar Hashimzade[3]
Oleg Kirsanov[4]
Tatiana Kirsanova[5]

January 21, 2026

Prepared for the Edward Elgar Handbook of Research Methods and Applications in Empirical Macroeconomics, 2nd ed.

[1]This paper should not be reported as representing the views of Norges Bank. The views expressed are those of the authors and do not necessarily reflect those of Norges Bank. All errors remain ours.

[2]Norges Bank; e-mail Junior.Maih@norges-bank.no
[3]Department of Economics and Finance, Brunel University of London, United Kingdom; e-mail: nigar.hashimzade@brunel.ac.uk
[4]Adam Smith Business School, University of Glasgow, United Kingdom; e-mail: oleg.kirsanov@glasgow.ac.uk
[5]Adam Smith Business School, University of Glasgow, United Kingdom; e-mail: tatiana.kirsanova@glasgow.ac.uk

Contents

1 Markov-Switching DSGE Modeling in RISE[1]

Junior Maih, Nigar Hashimzade, Oleg Kirsanov and Tatiana Kirsanova

1 Introduction

Many important episodes in modern macroeconomics are defined by *temporary* shifts between different economic conditions: monetary policy may switch between dovish and hawkish stances, external shocks between high and low volatility, financial markets between periods of tight and loose frictions, and so on. Standard linear DSGE models cannot accommodate such shifts in behavior. A natural extension is multiple-regime models, in which an economy at any given time is in one of several regimes and selected parameters take different values in each regime. One popular way to model transitions between regimes is via a finite-state Markov process.This framework captures recurrent episodes parsimoniously while preserving the structural discipline of DSGE modeling.

The main challenge for researchers is computational: a Markov-switching rational expectations model is considerably more complex to solve and estimate than its standard single-regime counterpart. Expectations must be treated consistently across regimes, and econometric inference requires specialized filters, which estimate both the probability of the economy being in each regime and the values of unobserved (latent) variables, such as the output gap.

The RISE toolbox for MATLAB is designed to make this workflow straightforward. It allows users to declare Markov chains and regime-specific parameters, solve switching models by perturbation methods, and estimate them using dedicated switching filters. The outputs—regime probabilities (updated and smoothed), latent variables, and regime-dependent impulse responses—are precisely what applied macroeconomists need for empirical work.

The RISE toolbox. RISE (Rationality in Switching Environments) is a MATLAB toolbox developed by Junior Maih since 2011 for solving, simulating, and estimating dynamic models with Markov switching. Although RISE can handle a broad class of state-space systems—not limited to DSGE models—this chapter focuses on DSGE applications.

The toolbox is designed to support the typical workflow of an empirical macroeconomist working with regime-switching models. It provides a modular framework that allows users to specify a model, assign parameters, obtain a solution, estimate the model, conduct simulations or counterfactual analyzes, and more. At each step, the results can be retrieved, inspected, and used as input for subsequent tasks. This modular structure also makes it easy to combine RISE's built-in functions with user-written MATLAB code, offering substantial flexibility for customized research applications.

The toolbox and documentation (the manual) are freely available at `https://github.com/jmaih/RISE_toolbox`. In this chapter, we use RISE release 20240624.

The scope of this chapter. This chapter provides a complete start-to-end workflow example. Starting from a minimal single-regime New Keynesian model, we extend it to include a policy rule that switches

[1]This chapter should not be reported as representing the views of Norges Bank. The views expressed are those of the authors and do not necessarily reflect those of Norges Bank. All errors remain ours.

between dovish and hawkish types of behavior. Using artificial data, we illustrate the entire pipeline—solution, simulation, filtering, and smoothing—and show how to interpret RISE's outputs, such as regime probabilities and latent-variable estimates. After that, we turn to empirical estimation—first by posterior maximization to obtain point estimates and then by MCMC methods to sample from the posterior distribution. These estimation results form the basis for posterior-based inference and the applications discussed in further detail later in the chapter.

Our aim is not to reproduce the user manual, but to explain the workflow—from model declaration to estimation and inference—through a concrete, reproducible example. The models used in this chapter for illustration are designed to make the code syntax intuitive and to clarify the logic of a typical workflow, enabling readers to adapt and extend the code for their own models and research questions. To keep the exposition clear, we focus on the essential commands and their most important outputs, omitting various optional arguments and diagnostic returns available in each function call. Some of these additional options are described in the final section of this chapter, while the complete list is available in the RISE manual.

2 The Single-Regime Baseline

Before introducing Markov switching, we present a linearized New Keynesian (NK) model that serves as a didactic baseline for illustrating the RISE workflow. The setup is nearly identical to Chen et al. (2017) and closely related to the richer NK models covered elsewhere in this volume (see, for example, Guerrón-Quintana and Nason, 2026). Most elements of a typical empirical workflow—model specification, solution, simulation, and estimation—can be illustrated using this linearized version. In Section 7, we return to the same model in its original nonlinear form to discuss additional computational aspects of solving such models, including finding the steady state and obtaining the dynamic solution.

2.1 A Minimal New Keynesian Example

The NK model includes habit formation, a hybrid Phillips curve, a generalized Taylor rule, and three persistent shocks. Log-linearized around the steady-state, it is described by the following system of equations:

$$\text{Consumption Euler equation} \quad c_t = \mathbb{E}_t c_{t+1} - \tfrac{1}{\sigma}\left(r_t - \mathbb{E}_t \pi_{t+1} - \mathbb{E}_t z_{t+1}\right) - \xi_t + \mathbb{E}_t \xi_{t+1}, \tag{1.1}$$

$$\text{Habit dynamics} \quad c_t = (1-\theta)^{-1}(y_t - \theta y_{t-1}), \tag{1.2}$$

$$\text{Phillips curve} \quad \pi_t = \chi_f \beta \mathbb{E}_t \pi_{t+1} + \chi_b \pi_{t-1} + \kappa \omega_t + \mu_t, \tag{1.3}$$

$$\text{Output gap} \quad \omega_t = y_t - \sigma\theta/(\sigma + (1-\theta)\varphi)\, y_{t-1}, \tag{1.4}$$

$$\text{Policy rule} \quad r_t = \rho_r r_{t-1} + (1-\rho_r)\left[\phi_1 \pi_t + \phi_2\left(y_t - y_{t-1} + z_t\right)\right] + \sigma_r \varepsilon_t^r, \tag{1.5}$$

$$\text{Technology shock} \quad z_t = \rho_z z_{t-1} + \sigma_z \varepsilon_t^z, \tag{1.6}$$

$$\text{Cost-push shock} \quad \mu_t = \rho_\mu \mu_{t-1} + \sigma_\mu \varepsilon_t^\mu, \tag{1.7}$$

$$\text{Preference shock} \quad \xi_t = \rho_\xi \xi_{t-1} + \sigma_\xi \varepsilon_t^\xi. \tag{1.8}$$

Equation (1.1) is the standard consumption Euler equation derived from the household optimization problem, where c_t is habits-adjusted consumption given by equation (1.2), and y_t denotes output, π_t is inflation, and r_t is the nominal interest rate. Here, σ is the inverse of the intertemporal elasticity of substitution, φ is the inverse of the Frisch elasticity, and θ is the habit persistence parameter. The

process ξ_t is a preference shock, and z_t is a technology shock. The firms' optimization decisions, in presence of both price and inflation inertia (Galí and Gertler, 1999), give rise to a hybrid New Keynesian Phillips curve (1.3) where ω_t is output gap given by equation (1.4), and the reduced form parameters are $\chi_f \equiv \gamma(1-\zeta(1-\gamma))/\varkappa$, $\chi_b \equiv \zeta/\varkappa$, $\kappa \equiv ((1-\theta)^{-1}\sigma+\varphi)(1-\gamma)(1-\zeta)(1-\gamma\beta)/\varkappa$, $\varkappa \equiv \gamma(1+\beta\zeta)$, where γ is the Calvo (1983) probability of keeping a price unchanged, β is the households' discount factor, and ζ is the proportion of firms setting prices who follow a backward-looking rule of thumb, rather than setting prices optimally. The process μ_t describes a cost push shock. The Taylor rule (1.5) includes interest–rate smoothing and a response to deviation of output growth from the trend, $y_t - y_{t-1} + z_t$, as in An and Schorfheide (2007). Equations (1.6)–(1.8) describe AR(1) processes for the technology, cost-push shock, and preference shocks, respectively. Random components ε_t^r, ε_t^z, ε_t^μ, and ε_t^ξ are exogenous i.i.d. innovations with zero mean and unit variance.

Thus, the model features eight endogenous variables—three of which are predetermined (lagged inflation, lagged output, and the lagged policy rate)—and four exogenous innovations. The model is simple yet rich enough to illustrate RISE's solution, simulation, and estimation mechanics.

2.2 The Model File

Box 1 shows the RISE model file (nk.rs) corresponding to equations (1.1)–(1.8). An .rs file is a plain-text file that can be edited in any text editor and is organized into blocks. Lines 1–9 declare the *endogenous* and *exogenous* variables and the model *parameters*. The central @*model* block (line 11 onward) contains the equations of the system, using curly braces for timing (e.g., Y{t-1} for Y_{t-1}).

Box 1: RISE model file (nk.rs)

```
 1  @endogenous
 2  Y, Pai, R, C, Omega, Z, Mu, Xi
 3
 4  @exogenous
 5  Ez, Emu, Exi, Ei
 6
 7  @parameters
 8  beta, sigma, varphi, theta, gamma, zeta, rho_z, rho_mu, rho_xi, rho_r,
 9  phi1, phi2, sig_z, sig_mu, sig_xi, sig_r
10
11  @model
12  % Definitions
13  # varkappa =  gamma*(1+beta*zeta);
14  # chi_f    = gamma*(1-zeta*(1-gamma))/varkappa;
15  # chi_b    = zeta/varkappa;
16  # kappa    = (sigma/(1 - theta) + varphi)*(1-gamma)*(1-zeta)*(1-gamma*beta)/varkappa;
17
18  % Equations
19  C{t} = (Y{t} - theta*Y{t-1} ) / (1 - theta);
20  C{t} = C{t+1} - (1/sigma)*( R{t} - Pai{t+1} - Z{t+1} ) - Xi{t} + Xi{t+1};
21  Pai{t} = chi_f*beta*Pai{t+1} + chi_b*Pai{t-1} + kappa*Omega{t}  + Mu{t};
22  Omega{t} = Y{t} - sigma*theta*Y{t-1}/(sigma +(1 - theta)*varphi);
23  Z{t}  = rho_z*Z{t-1}  + sig_z*Ez{t};
24  Mu{t} = rho_mu*Mu{t-1} + sig_mu*Emu{t};
25  Xi{t} = rho_xi*Xi{t-1} + sig_xi*Exi{t};
26  R{t} = rho_r*R{t-1} + (1 - rho_r)*( phi1*Pai{t} + phi2*( Y{t} - Y{t-1} + Z{t}) ) + sig_r*Ei{t};
```

In this file, the @*model* block begins with a *definitions* section (lines 12–16). Here, we place

composite, or derived, coefficients such as χ_f, χ_b, and κ, which depend on deeper structural parameters, including γ and ζ. Declaring these relationships inside the model file ensures that, whenever the underlying parameters change, the composite terms are automatically recomputed.

The structural equations follow in lines 19–26: the habit relation, the Euler equation, the hybrid Phillips curve, the output gap, the AR(1) processes for the shocks, and the Taylor rule. Saving this file in the working directory on the MATLAB path is sufficient for RISE to use it.

2.3 Minimal Driver

After defining the model in the `nk.rs` file, we turn to a MATLAB driver script (Box 2) that executes the workflow. In this script, we specify what we need from RISE: *parse* the model, *parameterize* it, and *solve*. The same script also performs two basic validation checks using the solution: *(i)* plotting impulse responses and *(ii)* simulating a sample to inspect the model's dynamics and the implied magnitudes of key variables. We will reuse this minimal skeleton in subsequent sections.

Box 2: Minimal driver script (`driver_nk.m`)

```
 1 clear all; close all; clc;
 2 m = rise('nk.rs');
 3      % Parameterization
 4 p = struct('beta',1/(1+0.706/400),'sigma',2.9,'varphi',2.5,'theta',0.82,'gamma',0.77,...
 5  'zeta',0.10,'rho_z',0.90,'rho_mu',0.70,'rho_xi',0.80,'rho_r',0.79,'phi1',1.72,...
 6  'phi2',0.49,'sig_z',0.50,'sig_mu',0.15,'sig_xi',0.10,'sig_r',0.10);
 7 m = set(m,'parameters',p);
 8      % Solve and print
 9 m = solve(m); vList = {'R','Y','Pai','C','Omega','Mu','Xi','Z'}; print_solution(m,vList);
10      % IRFs: compute and plot
11 IRF = irf(m,'irf_periods',24); quick_irfs(m,IRF,{'Pai','Y','R'},{'Ez','Emu','Exi','Ei'});
12      % Simulation: simulate, access the data and print basic statistics
13 rng(1234); nsim = 300; mysim = simulate(m,'simul_periods',nsim);
14 fprintf('Std Pai=%.3f, Y=%.3f,R=%.3f\n',...
15  std(mysim.Pai.data), std(mysim.Y.data), std(mysim.R.data));
```

Parsing and parameterization. Line 2 instructs RISE to parse the model file `nk.rs` and create the object `m`. Parsing checks the syntax of the file, reads and stores variables, parameter declarations and equations. RISE uses MATLAB's object-oriented features: when a model file is parsed, it creates a *model object*—a structured container that stores the model's variables, parameters, and equations together with the methods used for solution and estimation. At this stage, RISE also computes analytic derivatives in symbolic form, which are subsequently used to obtain the solution to the first-order (linear) model.

Lines 4–7 assign numerical values to model parameters. The parameters are collected in a MATLAB structure `p`, whose field names must match those declared in the `.rs` file. All values are calibrated for *quarterly frequency*: for example, the quarterly discount factor $\beta = 1/(1 + 0.706/400)$ corresponds to an annual steady-state nominal rate of about 0.706 percent. The command `set(m,'parameters',p)` attaches these values to the model object, producing a parameterized version of `m`.

Model objects in RISE. Once created, the model object `m` contains the full symbolic representation of the system but no numerical solution (`m.nsols = 0`). Most RISE commands—such as `set`, `solve`, and `estimate`—do not modify the object in place. Instead, each call returns an updated copy that reflects the requested change. Hence, the result must be assigned to preserve the latest state, for example,

`m = set(m,'parameters',p)` or `m = solve(m)`. This convention ensures that `m` always refers to the current version of the model. Experienced users sometimes keep separate instances—`ms` for the solved model, `me` for the estimated one—to compare results at different stages. However, for clarity, we reuse `m` where practical.

2.4 Solving the Model

After executing `m = solve(m);` in line 9, the object `m` now *stores* the solution (`m.nsols=1`) together with the state–space matrices; `print_solution(m)` merely displays what is already stored in `m`. Note that RISE does not retain the solution unless the output of `solve` is assigned. Box 3 shows the output of `print_solution(m)`.

Box 3: Solution printout (`print_solution(m,vList)`)

```
SOLVER :: rise_1

Regime    1 : const = 1
                 R           Y          Pai          C         Omega        Mu     Xi      Z
             ---------   ---------  ---------   ---------  ----------   ----   ---   ---

    Mu{-1}      0.41011    -0.14441     1.1766    -0.80225    -0.14441    0.7      0     0
    R{-1}        0.6054    -0.13557    -0.47246   -0.75317    -0.13557      0      0     0
    Y{-1}       0.080049     0.79599     0.27974   -0.13341     0.086135     0      0     0
    Pai{-1}     0.035796   -0.0082695     0.10146   -0.045942   -0.0082695     0      0     0
    Xi{-1}      -0.11383    -0.081693    -0.29187   -0.45385    -0.081693      0    0.8     0
    Z{-1}        0.27048      0.12642     0.45644    0.70235      0.12642      0      0   0.9
    Ei          0.076633    -0.017161   -0.059805   -0.095338   -0.017161      0      0     0
    Emu         0.087881    -0.030944     0.25212   -0.17191    -0.030944   0.15      0     0
    Exi        -0.014229    -0.010212   -0.036484   -0.056732   -0.010212      0    0.1     0
    Ez           0.15027     0.070235     0.25358    0.39019     0.070235      0      0   0.5
             ---------   ---------  ---------   ---------  ----------   ----   ---   ---

                 R           Y          Pai          C         Omega        Mu     Xi      Z
```

The top line in the output shows the solver used, `rise_1`, which is similar to the algorithm described in Klein (2000). The first column of the table lists the *predetermined state variables* (`Mu{-1}`, `R{-1}`, `Y{-1}`, `Pai{-1}`, `Xi{-1}`, `Z{-1}`), ordered alphabetically, followed by the *current shocks* (`Ei`, `Emu`, `Exi`, `Ez`). The first row lists the endogenous variables as given in (optional) `vList`: (`R`, `Y`, `Pai`, `C`, `Omega`, `Mu`, `Xi`, `Z`)—that is, the outcomes of economic agents' decisions (sometimes referred to as the "policy functions").

Each entry in the table is a coefficient linking the state variable or shock in that row to the corresponding column variable in the linearized solution representation

$$\alpha_t \;=\; T\,\alpha_{t-1} \;+\; R\,\varepsilon_t. \tag{1.9}$$

Here, $\alpha_t = \left[r_t, y_t, \pi_t, c_t, \omega_t, \mu_t, \xi_t, z_t\right]'$ is the vector of the contemporaneous endogenous variables, α_{t-1} is the lagged α_t, and $\varepsilon_t = \left[\varepsilon_t^r,\, \varepsilon_t^\mu,\, \varepsilon_t^\xi,\, \varepsilon_t^z\right]'$ is the vector of exogenous shock innovations. This equation, also known as the *transition equation*, maps the vector of lagged endogenous variables and current shocks to contemporaneous endogenous variables. Because all endogenous variables are functions only of predetermined states and current innovations, the square matrix T contains several columns of zeros. RISE therefore concatenates the non-zero columns of T with R, transposes the resulting matrix, and displays it as shown in Box 3.

In this baseline model, all variables are expressed as log deviations from the steady-state, so the constant term is zero and therefore not reported in (1.9). More generally, the first-order solution can include a vector of non-zero constants.

The call to `irf` on line 10 creates a database of impulse response trajectories for each variable and each unit shock. `quick_irfs` is a utility function that plots impulse responses. It allows many options—listed in the manual—to customize the output.

2.5 How and Why Artificial Data are Simulated

The minimal driver shown in Box 2 contains the command `simulate` on line 13—a call that uses the solved law of motion (1.9) as a stochastic data–generating process. Given matrices (T, R) and an initial state α_0—the steady-state by default—the simulation draws an i.i.d. vector $\varepsilon_t \sim \mathcal{N}(0, I)$ and iterates (1.9) forward to produce artificial time series from the model. Because the shocks process is stochastic, it is a good practice to set the random–number seed (e.g. using `rng`) to ensure the results are reproducible.

The simulation step is more than a technical convenience. It allows us to visualize the model's implied dynamics along the sample paths, to verify that simulated variables behave plausibly given the parameters, and—later in the workflow—to generate artificial datasets with known "true" states, which we use as benchmarks when discussing filtering in Section 4.

The output of `simulate` is stored in a MATLAB structure called `mysim`. Each field of `mysim` corresponds to a model variable (`mysim.Pai`, `mysim.Y`, etc.), and each field is itself a `ts` (time series) object. A `ts` object combines the numerical array with metadata such as frequency, start date, and end date—for example, `mysim.Pai` represents a quarterly series with its time index, while `mysim.Pai.data` extracts the underlying numerical values (a vector of doubles) from the simulated dataset. This organization makes it easy to plot series with the correct time axis or to pass them directly into estimation routines without worrying about alignment.

The workflow presented in Box 2 includes the main steps `solve` and `simulate` explicitly, rather than using the Dynare-style shorthand `res = stoch_simul(m)`. This convenience command reproduces Dynare's familiar output tables—model summary, simulated moments, correlations, and autocorrelations—and may help new users verify their setup. However, we do not rely on it here, as separating `solve` and `simulate` exposes the underlying mechanics of the solution and facilitates learning, debugging, and later extensions to multi-regime or estimated models.

2.6 What `RISE` Does Under the Hood: The Three–matrix Form

Within `solve` with the default first–order solution options, `RISE` (i) computes the steady-state (trivial here, as the model is log-linearized around the steady-state), (ii) linearly approximates the equilibrium conditions (in this model they are already supplied in the linear form), and (iii) casts the model into the standard rational expectations system:

$$A_{-1}\alpha_{t-1} \;+\; A_0\alpha_t \;+\; A_{+1}\mathbb{E}_t\alpha_{t+1} \;+\; B\,\varepsilon_t \;=\; 0, \tag{1.10}$$

where α_t is the vector of endogenous variables, and ε_t is the vector of exogenous shock innovations. Given (1.10), the solution takes the form of a VAR with shocks (Blanchard and Kahn, 1980), as already shown in (1.9), with stable T (all eigenvalues are inside the unit circle). We show in Section 7 how to retrieve matrices T, R, the three As, and other structural matrices for use in other tasks.

In more general nonlinear models, `RISE` provides a local perturbation approximation around the stochastic steady-state (Judd, 1996; Maih, 2015), of which the linear (first-order) case above is the simplest.

3 From Single to Multiple Regimes with Markov Switching

In the previous section, we demonstrated how to solve an NK model under the assumption of constant parameters. In reality, the observed dynamics of economic variables are often better described by models in which parameters switch between different values. For example, a model may capture shifts in monetary policy or in the volatility of exogenous shocks across episodes. The research question itself may aim to distinguish the effects of exogenous shocks from those of policy behavior. For instance, inflation may be low either because of good policy (the central bank reacts aggressively to inflation) or because of good luck (the shocks happen to be favorable).

A parsimonious way to capture such dynamics is through a multiple-regime model in which selected structural parameters or shock variances take different values depending on an auxiliary latent state variable s_t that follows a Markov process (Hamilton, 1989; Kim and Nelson, 1999).

In this section, we analyze a version of the model in which the discrete variable s_t follows a finite-state Markov chain with transition matrix $Q = (q_{ij})$, where $q_{ij} = \Pr(s_t = j \mid s_{t-1} = i)$ and each row sums to one. With a single Markov chain, each possible value of s_t—each *state* of the chain—defines a configuration of parameters, or *regime*. For example, suppose s_t takes two values representing "hawkish" and "dovish" policy behavior. In that case, the model has two states (and hence two regimes), and Q governs the probabilities of switching between them over time.

More generally, a model may include several Markov chains, each controlling a different subset of parameters—for example, one for policy behavior and another for shock volatility. Suppose each chain has two states: policy can be either dovish or hawkish, and volatility can be either high or low. At any point in time, the economy is in one of four possible combinations of these chain states, and each combination defines an overall regime. Hence, a model with two binary chains has four regimes: {Dovish–Low, Dovish–High, Hawkish–Low, Hawkish–High.}

In what follows, we focus on a simple example featuring a two-state Markov chain—that is, two regimes—and show how to implement it directly in a RISE model file. Extensions to multiple chains are straightforward; see Section 5 for an example of a model with four regimes.

3.1 Modifying the Model and the Driver

Model modification. A natural starting point is to allow the coefficients in the Taylor rule (1.5) to differ across states, thereby modeling a dovish versus a hawkish central bank. We assume that the policy rule depends on the state $s_t \in \{H, D\}$:

$$r_t = \rho_r r_{t-1} + (1-\rho_r)\left[\phi_1^{(s_t)}\pi_t + \phi_2^{(s_t)}\left(y_t - y_{t-1} + z_t\right)\right] + \sigma_r \varepsilon_t^r,$$

where a stronger long-run feedback on inflation defines the hawkish policy response, $\phi_1^H > \phi_1^D$.

Model file changes. The model equations themselves do not change; instead, we introduce a Markov chain and link selected parameters to it so that their values become state-specific. In RISE, a chain named `cname` with `h` states is created by (i) defining transition-probability parameters `cname_tp_i_j` for all *off–diagonal* elements ($i \neq j$; the diagonal elements are implied by the row-sum restrictions, $q_{ii} = 1 - \sum_{j \neq i} q_{ij}$), and (ii) *moving* the parameters that should switch into a chain–scoped block `@parameters(cname,h)`.

For example, to define a two–state policy chain, we add `policy_tp_1_2` and `policy_tp_2_1` as ordinary parameters (their names are arbitrary apart from the `_tp_i_j` pattern), and we place `phi1`, `phi2`

under `@parameters(policy,2)` while removing them from the constant `@parameters` block. The Taylor rule in the `@model` block remains unchanged; RISE automatically uses the appropriate state–specific values of `phi1` and `phi2` according to the current state of the `policy` chain.

To implement this, create a copy of the file `nk.rs` named `nk_ms.rs`. Then, amend the list of parameters (lines 7-9 in Box 1) as follows:

```
@parameters beta, sigma, varphi, theta, gamma, zeta, rho_z, rho_mu, rho_xi, rho_r,
sig_z, sig_mu, sig_xi, sig_r, policy_tp_1_2, policy_tp_2_1

@parameters(policy,2) phi1, phi2
```

The `@model` block itself remains unchanged.

Driver file changes. Only the parameterization needs to be modified by adding the two transition-probability parameters and redefining the policy-rule coefficients as state-specific. The chain name must match the one used in the `nk_ms.rs` file (`@parameters(policy,2)`). Transition probabilities must follow the off–diagonal pattern `policy_tp_i_j` with $i \neq j$ (the diagonal elements are implied by the row-sum restrictions, as noted above). Parameters `pname` that were constant before (i.e., `phi1`, `phi2`) are now defined separately for each state of the chain, following the naming convention `pname_cname_k`, where `k` indicates the numeric index of the state within that chain.

Create a new driver file `driver_nk_ms.m`, identical to `driver_nk.m` in Box 2, except that it loads the new model by calling `m = rise('nk_ms.rs')` in line 2, and the parameterization in lines 4-6 becomes:

```
p = struct('beta',1/(1+0.706/400),'sigma',2.9,'varphi',2.5,'theta',0.82, ...
   'gamma',0.77,'zeta',0.10,'rho_z',0.90,'rho_mu',0.70,'rho_xi',0.80, ...
   'rho_r',0.79,'sig_z',0.50,'sig_mu',0.15,'sig_xi',0.10,'sig_r',0.10,...
   'policy_tp_1_2',0.05,'policy_tp_2_1',0.05, ... % Tr. prob. (off-diagonal)
   'phi1_policy_1',1.72,'phi1_policy_2',1.20,...  % state-specific coefficients
   'phi2_policy_1',0.49,'phi2_policy_2',0.20 );   % state-specific coefficients
```

In RISE, chain states are indexed numerically rather than by labels: state 1 corresponds to the hawkish (H) state and state 2 to the dovish (D) state. The numeric suffixes in parameter names (e.g., `_policy_1`, `_policy_2`), therefore, indicate which regime each coefficient belongs to.
The remainder of the driver file (solution, impulse responses, and simulation) remains unchanged.

3.2 Inspecting the Solution

With Markov switching in policy, the model's transition equation becomes regime–dependent:

$$\alpha_t^{(s_t)} \;=\; T^{(s_t)}\,\alpha_{t-1} \;+\; R^{(s_t)}\,\varepsilon_t, \tag{1.11}$$

where $\alpha_t = [\, r_t, y_t, \pi_t, c_t, \omega_t, \mu_t, \xi_t, z_t \,]'$ and $s_t \in \{H, D\}$ denotes the policy regime.

Running `driver_nk_ms.m` parameterizes and solves the model, printing the regime–specific solution to the MATLAB command window. The output of `print_solution(m)` now contains two blocks, one for each regime (see Box 4). Each block reports the state–space matrices $T^{(s_t)}$ and $R^{(s_t)}$ for the corresponding regime, confirming that the equilibrium mapping varies with the current state s_t, as intended.

Box 4 shows the solver `mfi` and two labeled sections, `policy = 1` and `policy = 2`, corresponding to the hawkish and dovish regimes, respectively.

Box 4: Regime–contingent solution printout, file `driver_nk_ms.m`

```
SOLVER :: mfi

Regime    1 : const = 1 & policy = 1
                 R           Y          Pai          C         Omega        Mu     Xi      Z
              ---------   ---------   --------   ---------   ----------   ----   ---   ---

    Mu{-1}      0.40859     -0.14425     1.1723     -0.8014     -0.14425    0.7     0     0
    ... other lines ...
    Ez          0.16956     0.075239    0.30555      0.418     0.075239      0     0   0.5
              ---------   ---------   --------   ---------   ----------   ----   ---   ---

                 R           Y          Pai          C         Omega        Mu     Xi      Z


Regime    2 : const = 1 & policy = 2
                 R           Y          Pai          C         Omega        Mu     Xi      Z
              ---------   ---------   --------   ---------   ----------   ----   ---   ---

    Mu{-1}      0.32799      -0.1061     1.3192     -0.58944     -0.1061    0.7     0     0
    ... other lines ...
    Ez          0.16626     0.12143     0.5562      0.67459     0.12143      0     0   0.5
              ---------   ---------   --------   ---------   ----------   ----   ---   ---

                 R           Y          Pai          C         Omega        Mu     Xi      Z
```

In addition to the printed solution, the driver script generates impulse-response plots—one per shock, each showing responses under both policy regimes—and prints basic simulation statistics to the screen.

3.3 What RISE Does Under the Hood (switching): The Three–matrix Form

Formally, the two-regime model generalizes the single-regime system (1.10) to

$$A_{-1}^{(s_t)}\alpha_{t-1} + A_0^{(s_t)}\alpha_t + A_{+1}^{(s_t)}\mathbb{E}_t\alpha_{t+1} + B^{(s_t)}\varepsilon_t = 0, \tag{1.12}$$

where the discrete state variable $s_t \in \{1, \ldots, h\}$ follows a Markov chain with transition matrix $Q = (q_{ij})$, $q_{ij} = \Pr(s_{t+1} = j \mid s_t = i)$, and each row suns to one. Under rational expectations, agents account for the possibility of regime changes, so expectations are taken over future regimes:

$$\mathbb{E}_t\alpha_{t+1} = \sum_{j=1}^{h} \Pr(s_{t+1} = j \mid s_t = i)\, \mathbb{E}_t\left[\alpha_{t+1} \mid s_{t+1} = j\right]. \tag{1.13}$$

RISE solves this system using the functional-iteration algorithm `mfi` (Maih, 2015). This algorithm produces regime-specific transition equations of the form (1.11), with matrices $T^{(s_t)}$ and $R^{(s_t)}$ that are consistent with rational expectations across regimes. The resulting solution is *mean-square stable* in the sense of Saito and Mitsui (1996) and do Valle Costa et al. (2005). Beyond the first-order (linear) case, RISE can apply perturbation techniques to obtain higher-order approximations, but the first-order component always retains the structure of (1.11).

4 From Model to Data: Filtering and Smoothing

Adding observables. For estimation, we must connect the model's *theoretical* variables—those determined within the structural system but not observed directly—to empirically measured data series. We do so by introducing additional variables that link the model to the data.

Suppose our data set contains quarterly observations on the annualized inflation rate π_t^{data}, the annualized short-term nominal rate r_t^{data}, and quarter-on-quarter output growth Δy_t^{data}, all reported in percent. A simple mapping is:

$$\pi_t^{\text{data}} = 4\pi_t, \tag{1.14}$$

$$r_t^{\text{data}} = 4r_t, \tag{1.15}$$

$$\Delta y_t^{\text{data}} = y_t - y_{t-1} + z_t. \tag{1.16}$$

This specification ensures that the model's theoretical variables π_t, r_t, and $y_t - y_{t-1} + z_t$ are expressed in percent units, consistent with the data. Consequently, the parameter values for the standard deviations of shocks introduced in Box 2 should also be interpreted in percent terms. Equations (1.14)–(1.16) are *regime-invariant* in this example; Markov switching in policy does not affect the mapping from model variables to data.

In a model file, observables must be (i) added as *additional* variables in the existing `@endogenous` block, (ii) listed in an `@observables` block, and (iii) defined by corresponding equations. If we add three observables, we must provide three equations. For empirical work, the model must also include at least as many exogenous shocks as observables to avoid stochastic singularity.[1]

Let us create a copy of the file `nk_ms.rs`, called `nk_ms_obs.rs`. In `nk_ms_obs.rs`, we make the following changes:

```
@endogenous
...,  % existing endogenous variables
Pai_obs, R_obs, Dy_obs % add these to the list

@observables % new block placed before @model
Pai_obs, R_obs, Dy_obs

@model
... % existing equations
Pai_obs{t} = 4*Pai{t};
R_obs{t}  = 4*R{t};
Dy_obs{t} = Y{t} - Y{t-1} + Z{t};
```

These equations bridge the model and the empirical data.

Working with artificial data. Before turning to empirical analysis with actual data, it is instructive to work with *artificial data*. Simulated samples allow us to run the full workflow—simulation, filtering, and smoothing—in an environment where the true latent states are known. We can therefore assess directly how accurately the filter recovers them.

To this end, we modify the driver as follows. First, we create a copy of `driver_nk_ms.m` named `driver_nk_ms_obs.m`. In the new file, we instruct RISE to load the new model `m = rise('nk_ms_obs.rs')`. We then remove commands that print the solution, plot impulse responses, or compute additional statistics, since these are not needed for the artificial-data exercise. We keep the simulation step (lines 12–13

of Box 2), because we must simulate the model once to obtain a complete history for all endogenous variables, including those that will later be treated as latent. Finally, we construct a separate database containing only the observables, to mimic the empirical situation in which an econometrician sees only a subset of variables.

The advantage of artificial data is that the full simulation stored in `mysim` provides a convenient "ground truth". This lets us compare recovered latent variables (such as the output gap ω_t) with their true simulated counterparts.

In `driver_nk_ms_obs.m`, we keep the simulation line and append the following commands:

```
rng(1234); nsim = 300; mysim = simulate(m,'simul_periods',nsim);
% Identify the declared observables
obsnames = get(m,'obs_list');
% Collect observables into a separate database
db = rmfield(mysim, setdiff(fieldnames(mysim), obsnames));
```

When the modified driver is run, the call to `simulate` behaves as described in Section 2.5, except that the model object `m` now represents a switching model. In addition to initializing the state vector and drawing exogenous shocks, `RISE` also generates a sequence of regimes $s_1, \ldots, s_T$ from the Markov chain with transition matrix $Q = (q_{ij})$. Once full histories are simulated, the list of observables is obtained via `get(m,'obs_list')` and used to extract them into the structure `db`. This separation mirrors the empirical setting: we observe inflation, output growth, and the policy rate, but not the level of output, the output gap, structural shocks, or the realized policy regime.

At this point, the modified driver provides all inputs required for filtering and smoothing. Before returning to the concrete `RISE` implementation in Section 4.3, we briefly review the theoretical foundations of filtering and smoothing in regime-switching state-space models.

4.1 Filtering

Filtering and smoothing are long-standing tools for extracting latent signals from noisy macroeconomic data. Early frequency–domain approaches, such as the Wiener–Kolmogorov and Butterworth filters (see the review in Pollock, 2026 in this volume), treated the task as optimal signal extraction, leading to widely used filters such as the Hodrick–Prescott and TRAMO–SEATS procedures. Modern model–based approaches, by contrast, formulate the problem in the time domain using state–space models and exploit recursive algorithms such as the Kalman filter and its extensions (see discussion in Proietti and Luati, 2026 in this volume). The routines implemented in `RISE` follow this state-space framework and extend it to nonlinear and Markov–switching DSGE settings.

Filtering algorithms are the backbone of empirical inference in DSGE analysis: they link the theoretical model to the observed data by updating estimates or conditional distributions of latent variables, regimes, and shocks as new observations arrive. Because they underpin likelihood evaluation, estimation, and forecasting, we briefly outline the key concepts and notations used later in the chapter. We proceed from the core ideas of state-space inference to the implementation in `RISE`, emphasizing its default Interacting Multiple Model (IMM) filter and the associated smoother.

Goal. In empirical macroeconomics, we rarely observe all variables that matter for a model's internal dynamics. Some variables—such as the output gap, random shocks, and expectations—are unobservable (latent) by construction. For others, including inflation, interest rates, or output growth, the available data provide only imperfect representations of the theoretical counterparts used in the model. Given a solved state–space model and a sample of observed data $\Psi_t = \{\psi_1, \ldots, \psi_t\}$, our task is to infer the latent

variables most likely to have generated the data and to evaluate how plausible these data are under a candidate parameter vector Θ.

Under Gaussian assumptions, the joint probability density $f(\Psi_t \mid \Theta)$ factorizes into one–step prediction densities:

$$f(\Psi_t \mid \Theta) = \prod_{\tau=1}^{t} f(\psi_\tau \mid \Psi_{\tau-1}, \Theta)\,, \tag{1.17}$$

which defines the (sample) *likelihood*. Its logarithm, $\log f(\Psi_t \mid \Theta) = \sum_{\tau=1}^{t} \log f(\psi_\tau \mid \Psi_{\tau-1}, \Theta)$, is our central measure of fit—the *log-likelihood*. Later, in estimation, this likelihood (or its posterior counterpart) will be maximized over Θ. For now, our goal is simply to compute this likelihood—and to do so efficiently—by avoiding the exponential growth in the number of mixture components, ensuring numerical stability, and recovering the conditional distributions that appear in each term of (1.17). The algorithm that performs these recursive computations of likelihood and conditional distributions as new observations arrive is called a *filter*.[2]

Objects of interest. Let α_t denote the unobserved state vector and $s_t \in \{1, \ldots, h\}$ the regime indicator at time t. A switching filter recursively computes, for each t, the following objects:

the *predicted* state vector mean and covariance (best forecast given Ψ_{t-1}),

$$\alpha_{t|t-1} := \mathbb{E}[\alpha_t \mid \Psi_{t-1}], \quad P_{t|t-1} := \mathbb{E}\big[(\alpha_t - \alpha_{t|t-1})(\alpha_t - \alpha_{t|t-1})' \,\big|\, \Psi_{t-1}\big]\,,$$

the *updated* state vector mean and covariance (after seeing ψ_t),

$$\alpha_{t|t} := \mathbb{E}[\alpha_t \mid \Psi_t], \quad P_{t|t} := \mathbb{E}[(\alpha_t - \alpha_{t|t})(\alpha_t - \alpha_{t|t})' \mid \Psi_t],$$

and, in the regime–switching case, the *updated regime probabilities*

$$\mu^{j}_{t|t} := \Pr[s_t = j \mid \Psi_t], \qquad j = 1, \ldots, h.$$

From these objects, the filter also delivers the predictive density $f(\psi_t \mid \Psi_{t-1}, \Theta)$ which appears in (1.17), so the cumulative log–likelihood can be computed sequentially.

Extension to multiple regimes. If the model has a single regime ($h = 1$), the Gaussian filter reduces to the standard Kalman filter, which is exact and optimal for linear systems (Kalman, 1960). In a multiple-regime model with Markov switching, however, the conditional law of (α_t, s_t) given Y_t is a mixture of Gaussian densities whose number of components grows exponentially with t, making exact evaluation infeasible. To keep computations tractable, the default routine in RISE uses the *Interacting Multiple Model (IMM)* approximation (Blom, 1984; Blom and Bar-Shalom, 1988). The IMM approximates the full mixture by retaining a single Gaussian density for each regime at every time step. At each iteration, it performs three operations—*mixing*, *prediction*, and *update*—so that the filter evolves forward in time without the number of Gaussian components exploding.

From likelihood to a working recursion (IMM). Let $Q_{ij} = \Pr(s_t = j \mid s_{t-1} = i)$ be the Markov transition matrix, and let each regime-specific model at time t be linear Gaussian:[3]:

$$\begin{aligned} \alpha_t &= c_{\alpha,j,t} + T_{j,t}\,\alpha_{t-1} + R_{j,t}\,\varepsilon_t, \quad \varepsilon_t \sim \mathcal{N}(0, I), \\ \psi_t &= c_{\psi,j,t} + Z_{j,t}\,\alpha_t + g_{j,t}\,\eta_t, \quad \eta_t \sim \mathcal{N}(0, I). \end{aligned}$$

Collect the parameters into $\mathcal{M}_{j,t} \equiv \{c_{\alpha,j,t}, T_{j,t}, R_{j,t};\ c_{\psi,j,t}, Z_{j,t}, g_{j,t}\}$. Let $\mathcal{K}(\cdot)$ denote the standard *one–step Kalman update* for a single regime:

$$\left(\alpha_{t|t-1},\ P_{t|t-1},\ \alpha_{t|t},\ P_{t|t},\ \Lambda_t\right) = \mathcal{K}(\mathcal{M}_t; \alpha_{t-1|t-1}, P_{t-1|t-1}, \psi_t),$$

which returns the predicted and updated moments together with the predictive density contribution $\Lambda_t \equiv f(\psi_t \mid \Psi_{t-1}, \Theta)$ from decomposition (1.17).

IMM recursion.

Initialization at $t-1$. For each regime i we have

$$\alpha^i_{t-1|t-1}, \quad P^i_{t-1|t-1}, \quad \mu^i_{t-1|t-1}.$$

1) Mixing of initial conditions. Compute the conditional mixing weights:

$$\mu^{i|j}_{t-1|t-1} \;=\; \frac{Q_{ij}\,\mu^i_{t-1|t-1}}{\sum_{k=1}^h Q_{kj}\,\mu^k_{t-1|t-1}}.$$

Form regime-j initial conditions by averaging over i, thus reducing the dimensionality:

$$\alpha^{0j}_{t-1|t-1} = \sum_{i=1}^h \mu^{i|j}_{t-1|t-1}\,\alpha^i_{t-1|t-1},$$

$$P^{0j}_{t-1|t-1} = \sum_{i=1}^h \mu^{i|j}_{t-1|t-1}\left(P^i_{t-1|t-1} + (\alpha^i_{t-1|t-1} - \alpha^{0j}_{t-1|t-1})(\alpha^i_{t-1|t-1} - \alpha^{0j}_{t-1|t-1})'\right).$$

2) Regime–specific prediction and update (Kalman one–step). For each regime j,

$$\left(\alpha^j_{t|t-1},\ P^j_{t|t-1},\ \alpha^j_{t|t},\ P^j_{t|t},\ \Lambda^j_t\right) = \mathcal{K}\left(\mathcal{M}_{j,t}; \alpha^{0j}_{t-1|t-1}, P^{0j}_{t-1|t-1}, \psi_t\right).$$

3) Regime-probability update. Apply Bayes' rule:

$$\mu^j_{t|t} \;=\; \frac{\Lambda^j_t \sum_{i=1}^h Q_{ij}\mu^i_{t-1|t-1}}{\sum_{k=1}^h \sum_{m=1}^h \Lambda^m_t Q_{km}\mu^k_{t-1|t-1}}.$$

The denominator is the *likelihood increment*

$$\ell^*_t \;=\; \sum_{k=1}^h \sum_{m=1}^h \Lambda^m_t\, Q_{km}\, \mu^k_{t-1|t-1},$$

and the corresponding log-likelihood contribution is $\ell_t = \log(\ell^*_t)$ so that the sample log–likelihood in (1.17) is $\log(f(\Psi_t \mid \Theta)) = \sum_{t=1}^T \ell_t$.

4) Combination of regime-specific results. Once we know regime-specific *updated variables*, we compute *expected updated variables* by weighting them with regime probabilities $\mu^j_{t|t}$:

$$\alpha_{t|t} \;=\; \sum_{j=1}^h \mu^j_{t|t}\,\alpha^j_{t|t}, \qquad P_{t|t} \;=\; \sum_{j=1}^h \mu^j_{t|t}\left(P^j_{t|t} + (\alpha^j_{t|t} - \alpha_{t|t})(\alpha^j_{t|t} - \alpha_{t|t})'\right).$$

5) Forward recursion. Set the updated quantities $\{\alpha^j_{t|t}, P^j_{t|t}, \mu^j_{t|t}\}$ as the initial values for the next step $t+1$ and return to step 1.

To summarize, the IMM is a *forward recursion* that sequentially moves through the sample period. At each time t, it mixes regime-specific priors, performs prediction and updating for each regime, and re-weights the regimes by their posterior probabilities. By keeping only h Gaussian components—one per regime—it provides a computationally efficient approximation to the exact mixture filter, whose number of components would otherwise grow exponentially with t. When $h = 1$, the IMM coincides with the standard Kalman filter.

4.2 Smoothing

A filter delivers *updated* state variables $\alpha_{t|t}$: the best estimate of the latent states given information available up to time t. After filtering is completed and the full sample Ψ_T is available, these estimates can be refined by incorporating future observations from $t+1$ to T.

A *smoother* performs this refinement through a *backward recursion*, producing the *smoothed* estimates $\alpha_{t|T} = \mathbb{E}[\alpha_t \mid \Psi_T]$. Running backward from the end of the sample, each state vector estimate is adjusted according to how well it predicted the next observation: if the forecast error at $t+1$ was positive, the smoother raises the earlier estimate of α_t; if negative, it lowers it. This backward correction reduces variance and mitigates distortions caused by initialization.

For regime probabilities, the classic reference is Kim (1994), whose backward recursion yields smoothed regime probabilities $\mu_{t|T}(j) = \Pr(s_t = j \mid \Psi_T)$. For the state vector, Hashimzade et al. (2024) describe a numerically stable switching smoother that computes $\mathbb{E}[\alpha_t \mid \Psi_T]$. Both smoothers are implemented in `RISE` and are automatically called when the default setting of `filter` is used.

4.3 Filtering and Smoothing in `RISE`

Purpose of the example. Box 5 reproduces the complete driver file `driver_nk_ms_obs.m`, which was introduced earlier in the chapter. In the preceding discussion, individual commands (`rise`, `set`, `solve`, `simulate`, and `filter`) were explained in isolation; here we bring them together in a single, self-contained script.

Lines 1–11 perform the core steps—model parsing, parameterization, solution, simulation, and filtering—while lines 12–35 illustrate how to extract, visualize, and evaluate filtered and smoothed objects. Although the plotting commands are deliberately simple, they demonstrate how to access key elements of the filter output (such as `Expected_updated_variables` and `smoothed_state_probabilities`) and how to compute basic accuracy measures. Readers are encouraged to inspect the structure `myfilt` in MATLAB to explore additional outputs, including regime-specific quantities.

Running the filter. With a solved switching model and a database `db` built from simulated observables, the default filter is called by

```
[myfilt, loglik] = filter(m,'data',db);
```

see line 11 in Box 5. The resulting structure `myfilt` contains, among other fields, (i) expected *updated* and *smoothed* state vectors, and (ii) *updated* and *smoothed* state probabilities for each regime.

Assessing accuracy: RMSE and smoothing gains. In this model, the output gap ω_t is the only latent state variable. Because the true simulated series is available, we can compare (i) the true values, (ii)

their updated estimates, and (iii) their smoothed estimates. The same comparison can be made for the probability of being in the hawkish regime. Accuracy is summarized by the root–mean–square error (RMSE) for a latent variable α_t observed over T periods

$$\mathcal{R}_{t|s} \;=\; \sqrt{\frac{1}{n}\sum_{t=1}^{n}\left(\alpha_t - \alpha_{t|s}\right)^2},$$

where $s = t$ gives the update error and $s = T$ the smoothing error. The proportional gain from smoothing is defined as Gain $= 1 - \frac{\mathcal{R}_{t|T}}{\mathcal{R}_{t|t}}$, a concise measure of how much smoothing improves over updating.

Box 5 shows how to retrieve the relevant objects (lines 12-19), plot them (lines 20-26), and compute and print the statistics (lines 27-35).

Box 5: Filtering and plotting results, computing RMSEs. File `driver_nk_ms_obs.m`

```
 1 clear all; close all; clc;
 2 m = rise('nk_ms_obs.rs');
 3 p = struct('beta',1/(1+0.706/400),'sigma',2.9,'varphi',2.5,'theta',0.82,'gamma',0.77, ...
 4 'zeta',0.10,'rho_z',0.90,'rho_mu',0.70,'rho_xi',0.80,'rho_r',0.79,'sig_z',0.50, ...
 5 'sig_mu',0.15,'sig_xi',0.10,'sig_r',0.10,'policy_tp_1_2',0.05,'policy_tp_2_1',0.05, ...
 6 'phi1_policy_1',1.72,'phi1_policy_2',1.20,'phi2_policy_1',0.49,'phi2_policy_2',0.20 );
 7 m = set(m,'parameters',p); m = solve(m); print_solution(m);
 8 rng(1234); nsim = 300; mysim = simulate(m,'simul_periods',nsim);
 9 obsnames = get(m,'obs_list');
10 db = rmfield(mysim, setdiff(fieldnames(mysim), obsnames));
11 [myfilt, loglik] = filter(m,'data',db);
12      % True vs updated vs smoothed (omega and hawkish probability)
13 t         = 1:nsim;
14 gap_data  = mysim.Omega.data;
15 gap_upd   = myfilt.Expected_updated_variables.Omega.data;
16 gap_smo   = myfilt.Expected_smoothed_variables.Omega.data;
17 hawk_data = 2 - mysim.policy.data;  % recode so 1 = hawkish
18 hawk_upd  = myfilt.updated_state_probabilities.policy_1.data;
19 hawk_smo  = myfilt.smoothed_state_probabilities.policy_1.data;
20 figure('Name','Results','Color','w');
21 subplot(2,1,1)
22 plot(t,gap_data,'k-',t,gap_upd,'k-.',t,gap_smo,'k--','LineWidth',1);title('A: Output Gap');
23 subplot(2,1,2)
24 plot(t,hawk_data,'k-',t,hawk_upd,'k-.',t,hawk_smo,'k--','LineWidth',1);
25 title('B: Probability to be in Hawkish policy state');
26 legend('data','updated variables','smoothed variables','Location','best');
27      %--- RMSEs and smoothing gains ---
28 gap_upd_RMSE  = sqrt(mean((gap_upd - gap_data).^2));
29 gap_smo_RMSE  = sqrt(mean((gap_smo - gap_data).^2));
30 gap_gain      = (1 - gap_smo_RMSE/gap_upd_RMSE)*100;
31 hawk_upd_RMSE = sqrt(mean((hawk_upd - hawk_data).^2));
32 hawk_smo_RMSE = sqrt(mean((hawk_smo - hawk_data).^2));
33 hawk_gain     = (1 - hawk_smo_RMSE/hawk_upd_RMSE)*100;
34 fprintf('GAP : RMSE upd=%.4f, smo=%.4f, gain=%.1f%%\n', gap_upd_RMSE, gap_smo_RMSE, gap_gain);
35 fprintf('Hawk: RMSE upd=%.4f, smo=%.4f, gain=%.1f%%\n', hawk_upd_RMSE, hawk_smo_RMSE, hawk_gain);
```

Latent variables and regimes. Figure 1 compares the true values (solid line) with the updated (dash-dotted line) and smoothed (dashed line) estimates. Panel A shows that the filter tracks ω_t closely; but smoothing still yields a modest improvement: RMSE declines from 0.0348 to 0.0315, a gain of around

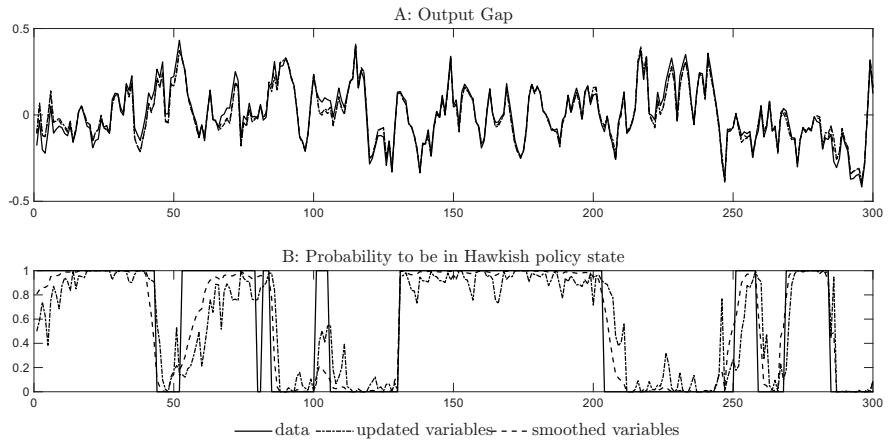

Figure 1: Updating and smoothing. 300 observations of artificial data.

9.7%. Panel B displays regime inference: updated probabilities respond quickly but remain noisy near switches, whereas smoothed probabilities use future information to refine inference, reducing RMSE from 0.2899 to 0.2072 (a gain of roughly 28.5%). Gains of this order are meaningful: in a model where the filter is already performing well, a 10–20% reduction in RMSE is sizable. For comparison, Hashimzade et al. (2024) report average gains of around 25% in a richer model. Overall, the IMM filter performs nearly optimally in simple settings, yet smoothing still delivers sizable gains in accuracy.

5 Empirical Analysis

In this section, we illustrate how the filtering–based methods introduced above apply to a typical empirical setting. The example follows a well–established line of research that seeks to disentangle the effects of exogenous disturbances from those of monetary policy actions—the distinction between "good luck" and "good policy"—in explaining macroeconomic outcomes (Lubik and Schorfheide, 2005; Sims and Zha, 2006; Leith et al., 2025). Using U.S. data over the last seventy years, we estimate a standard specification in which both the volatility of shocks and the stance of monetary policy can switch between regimes. To demonstrate the workflow, we employ the same linear NK model (1.1)–(1.8), extended to include two independent Markov chains—one governing policy behavior and the other governing volatility.

Before turning to estimation, we briefly recall the key elements of the theoretical framework that underpins this empirical analysis.

5.1 Posterior Kernel Maximization

Maximum likelihood versus Bayesian estimation. From Section 4 we know how to evaluate the likelihood $p(\Psi_T \mid \Theta)$—the probability of observing the dataset Ψ_T conditional on the parameter vector Θ. In the classical maximum likelihood estimation (MLE), we choose the parameter vector that maximizes this likelihood:

$$\hat{\Theta}_{\text{MLE}} = \arg\max_{\Theta} \; p(\Psi_T \mid \Theta).$$

In practice, researchers often have prior knowledge about some parameters before observing the data.

For instance, we may know that the discount factor β is close to one, or that a persistence parameter in a shock process should lie between zero and one. In Bayesian estimation (BE), such information is formalized as a *prior distribution* over parameters, denoted $p(\Theta)$. Combining the prior with the likelihood gives the *joint* probability of data and parameters:

$$p(\Psi_T, \Theta) = p(\Psi_T \mid \Theta)\, p(\Theta).$$

The expression on the right-hand side is called the *posterior kernel*.

Maximizing this kernel with respect to Θ yields the *posterior mode*—the Bayesian analog of the maximum likelihood estimate.

The posterior kernel is maximized numerically. Because it is a multivariate, nonlinear function of Θ, optimization typically relies on second-order (Newton-type) methods.The Hessian matrix of second derivatives summarizes the curvature of the kernel around the optimum. Once the optimal parameter vector $\hat{\Theta}$ is found, the inverse of the negative Hessian provides an approximation to the covariance matrix of the estimates. Intuitively, a sharply peaked posterior kernel corresponds to small standard errors, while a flatter peak implies greater uncertainty.

Beyond estimating parameters, we often want a measure of how well the model explains the observed data. In Bayesian analysis, this role belongs to the *marginal data density* (MDD), which answers the question: what is the probability that the observed data were generated by the model, *without conditioning on any specific value of* Θ? Formally, this involves integrating the posterior kernel over the entire parameter space:

$$p(\Psi_T) = \int p(\Psi_T \mid \Theta)\, p(\Theta)\, d\Theta.$$

The result is a single number summarizing the model's overall fit—it averages the likelihood across all parameter values, weighted by their prior plausibility. The MDD does not depend on the optimal parameter vector $\hat{\Theta}$; rather, it reflects how well the model-prior combination accounts for the data. Because there is no absolute benchmark for its magnitude, the MDD is used primarily for *relative* model comparison: the ratios of MDDs across competing models are known as *Bayes factors*, the standard criterion for Bayesian model selection.[4]

Having outlined the concepts, we now move to implementation. Estimating a model in RISE requires four ingredients: a model specification, a database of observed series (our Ψ_T), a set of priors $p(\Theta)$ reflecting prior knowledge, and an estimation routine that searches for the posterior mode. The remainder of this section describes how to specify these elements within the RISE workflow and illustrates the resulting estimation output.

Model file adjustments. In addition to the Markov chain that governs policy states (Hawk and Dove, $s_t \in \{H, D\}$), we introduce a separate Markov chain v_t that governs shock volatility, alternating between a *Calm* state (C) with low volatility and a *Turbulent* state (T) with high volatility ($v_t \in \{C, T\}$).

To adjust the model file, we create a copy of the `nk_ms_obs.rs` file, named `nk_ms_est.rs`. We then amend the list of parameters as follows:

```
@parameters
beta, sigma, varphi, theta, gamma, zeta, rho_z, rho_mu, rho_xi, rho_r,sig_r,
policy_tp_1_2, policy_tp_2_1,vol_tp_1_2, vol_tp_2_1

@parameters(policy,2) phi1, phi2

@parameters(vol,2) sig_z, sig_mu, sig_xi
```

Thus, in addition to the chain labeled `policy`, which controls the monetary policy parameters, we also introduce a second chain labeled `vol`, which governs the standard deviations of the three structural shocks. The transition–probability parameters for this new chain (`vol_tp_1_2` and `vol_tp_2_1`) are also added to the main parameter list.

The `@model` block itself remains unchanged; RISE automatically applies the appropriate state-specific values of the switching parameters for each chain's current state.

Data preparation. We collect quarterly data on inflation, the interest rate, and output growth from the FRED database,[5] store them in an Excel file `data_US.xlsx` with dates in the first column and observables in the following columns:

```
observation_date    R_obs    Pai_obs    Dy_obs
1954-07-01          1.0300   -0.0124    1.1297
1954-10-01          0.9900   -0.4959    1.9570
1955-01-01          1.3400   -0.5936    2.8566
1955-04-01          1.5000   -0.5698    1.6275
...                   ...       ...       ...
```

Priors. In Bayesian estimation, priors $p(\Theta)$ are specified for the parameters to be estimated. To keep the driver script concise, we define them in a separate file, `create_priors.m` shown in Box 6, which builds a structure named `priors`. Each field of this structure is a cell array with six entries: {initial value, mean, standard deviation, distribution, lower bound, upper bound}. The first entry provides the starting value used by the optimizer or filter; the second, third, and fourth entries specify the prior mean, standard deviation, and distribution family; the last two define the admissible range for the parameter. For example, line 4 in Box 6 means that the intertemporal elasticity parameter σ is initialized at 3.0, has a Normal prior centered at 2.5 with standard deviation 0.25, and is restricted to the range 0.5 to 5.[6]

When coding, we use numerical indices rather than letter labels to distinguish the states of each Markov chain. For the `policy` chain, index 1 corresponds to the *Hawkish* regime and index 2 to the *Dovish* one. In the volatility (`vol`) chain, index 1 denotes the *Calm* state with low volatility, and index 2 denotes the *Turbulent* state with high volatility. This convention carries through to parameter naming: for example, `sig_z_vol_1` and `sig_z_vol_2` represent the standard deviation of the technology shock under Calm and Turbulent conditions, respectively. Similarly, `phi1_policy_1` and `phi1_policy_2` correspond to the Taylor rule inflation response coefficient in the hawkish and dovish regimes, respectively. The priors in Box 6 are set to reflect these economic distinctions: the hawkish regime places a strong response to inflation (with prior mass above one), whereas the dovish regime restricts the coefficient to values below one. For the volatility chain, priors assign smaller variances in the Calm state and larger ones in the Turbulent state.

In some applications, we may fix specific parameters at calibrated values. Such parameters do not require priors. Derived parameters defined within the `@model` block adjust automatically whenever their underlying parameters change.

Driver file adjustments. With the updated model file, the Excel data file, and the MATLAB priors file in place, we now turn to the driver script used for estimation. Box 7 presents the complete estimation driver. Rather than modifying a specific earlier box line by line, this script builds on the fully developed drivers introduced in the previous sections and incorporates the additional elements required for estimation with switching volatility.

The changes relative to the earlier simulation and filtering drivers are as follows. First, we add the transition–probability parameters for the volatility chain, `vol_tp_1_2` and `vol_tp_2_1`, to the parameter structure `p` (line 5). Second, the volatility parameters `sig_z`, `sig_mu`, and `sig_xi` are redefined as switching parameters by appending the appropriate regime suffixes (lines 7–8). Third, the data are read from an Excel file (lines 10–11)[7] and demeaned in line 12,[8] before being assembled into a time–series database `db` on lines 13–16. Fourth, we include a simple diagnostic plot of the demeaned observables (lines 17–18) to verify that the imported series behave as expected. Finally, the priors structure is constructed in line 19, completing the setup for estimation.

Box 6: Defining priors in `create_priors.m`

```
 1 function priors = create_priors(model)
 2 priors = struct();
 3 % --- Structural parameters (Chen et al. 2017, Table 1) ---
 4 priors.sigma  = {3.2512, 2.50, 0.15, 'normal',  0.5,  5};        % inv. IES
 5 priors.varphi = {2.6795, 2.50, 0.15, 'normal',  0.5,  5};        % inv. Frisch
 6 priors.theta  = {0.27175, 0.25, 0.05, 'beta',   0.01, 0.999};    % habit
 7 priors.gamma  = {0.74198, 0.75, 0.02, 'beta',   0.01, 0.999};    % Calvo
 8 priors.zeta   = {0.93025, 0.75, 0.05, 'beta',   0.01, 0.999};    % indexation
 9 % --- AR(1) coefficients ---
10 priors.rho_xi = {0.91315, 0.50, 0.15, 'beta', 0.01, 0.999};
11 priors.rho_mu = {0.17067, 0.50, 0.15, 'beta', 0.01, 0.999};
12 priors.rho_z  = {0.6328,  0.50, 0.15, 'beta', 0.01, 0.999};
13 priors.rho_r  = {0.836,   0.50, 0.15, 'beta', 0.01, 0.999};
14 % --- Shock volatilities (switching) ---
15 priors.sig_xi_vol_1 = {0.97639, 0.50, 0.50, 'igamma1', 1e-4, 10};
16 priors.sig_xi_vol_2 = {4.7282,  2.00, 0.50, 'igamma1', 1e-4, 10};
17 priors.sig_mu_vol_1 = {0.0858,  0.25, 0.50, 'igamma1', 1e-4, 10};
18 priors.sig_mu_vol_2 = {0.76634, 1.00, 0.50, 'igamma1', 1e-4, 10};
19 priors.sig_z_vol_1  = {0.33166, 0.50, 0.50, 'igamma1', 1e-4, 10};
20 priors.sig_z_vol_2  = {1.0837,  1.50, 0.50, 'igamma1', 1e-4, 10};
21 priors.sig_r        = {0.18226, 0.25, 0.05, 'igamma1', 1e-4, 10};
22 % --- Policy rule coefficients (switching) ---
23 priors.phi1_policy_1 = {1.8601,  2.00, 0.15, 'normal', 0.001, 10};
24 priors.phi1_policy_2 = {0.34163, 0.50, 0.50, 'normal', 0.001, 10};
25 priors.phi2_policy_1 = {0.85946, 0.50, 0.15, 'beta',   0.01, 0.999};
26 priors.phi2_policy_2 = {0.35774, 0.50, 0.15, 'beta',   0.01, 0.999};
27 % --- Transition probabilities (off-diagonals) ---
28 priors.policy_tp_1_2 = {0.061718, 0.10, 0.05, 'beta', 0.001, 0.999};
29 priors.policy_tp_2_1 = {0.18273,  0.10, 0.05, 'beta', 0.001, 0.999};
30 priors.vol_tp_1_2    = {0.0260,   0.10, 0.05, 'beta', 0.001, 0.999};
31 priors.vol_tp_2_1    = {0.2013,   0.10, 0.05, 'beta', 0.001, 0.999};
```

With priors and data in place, estimation reduces to a single call to `estimate` (lines 20-23). At a minimum, we supply the model and the data; `RISE` then uses the default settings for the filter, sample period, and optimization routine. In our example, explicit options are included for clarity but are not strictly necessary.

Conceptually, the `estimate` function evaluates the likelihood via the filter, combines it with the priors to form the posterior kernel, and searches for its mode using the specified optimizer. The function returns a new model object `me`, parameterized at the posterior mode and augmented with estimation results stored under `me.estimation`, together with a `filtration` structure containing the likelihood contributions and smoothed series. The original model object `m` remains unchanged, allowing direct comparison between pre- and post-estimation versions.

The printed output includes a table of posterior modes and their associated standard deviations, based on a second-order approximation to the posterior kernel's curvature, along with summary statistics such as the log-likelihood and the Laplace approximation of the marginal data density.

Finally, because estimation can be time–consuming, we save results for later use (lines 25–26 in Box 7). The automatic timestamp in the filename makes it easy to distinguish between runs and to retrieve results for subsequent analysis or MCMC simulation.

Box 7: Estimation driver `driver_nk_ms_est.m`

```
clear all; close all; clc;
m = rise('nk_ms_est.rs');
p = struct('beta',1/(1+0.706/400),'sigma',2.9,'varphi',2.5,'theta',0.82,'gamma',0.77, ...
  'zeta',0.10,'rho_z',0.90,'rho_mu',0.70,'rho_xi',0.80,'rho_r',0.79,'sig_r',0.10, ...
  'policy_tp_1_2',0.05,'policy_tp_2_1',0.05,'vol_tp_1_2',0.05,'vol_tp_2_1',0.05, ...
  'phi1_policy_1',1.72,'phi1_policy_2',1.20,'phi2_policy_1',0.49,'phi2_policy_2',0.20,...
  'sig_z_vol_1',0.50,'sig_z_vol_2',1.00,'sig_mu_vol_1',0.15,'sig_mu_vol_2',0.30,...
  'sig_xi_vol_1',0.10,'sig_xi_vol_2',0.20);
m = set(m,'parameters',p); m = solve(m); print_solution(m);
data   = readmatrix('data_US.xlsx','Sheet',1,'Range','B2:D285');
vnames = readcell('data_US.xlsx','Sheet',1,'Range','B1:D1');
data = data-mean(data,1);
start  = '1954Q3'; db = struct();
for iv=1:numel(vnames)
  db.(vnames{iv}) = ts(start,data(:,iv));
end
figure('name','Observables') % quick plot of observables
for ii=1:numel(vnames) subplot(3,1,ii); plot(db.(vnames{ii})); title(vnames{ii}); end
priors = create_priors(m);          % Load priors structure23
[me, filtration] = estimate(m, 'data', db, 'estim_priors', priors, ...
  'estim_start_date', '1954Q3', ...  % Optional: restrict sample start, i.e. exclude pre-1980s
  'estim_end_date',   '2025Q2', ...  % Optional: restrict sample end, i.e. exclude pandemics
  'optimizer', 'fmincon');           % Optional: choose optimizer
     % Save estimation results
rndName = ['Estimation_NKUS5425_',replace(char(datetime("now")),{'-',':',' '},{'_','_','_'})];
pmode = get(me,'mode'); save(rndName,'pmode','priors','me','filtration','m','db');
```

Inspecting results. Suppose that the file `Estimation_NKUS5425_15_Oct_2025_19_23_31.mat` stores the results of the estimation. The short script in Box 8 illustrates how to visualize and further analyze these results.

Line 2 calls `plot_probabilities(me)`, which generates a basic but informative plot of regime and state probabilities for both chains. The same line also invokes `print_estimation_results(me)`, which reproduces the table of posterior modes and standard deviations obtained during estimation. Line 3 re-runs the filter using the estimated model object `me`. Since `me` already stores both the parameter values and the database used for estimation, there is no need resupply them as options.

Box 8: Estimation results with `analyze_estimation_results.m`

```
clear all; load('Estimation_NKUS5425_20_Jan_2026_15_29_36');
plot_probabilities(me); print_estimation_results(me);
[myfilt_e, LogLik_e] = filter(me);  % The model is parameterized by estimated parameters
```

Figure 2 is obtained by adapting the plotting code in Box 5 from line 15 onward, replacing the filter output with `myfilt_e` and accounting for the additional volatility chain.[9]

Unlike in the artificial–data experiments, Figure 2 contains no "true" values for latent variables or regimes, so it cannot be used to assess filtering accuracy directly. Instead, the figure invites historical interpretation. The dovish policy state is assigned a high probability during the Great Inflation of the 1970s, the Global Financial Crisis of 2008, and the Cost-of-Living crisis following the COVID–19 pandemic in 2020-21. High–volatility regimes appear around the 1981 recession, the Great Recession of 2008, and the pandemic recession of 2020. These patterns are consistent with the findings of Chen et al. (2017) and Leith et al. (2025), who estimated similar models with comparable specifications. The ability to replicate these results with a short, general-purpose script—rather than lengthy model-specific code—demonstrates both the reliability and accessibility of RISE for applied macroeconomic analysis.

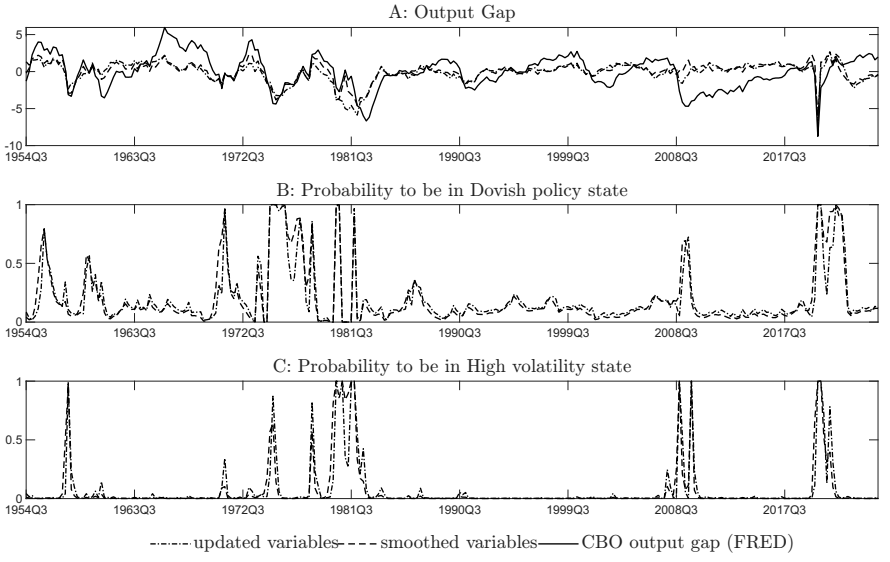

Figure 2: Estimated variables and state probabilities.

The top panel of Figure 2 compares the estimated output gap with the Congressional Budget Office's measure (GDPC1_GDPPOT) from FRED. Despite the model's simplicity, the two series exhibit a correlation of roughly 0.55—a strong correspondence given their entirely different construction methods. This should not be interpreted as structural validation, but rather as evidence that even a compact New Keynesian model with regime switching can meaningfully capture cyclical macroeconomic dynamics. The ease with which these results are obtained highlights the practical value of such models and the convenience of the RISE toolbox for empirical research.

5.2 Sampling the Posterior with MCMC

Maximizing the posterior kernel, as we did in the previous section, is a natural first step. It yields the *posterior mode* $\hat{\Theta}$ and, through a local second–order approximation, provides an estimate of curvature and hence approximate standard errors. However, such a single point estimate gives only a narrow view of parameter uncertainty. In practice, policymakers and researchers are interested not in the most likely parameter values but also in how model outcomes vary across other plausible configurations. Posterior distributions provide precisely this information: applied analyses—such as impulse responses, multipliers, and counterfactual simulations—depend on understanding the entire distribution, not merely its peak.

The next step, therefore, is to approximate the full shape of the posterior kernel rather than a quadratic expansion around its maximum. A convenient way to achieve this is by stochastic simulation. By generating many points in the parameter space, with frequencies proportional to their posterior probabilities, we obtain a numerical representation of the posterior distribution.

Among such simulation methods—collectively known as *Markov Chain Monte Carlo* (MCMC)—the Metropolis–Hastings (MH) algorithm is the workhorse. It constructs a Markov chain that "walks" through the parameter space, proposing new candidate values of Θ and accepting them with probabilities that ensure the chain spends proportionally more time in regions of high posterior density. In doing so, the algorithm not only identifies the mode but also traces the surrounding landscape, turning the abstract posterior into a set of usable parameter draws.

Although the initial mode search could be skipped, in practice, it is an essential step. It provides an efficient starting point: without it, an MH chain initialized at a random location may spend many iterations wandering in low-probability regions before reaching the relevant part of the posterior. Starting the sampler near the mode ensures that MCMC efficiently explores the posterior and delivers a reliable representation of uncertainty around the estimated parameters.

Metropolis-Hastings algorithm. At an intuitive level, the Metropolis–Hastings sampler lets a Markov chain explore the parameter space through successive proposals. At each iteration, a new candidate point is drawn from a *proposal distribution*. If the candidate has a higher posterior value than the current point, it is automatically accepted; if lower, it is accepted only with a probability proportional to the ratio of posterior densities. This acceptance rule allows the chain to spend most of its time in high-density regions while occasionally moving into lower-density areas to ensure thorough exploration. Over many iterations, the accepted points form a sample that approximates the posterior distribution.[10]

In practice, the Metropolis–Hastings algorithm must be tuned carefully. The proposal distribution should be neither too narrow nor too wide: a chain that accepts nearly all proposals moves too slowly, while one that rejects most proposals explores too little of the space. A common rule of thumb is to target an acceptance rate between 20% and 40%, with values near 30% typically performing well in medium-sized DSGE models. Achieving this rate usually requires some preliminary tuning of the proposal covariance matrix before running the full sampler (see Section 7).

Running Metropolis–Hastings in `RISE`. Having motivated the need for MCMC, we now demonstrate how to run the MH sampler in `RISE`. The example below assumes that the model has already been estimated and saved, so that the sampler can be initialized close to the posterior mode obtained earlier.

Box 9: Markov Chain Monte Carlo with `driver_mcmc.m`

```
1 clear all; load('Estimation_NKUS5425_20_Jan_2026_15_29_36')
2 [objective,lb,ub,mu,SIG]=pull_objective(me,...
3     'solve_check_stability',false,'fix_point_TolFun',1e-6);
4 scale=0.15;
5 myOpts=struct(); myOpts.tunedCov=scale*SIG;myOpts.nchain=2; myOpts.N=100000;
6 energy=@(varargin)-objective(varargin{:});
7 results=sample(rsamplers.rwmh(energy,mu,lb,ub,myOpts));
8 mddobj=mdd(results,energy,lb,ub,[],[],true); mdd_bridge = bridge(mddobj,true);
9 rndName1=['MCMC_NKUS5725_',replace(char(datetime("now")),{'-',':',' '},{'_','_','_'})];
10 save(rndName1,'pmode','priors','me','filtration','m','db','results','objective',...
11   'lb','ub','mdd_bridge');
```

Line 1 loads the saved estimation file, which contains the posterior mode and related outputs. Lines 2–3

call pull_objective, a convenience function that reconstructs the posterior kernel in a form suitable for MCMC sampling. It returns (i) a function handle evaluating the (negative) log-posterior, objective, (ii) lower and upper bounds on parameters, lb and ub, (iii) the posterior mode (starting point for the chain) mu, and (iv) an estimated covariance matrix SIG (typically the inverse of the Hessian at the mode). All of these are derived automatically from the estimated model object me, so no additional coding is required.

Lines 4–5 define the distribution of the proposal for the sampler. The covariance matrix SIG is scaled by a factor scale to control step size, while myOpts.N sets the total number of draws and myOpts.nchain specifies the number of parallel chains. Because the sampler operates on an *energy function*—the negative of the log-posterior—line 6 reverses the sign of the objective. Line 7 then calls the random–walk Metropolis–Hastings sampler (rwmh) through the general-purpose function sample, which generates a Markov chain of posterior draws stored in the structure results.

Line 8 computes marginal data density using Meng and Wong (1996) bridge sampling method; see the manual for alternative methods.

Finally, lines 9–11 save the sampler output, automatically appending a timestamp to the filename for traceability. The stored object results contains the complete set of posterior draws and related diagnostics, which can then be used to compute posterior means, standard deviations, and credible intervals, or to propagate parameter uncertainty into impulse responses, forecasts, and counterfactual policy experiments. In essence, the MCMC procedure replaces the local quadratic approximation with a numerical representation of the posterior distribution itself.

Diagnostics for MCMC output. After running the Metropolis–Hastings sampler, the first task is to check whether the chains have converged and whether they provide a reliable representation of the posterior distribution. RISE includes a set of standard diagnostic tools, accessible through the mcmc command. The code in Box 10 illustrates a minimal workflow.

Box 10: MCMC diagnostics in RISE with analyze_mcmc_diagn.m

```
clear all; load('MCMC_NKUS5725_20_Jan_2026_19_33_58');
priornames = fieldnames(me.estimation.priors);
ndraw = 100000;
res = mcmc(results,priornames,{1:5:ndraw,1:2});
[summary_tables, MyQuantiles] = summary(res);
figure('Name','Diagnostics')
subplot(2,4,1);autocorrplot(res,'theta');subplot(2,4,2);densplot(res,'theta');
subplot(2,4,3);meanplot(res,'theta');subplot(2,4,4);traceplot(res,'theta');
subplot(2,4,5);autocorrplot(res,'sig_mu_vol_1');subplot(2,4,6);densplot(res,'sig_mu_vol_1');
subplot(2,4,7);meanplot(res,'sig_mu_vol_1');subplot(2,4,8);traceplot(res,'sig_mu_vol_1');
```

Line 1 loads the saved MCMC results, which contain the raw parameter draws. Lines 2–4 select the parameters to analyze (here, all prior-defined parameters, but only every fifth draw from two chains) and create the mcmc object. Line 5 produces numerical summaries—posterior means, quantiles, and credible intervals—reported in summary_tables. Lines 6–10 generate standard graphical diagnostics, illustrated in Figure 3.[11] The *autocorrelation plots* (first column) show how persistent successive draws are. For well-mixed chains, autocorrelations decay quickly; slow decay signals poor exploration and the need for longer runs. *Density plots* (second column) estimate the marginal posterior for each parameter, which should be smooth and stable across chains. *Mean plots* (third column) trace the running average of each parameter; stability over time indicates convergence to the stationary distribution. Finally, *trace plots* (fourth column) display raw draws across iterations; well-behaved chains "wander" freely across the posterior support rather than sticking or drifting.

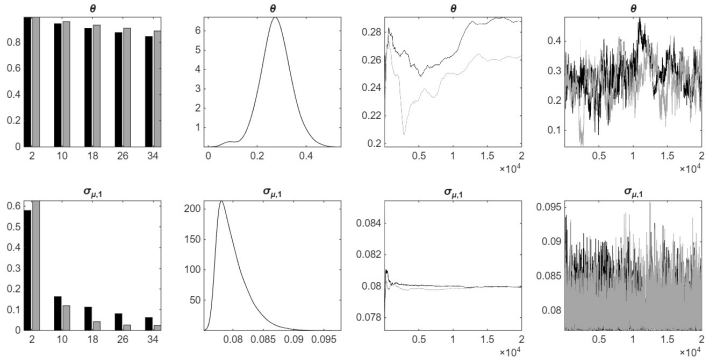

Figure 3: MCMC diagnostics for two selected parameters: θ (top row) and $\sigma_{\mu,1}$ (bottom row). The four columns show autocorrelation, marginal densities, running means, and trace plots.

Figure 3 contrasts two parameters with different mixing properties. The persistence parameter θ exhibits slow convergence and noticeable variation across chains, while the shock volatility $\sigma_{\mu,1}$ converges quickly with tight posterior support. This contrast illustrates why visual diagnostics are indispensable: they reveal whether 100,000 draws are sufficient or whether longer runs and a larger burn-in are required. In applied work, it is common to generate around 500,000 draws and discard the first half to reduce sensitivity to the initialization of the MH algorithm.

Box 11: Posterior analysis with `analyze_mcmc.m`. Part I: computations

```
clear all; load('MCMC_NKUS5725_20_Jan_2026_19_33_58');
% --- Settings ---
Nd        = 100;             % number of posterior draws to use
chain_id  = 1;               % which chain in 'results' to use
horizon   = 20;              % IRF horizon (used below for deterministic IRFs)
pct       = [5 50 95];       % credible bands
vnames_db  = fieldnames(db); %
T         = db.(vnames_db{1}).NumberOfObservations; % Sample length
t         = 1:T;                                    % Time axis
% --- Storage for probabilities and smoothed gap across draws ---
prob_dove  = NaN(Nd,T);     % P(policy state = 2)
prob_volhi = NaN(Nd,T);     % P(volatility state = 2)
gap_smooth = NaN(Nd,T);     % smoothed output gap (omega)
% --- Main loop over posterior draws ---
for d = 1:Nd
    for k = 1:20   % max re-draws (pick a small-ish cap)
       [draw, me_draw] = draw_parameter(me, results{chain_id}.pop);
       [filt, ~] = filter(me_draw,'data',db);
       if ~isempty(filt), break; end
    end
    if isempty(filt), error('filter returned empty after max re-draws.'); end
    % Collect smoothed probabilities and smoothed output gap
    prob_dove(d,:)  = filt.smoothed_state_probabilities.policy_2.data;
    prob_volhi(d,:) = filt.smoothed_state_probabilities.vol_2.data;
    gap_smooth(d,:) = filt.Expected_smoothed_variables.Omega.data;
end
```

Working with posterior draws. Once MCMC has produced posterior draws, we can propagate parameter uncertainty to the model's main outputs. The analysis proceeds in two stages, shown in Box 11 (computations) and Box 12 (output and visualization).

The driver reuses the saved MCMC object `results` together with the estimated model `me`. For each selected posterior draw, we call `draw_parameter` to generate a parameterized model object (`me_draw`). We then refilter the data with the `filter` procedure to extract smoothed regime probabilities and the smoothed output gap ω_t. From these, we later compute posterior means and 90% credible bands (5th, 50th, and 95th percentiles) for quantities such as state probabilities and latent variables, see Figure 4.

Box 12: Posterior analysis with `analyze_mcmc.m`. Part II: output and visualization

```
% --- Posterior summaries (means and 90% credible bands) ---
prob_dove_q  = prctile(prob_dove,  pct, 1);   % 3-by-T (5th, 50th, 95th)
prob_volhi_q = prctile(prob_volhi, pct, 1);
gap_q        = prctile(gap_smooth,pct, 1);
% --- Plot probabilities and gap with 90% bands (simple skeleton) ---
start_dt = datetime(1954,7,1); % 1954Q3 starts July 1954
timevec = start_dt + calquarters(0:T-1);
figure('Name','Posterior probabilities and output gap','Color','w');
subplot(3,1,1);fill([t, fliplr(t)],[gap_q(1,:),fliplr(gap_q(3,:))],...
   0.9*[1 1 1],'EdgeColor','none'); hold on;
plot(t, gap_q(2,:), 'k', 'LineWidth', 1);
xticks(1:20:T); xticklabels(datestr(timevec(1:20:T),'yyyyQQ')); xlim([1 T]);
title('A: Output Gap','FontSize',14);
subplot(3,1,2);fill([t, fliplr(t)],[prob_dove_q(1,:),fliplr(prob_dove_q(3,:))],...
   0.9*[1 1 1],'EdgeColor','none');hold on;
plot(t, prob_dove_q(2,:), 'k', 'LineWidth', 1);
xticks(1:20:T);xticklabels(datestr(timevec(1:20:T),'yyyyQQ'));xlim([1 T]);
title('B: Probability to be in Dovish policy state','FontSize',14);
subplot(3,1,3);fill([t, fliplr(t)],[prob_volhi_q(1,:),fliplr(prob_volhi_q(3,:))],...
   0.9*[1 1 1],'EdgeColor','none');hold on;
plot(t, prob_volhi_q(2,:), 'k', 'LineWidth', 1);
xticks(1:20:T);xticklabels(datestr(timevec(1:20:T),'yyyyQQ'));xlim([1 T]);
title('C: Probability to be in High volatility state','FontSize',14);
```

In practice, the same loop can be extended to other quantities of interest. For example, one can compute variance decompositions by inserting `[vardec,~] = variance_decomposition(me_draw);` inside the loop (lines 14–23 in Box 11) and then aggregating the results across draws to compute posterior means and quantiles. Similarly, historical decompositions can be generated by `hd = historical_decomposition_switch(me);` and visualized with `plot_decomp('1957Q3:2025Q3', hd.(varnameind))` for a chosen variable and sample range. These should be called after the loop.

Box 11 performs the core computations: loading posterior draws, parameterizing the model at each draw, filtering the data, and collecting the resulting smoothed series. The subsequent step, shown in Box 12, summarizes these results by forming credible intervals and plotting them for the output gap and regime probabilities. The resulting figures illustrate how estimated states evolve over time and how parameter uncertainty translates into confidence bands.

6 Conditional Simulations or "Counterfactuals"

Once a regime–switching DSGE model has been parameterized, solved, and typically estimated, it can be used not only for forecasting or unconditional simulation, but also for *conditional simulations*—often

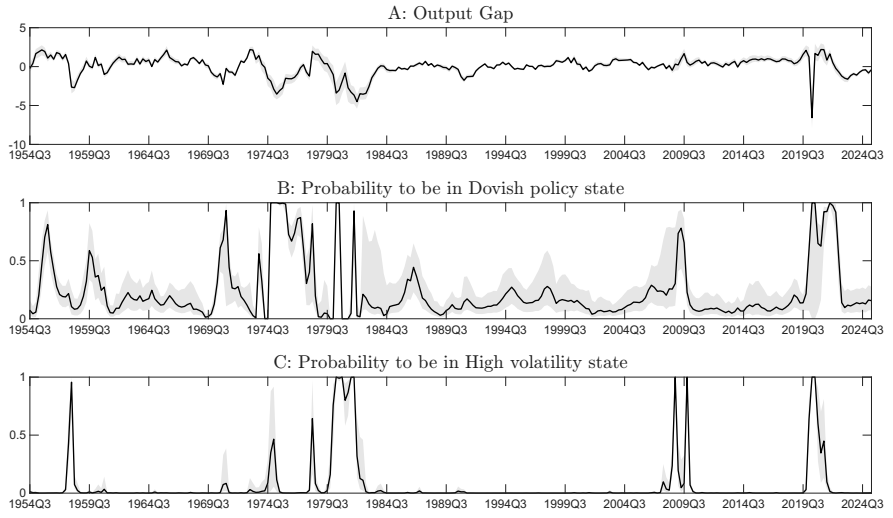

Figure 4: Estimated variables and state probabilities.

called *counterfactuals*. A counterfactual exercise addresses the question: What would the economy have looked like if a particular shock, policy state, or behavioral response had differed, holding all other factors consistent with the model's structure?

6.1 Concept and Motivation

Formally, a counterfactual simulation starts from an initial state (or a whole sequence of inferred shocks and regimes) and alters one element of the system—such as a policy regime, a structural shock, or a behavioral parameter—while preserving all other model-consistent dynamics. The goal is to generate an internally consistent alternative history that answers questions like: "What if monetary policy had remained in state 2?" or "What if technology shocks had been absent?"

These exercises are most meaningful once the model has been estimated, since the estimated parameters and smoothed shocks summarize how the model interprets historical data. Conditional simulations can then be used to quantify the role of different drivers—policy, luck, or structural persistence—behind the observed dynamics.

Even so, counterfactuals are not confined to estimated models. Any parameterized and solved DSGE model can be simulated conditionally, provided that the initial state and relevant sequences (states or shocks) are explicitly specified.

6.2 Historical Replication as a Baseline

Before altering history, it is necessary to verify that the estimated model can replicate it. In constant-parameter models, feeding the smoothed shocks from the Kalman filter back into the model reproduces the data exactly. In regime-switching models, perfect replication is not possible because the realized regime is unobserved. Instead, we approximate history by conditioning on the *most likely regime* at each date, as inferred from the smoothed regime probabilities. If the model fits the data well, this historical replication should closely match the observed series within the credible bands implied by parameter uncertainty. Box 13 shows how to code a replication of history.

Explanation of the code. Box 13 shows that the key command is `simulate`, which produces model-consistent trajectories once a simulation plan (`simplan`) specifies initial conditions and sequences for shocks and regimes. The solution of the model is already available from the previous `solve` step; `simplan` simply tells RISE which shocks and regimes to apply at each date during the simulation.

Box 13: History replication with `replicate_history.m`

```
clear all; load('MCMC_NKUS5725_20_Jan_2026_19_33_58');
% Dimensions and variable names
T = me.options.data.NumberOfObservations; chain_id = 1; K = 100;rng(123);
exo_names = get(me,'exo_list'); endo_names = get(me,'endo_list');
for indx = 1:K
   for k = 1:20   % max re-draws (pick a small-ish cap)
      [draw, me_draw] = draw_parameter(me, results{chain_id}.pop);
      [filt, ~] = filter(me_draw,'data',db);
      if ~isempty(filt), break; end
   end
   if isempty(filt), error('filter returned empty after max re-draws.'); end
   % Build regime path from smoothed probabilities
    [~, pol_path] = max([filt.smoothed_state_probabilities.policy_1.data,...
                         filt.smoothed_state_probabilities.policy_2.data],[],2);
    [~, vol_path] = max([filt.smoothed_state_probabilities.vol_1.data,...
                         filt.smoothed_state_probabilities.vol_2.data],[],2);
    regime_path = 2*(pol_path-1) + vol_path;  % (1,1)->1 ... (2,2)->4
    % Construct plan with initial conditions
    end_hist = 1; end_forecast = T; init_reg = regime_path(end_hist);
    plan_hist = simplan(me_draw,[end_hist,end_forecast],init_reg);
    % Initialise all endogenous vars (incl. forward-looking ones)
    for k = 1:numel(endo_names)
        v = endo_names{k}; x0 = filt.Expected_smoothed_variables.(v).data(1);
        plan_hist = append(plan_hist,{v,end_hist,x0});
    end
    % And exogenous shocks
    for k = 1:numel(exo_names)
        v = exo_names{k}; x0 = filt.Expected_smoothed_shocks.(v).data(1);
        plan_hist = append(plan_hist,{v,end_hist,x0});
    end
    % Pin regime and smoothed shocks for t = 1..T
    plan_hist = append(plan_hist,{'regime',end_hist+1:end_forecast,...
                                  regime_path(end_hist+1:end_forecast)});
    for j = 1:numel(exo_names)
        sj = filt.Expected_smoothed_shocks.(exo_names{j}).data(:);
        plan_hist = append(plan_hist,{exo_names{j},end_hist+1:end_forecast,...
                                      sj(end_hist+1:end_forecast)});
    end
    % Historical simulation
    sim_hist = simulate(me_draw,'simul_historical_data',plan_hist);
    R_obs_sim(indx,:)  = sim_hist.R_obs.data; Pai_obs_sim(indx,:) = sim_hist.Pai_obs.data;
    Dy_obs_sim(indx,:) = sim_hist.Dy_obs.data;
end
% Posterior summaries (means and 90% credible bands)
pct = [5 50 95]; R_obs_sim_q  = prctile(R_obs_sim,pct,1);
Pai_obs_sim_q = prctile(Pai_obs_sim,pct,1); Dy_obs_sim_q = prctile(Dy_obs_sim,pct,1);
```

Lines 1–8 load the stored MCMC database and, tto reflect parameter uncertainty, randomly draw $K = 100$ parameter vectors from the posterior sample (`results{chain_id}.pop`). For each draw, the

model is re-parameterized (`me_draw`) and re-filtered using the observed data `db`. This yields the best available information on latent objects—smoothed states, shocks, and state probabilities—conditional on that parameter draw. Basing the simulation on these smoothed quantities ensures that we start from a realistic estimate of the economy's position and shock history, consistent with both the model and the data.

Lines 12–17 reconstruct the most likely regime sequence from the smoothed regime probabilities of each Markov chain (here, `policy` and `vol`). Since the true sequence of regimes is unobserved, we approximate it by taking, at each date, the most probable state for each chain (`argmax`) and combining them into a single regime index. In models with more chains, the mapping from chain states to regime numbers follows the Kronecker product of chain indices.

Lines 18–33 then build a *historical plan*: a time structure that pins down all information needed for the conditional simulation. The plan is initialized with the smoothed starting values of all endogenous and exogenous variables so that the simulation begins exactly from the state inferred by the smoother at the start of the sample. Next, lines 34–40 feed back the entire sequence of smoothed structural shocks and fix the regime path. Once the plan is complete, the `simulate` command advances the system using the model's decision rule, generating paths for all variables—both latent and observable.

Lines 41–42 store the simulated paths for observable series, since in most applications, researchers are primarily interested in alternative histories for observables (e.g., inflation, output, and the policy rate). However, the same code can easily be extended to save and analyze latent variables as well. Finally, repeating this process across many posterior draws provides a distribution of simulated trajectories, summarized in lines 44–46. This posterior distribution shows how closely the estimated model can replicate the observed history while accounting for parameter uncertainty.

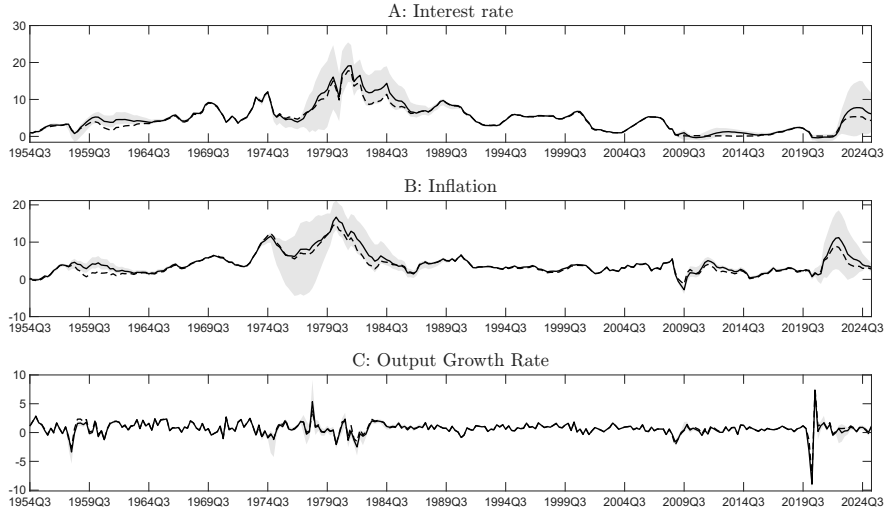

Figure 5: Historical and simulated data.

Interpretation of the output. Figure 5 shows that the replicated histories (posterior median and 90% credible bands) closely track the demeaned data for inflation, the interest rate, and output growth.[12] Discrepancies arise mainly in periods when regime probabilities are uncertain, highlighting that perfect reproduction is infeasible when regimes are latent.

6.3 Designing Counterfactual Scenarios

Once the historical baseline is reproducible, we can conduct genuine counterfactual exercises by altering one component of the simulation plan while keeping everything else fixed. In a Markov–switching DSGE model, this often means changing the realized regime path for a subset of periods, while preserving the same estimated parameters, shocks, and initial states. Because the model structure enforces consistency across variables, such an experiment provides a disciplined answer to the question: *What would have happened if policy, luck, or behavior had been different?*

In practice, counterfactuals are implemented by cloning the historical plan and overriding only the relevant entries. In our application, the model attributes the post-pandemic inflation surge of 2021–2022 to a combination of strong cost-push shocks and a dovish policy stance. In the data, inflation exceeded 4% per annum in 2021Q2 and peaked near 9% in 2022Q2, while the Federal Reserve began tightening only in March 2022. The estimated model suggests that in 2022Q2, the most likely regime was the dovish one. We can therefore ask: *what if policy had turned hawkish two quarters earlier, from 2021Q4 onward?*

Immediately after defining the baseline plan `plan_hist` is specified (after line 42 in Box 13 and before the end of the loop), we create a modified version of the plan that forces the policy chain to switch earlier and simulate it as well:

```
plan_cf = plan_hist;              % start from the historical plan
pol_path_cf  = pol_path;          % copy historical policy state
pol_path_cf(270:end) = 1;         % force hawkish state from 2021Q4
regime_path_cf = 2*(pol_path_cf - 1) + vol_path; % rebuild regime index
plan_hist_cf = append(plan_cf, {'regime', 270:end_forecast, ...
                                regime_path_cf(270:end_forecast)});
sim_hist_cf = simulate(me_draw,'simul_historical_data',plan_hist_cf);
```

Because `RISE` processes plan entries sequentially, the newly appended lines override the old ones, so the only change is the policy regime assignment for the specified period. The modified plan is then used in a second call to `simulate` and the simulated observables are stored alongside the baseline results. Since the parameters, shocks, and initial conditions are identical, the difference between the two simulations isolates the effect of policy timing alone.

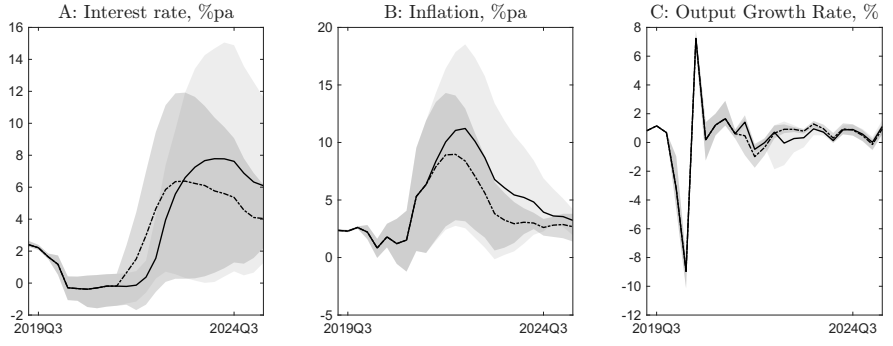

Figure 6: Cost of Living Crisis.

We also need to compute posterior summaries of counterfactual simulations outside the loop, similar to how it was done in lines 45-46 in Box 13. We then can visualize the difference between baseline and counterfactual simulations. Figure 6 reports the results. In each subplot, the solid line shows the mean of the baseline simulation and the dash–dotted line shows the mean of the counterfactual; shaded

bands represent 90% posterior credible intervals across 100 draws. The experiment suggests that an earlier tightening would have raised the policy rate sooner (Panel A) and lowered the inflation peak by around one percentage point (Panel B), at the cost of a modest short-run decline in output (Panel C). Subsequent inflation would have remained below the observed path, consistent with quicker re-anchoring of expectations. These results illustrate how regime-switching DSGE models can be used to evaluate the timing and effects of alternative policy choices in a fully structural, model-consistent framework.

7 Practical Issues and Troubleshooting in RISE

Up to now, we have focused on the core workflow—solving, simulating, filtering, and estimating regime-switching DSGE models. To keep the exposition clean, we deferred several practical issues that arise in applied work. This section brings them together for convenience.

7.1 Solving Models with Nontrivial Steady-states

For linearized models, such as the baseline New Keynesian example introduced earlier, steady states pose no difficulty: all variables are expressed as deviations from their steady-state values, and their steady-state levels are zero. For nonlinear models, however—even for the nonlinear version of the same NK model—the main challenge often lies in finding the steady state, which forms part of the model's solution step. RISE provides several tools to facilitate this task. We illustrate them using the nonlinear model whose log-linearization yields equations (1.1)–(1.8).[13]

Specifying a nonlinear model. We begin by presenting the full nonlinear version of the baseline New Keynesian model. This is the structural system that, when log-linearized around its steady state, collapses exactly to the equations of the linear model:

$$\text{Consumption Euler equation} \qquad 1 = \frac{\beta}{\nu} R_t \mathbb{E}_t \left[\left(\frac{C_{t+1} e^{\xi_{t+1}}}{C_t e^{\xi_t}} \right)^{-\sigma} \frac{\Pi_{t+1}^{-1}}{e^{z_{t+1}}} \right], \tag{1.18}$$

$$\text{Habit dynamics} \qquad C_t = y_t - \theta y_{t-1}, \tag{1.19}$$

$$\text{Optimal price} \qquad p_t^f = \frac{\eta}{\eta - 1} \frac{G_{1,t}}{G_{2,t}}, \tag{1.20}$$

$$\text{Helper equation} \qquad G_{1,t} = C_t^{-\sigma} e^{\xi_t} e^{\mu_t} w_t y_t + \gamma \beta \mathbb{E}_t \left[\Pi_{t+1}^{\eta} G_{1,t+1} \right], \tag{1.21}$$

$$\text{Helper equation} \qquad G_{2,t} = C_t^{-\sigma} e^{\xi_t} y_t + \gamma \beta \mathbb{E}_t \left[\Pi_{t+1}^{\eta-1} G_{2,t+1} \right], \tag{1.22}$$

$$\text{Rule-of-thumb price} \qquad p_t^b = p_{t-1}^* \Pi_{t-1}, \tag{1.23}$$

$$\text{Aggregate reset price} \qquad p_t^* = (p_t^f)^{1-\zeta} (p_t^b)^{\zeta}, \tag{1.24}$$

$$\text{Gross inflation} \qquad 1 = \gamma \Pi_t^{\eta-1} + (1-\gamma)(p_t^*)^{1-\eta}. \tag{1.25}$$

$$\text{Labor supply} \qquad w_t = N_t^{\varphi} C_t^{\sigma}, \tag{1.26}$$

$$\text{Production function} \qquad N_t = y_t \Delta_t, \tag{1.27}$$

$$\text{Price dispersion} \qquad \Delta_t = (1-\gamma)\,[p_t^*]^{-\eta} + \gamma \Pi_t^{\eta} \Delta_{t-1}, \tag{1.28}$$

$$\text{Policy rule} \qquad \frac{R_t}{R} = \left(\frac{R_{t-1}}{R}\right)^{\rho} \left[\left(\frac{\Pi_t}{\Pi^T}\right)^{\phi_1} \left(\frac{y_t z_t}{y_{t-1}}\right)^{\phi_2}\right]^{1-\rho} e^{\sigma_r \varepsilon_t^r}. \tag{1.29}$$

If log-linearized around the steady state, this system collapses exactly to equations (1.1)–(1.5). To close the model, equations (1.6) and (1.8) for technology and preference shocks can be reused. However, the stochastic process for the cost–push shock must be rescaled to match the linearized model, where the cost–push term in the Phillips curve (1.3) enters with a unit coefficient. Specifically, μ_t in equation (1.21) should follow $\mu_t = \rho_\mu \mu_{t-1} + \lambda \sigma_\mu \varepsilon_t^\mu$, where $\lambda = \gamma(\beta\zeta + 1)/((1-\gamma\beta)(1-\zeta)(1-\gamma))$.

Box 14 shows the associated model file with one Markov chain (`policy`). There is no difference between specifying linear and nonlinear models in a `.rs` file, but note the use of the # symbol to append steady-state relations (lines 11–12), not only to declare derived parameters in line 9. Such relations can be appended to any equation and often help the solver.

Box 14: Model file (`nk_ms_nonlinear.rs`)

```
 1 @endogenous(log) Y , Pai, R, C, G1, G2, Pstar, Pf, Pb, D, W, N
 2 @endogenous Z, Mu, Xi
 3 @exogenous Ez, Emu, Exi, Ei
 4 @parameters beta, nu, sigma, varphi, theta, gamma, zeta, eta, PaiT, rho_z, rho_mu, rho_xi,
 5   sig_z, sig_mu, sig_xi, rho_r, sig_r, policy_tp_1_2, policy_tp_2_1
 6 @parameters(policy,2) phi1, phi2
 7
 8 @model
 9   # lambda = ((gamma*(beta*zeta+1))/((1-gamma*beta)*(1-zeta)*(1-gamma)));
10   1 = beta/nu*(R{t})*(C{t+1}*exp(Xi{t+1})/C{t}/exp(Xi{t}))^(-sigma)*1/exp(Z{t+1})/Pai{t+1}
11   # 1 = beta/nu*(R{stst})/PaiT;
12   C{t} = Y{t} - theta*Y{t-1};
13   Pf{t} = eta/(eta-1)*G1{t}/G2{t};
14   G1{t} = (C{t}*exp(Xi{t}))^(-sigma)*Y{t}*exp(Mu{t})*W{t} + gamma*beta*Pai{t+1}^eta*G1{t+1};
15   G2{t} = (C{t}*exp(Xi{t}))^(-sigma)*Y{t} + gamma*beta*Pai{t+1}^(eta-1)*G2{t+1};
16   Pb{t} = Pstar{t-1}*Pai{t-1};
17   Pstar{t} = Pf{t}^(1-zeta)*Pb{t}^zeta;
18   1 = gamma*Pai{t}^(eta-1) + (1-gamma)*(Pstar{t})^(1-eta);
09   W{t} = N{t}^varphi*C{t}^sigma;
20   N{t} = D{t}*Y{t};
21   D{t} = (1-gamma)*Pstar{t}^(-eta) + Pai{t}^eta*gamma*D{t-1};
22   R{t}/R{stst} = (R{t-1}/R{stst})^rho_r
23     *[(Pai{t}/PaiT)^phi1*(Y{t}/(Y{t-1})*exp(Z{t}))^phi2]^(1-rho_r)*exp(sig_r*Ei{t});
24   Z{t}  = rho_z*Z{t-1}  + sig_z*Ez{t};
25   Mu{t} = rho_mu*Mu{t-1} + lambda*sig_mu*Emu{t};
26   Xi{t} = rho_xi*Xi{t-1} + sig_xi*Exi{t};
```

Two parameters appear only in the nonlinear specification: ν (trend output growth) and η (the elasticity of substitution between differentiated goods). Together with the inflation target Π^T in equation (1.29), they must be included in the parameter set.

The inflation target Π^T—a policy-choice parameter—pins down the steady state of gross inflation and thereby the steady-state gross nominal interest rate R through equation (1.29).

While we must use the new parameter `PaiT` in the policy rule (lines 22-23 in Box 14), we may denote the unknown steady-state policy rate by `R{stst}`; `RISE` solves for it as part of the steady-state system.

Endogenous variables are divided into those that are strictly positive at all times (line 1) and the

remainder (line 2). The (log) tag applied to the first group instructs RISE to work internally with their logarithms, often simplifying multiplicative relationships and easing steady-state computation.

The default solver may fail. Box 15 shows a complete driver file for the nonlinear model. Lines 1–11 parameterize the model (lines 3–9), verify that all parameters are assigned (line 10), and then call the default solve routine (line 11). Shock standard deviations are divided by 100 so that they are measured in decimal units rather than percentages.

If lines 1–11 are executed in isolation, the default solver fails to find a steady state and returns m0.nsols = 0, despite the presence of steady-state shortcuts in the model file nk_nonlinear.rs. This outcome is common in nonlinear DSGE models: even well-specified systems often require additional guidance for steady-state computation.

Box 15: Driver (driver_nk_nonlinear.m)

```
 1 clear all; close all; clc;
 2 m = rise('nk_ms_nonlinear.rs');
 3 p = struct('beta',1/(1+0.706/400),'sigma',2.9,'varphi',2.5,'theta',0.82,...
 4   'gamma',0.77,'zeta',0.10,'eta',10,'nu',1,'rho_z',0.90,'rho_mu',0.70,...
 5   'rho_xi',0.80,'rho_r',0.79,'PaiT',1.,'sig_r',0.10/100,...
 6   'sig_z',0.50/100,'sig_mu',0.15/100,'sig_xi',0.10/100,...
 7   'policy_tp_1_2',0.05,'policy_tp_2_1',0.05, ... %% Tr. prob. (off-diagonal)
 8   'phi1_policy_1',1.72,'phi1_policy_2',1.20,... % state-specific coefficients
 9   'phi2_policy_1',0.49,'phi2_policy_2',0.20);  % state-specific coefficients
10 m = set(m,'parameters',p);isnan(m);
11 m0 = solve(m); print_solution(m0); % brute force
12 %% --- approaches to solution ---
13 %      provide a steady state fiile
14 ms1 = solve(m,'sstate_file',@nk_nonlinear_ssfile); print_solution(ms1);
15 resid(set(m,'sstate_imposed',true,'sstate_file',@nk_nonlinear_ssfile));
16 %      brute force + block decomposition
17 ms2 = solve(m,'sstate_blocks',true); print_solution(ms2);
18 ms21 = solve(m,'sstate_blocks',true,'debug',true,'sstate_solver',...
19    {'lsqnonlin','MaxIter',2000,'MaxFunEvals',1000000}); print_solution(ms21);
20 ms22 = solve(m,'sstate_blocks',true,'debug',true,'sstate_solver',...
21    {'fsolve','MaxIter',2000,'MaxFunEvals',1000000}); print_solution(ms22);
22 %      smart initial conditions and bounds
23 bnds = struct();
24 bnds.Y = [2.4630 0 Inf]; bnds.Pai = [1.0000 0 Inf]; bnds.R = [0.0018 0 Inf];
25 bnds.C = [0.4433 0 Inf]; bnds.G1 = [101.3613 0 Inf]; bnds.G2 = [112.6237 0 Inf];
26 bnds.Pstar = [1.0000 0.5 2]; bnds.Pf = [1.0000 0.5 2]; bnds.Pb = [1.0000 0.5 2];
27 bnds.D = [1.0000 0.5 2]; bnds.W = [0.90000 0.5 2];
28 ms3 = solve(m,'sstate_bounds',bnds,'debug',true,'sstate_blocks',true); print_solution(ms3);
```

The remainder of the driver (lines 12–28) illustrates alternative solution strategies. Two approaches are particularly effective in practice. The first supplies the solver with an explicit steady-state file. The second exploits block decomposition and numerical solvers, optionally combined with bounds. We discuss each approach in turn, referring to the corresponding lines in Box 15.

Approach 1: User-provided steady-state file. When the solver cannot find a steady state on its own, it is often helpful to provide a dedicated steady-state routine. Such a file may return an exact solution (as in this example), an approximate one, or a subset of steady-state values, leaving the remaining variables to be determined by RISE. Box 16 shows a simple implementation.

Box 16: Steady-state file (nk_nonlinear_ssfile.m)

```
 1 function [y_,newp,retcode] = nk_nonlinear_ssfile(obj, y_, p, d, id)
 2 retcode = 0;
 3 if nargin == 1
 4   % list of endogenous variables to be calculated
 5   y_ = {'Y', 'Pai', 'R', 'C', 'G1', 'G2', 'Pstar', 'Pf', 'Pb', 'D', 'W', 'N'};
 6   % list of parameters to be computed during steady state calculation
 7   newp = {};
 8 else % provide steady state for everything except LMs
 9   % no parameters to update or create in the steady state file
10   newp = [];
11   Pai = p.PaiT;
12   Pstar = ((1-p.gamma*p.PaiT^(p.eta-1))/(1-p.gamma))^(1/(1-p.eta));
13   Pb = Pstar*p.PaiT;
14   Pf = (Pstar*Pb^(-p.zeta))^(1/(1-p.zeta));
15   W = (1-p.gamma*p.beta*p.PaiT^p.eta)/(1-p.gamma*p.beta*p.PaiT^(p.eta-1))*(p.eta-1)/p.eta*Pf;
16   D = (1-p.gamma)*Pstar^(-p.eta)/(1-p.PaiT^p.eta*p.gamma);
17   Y = (W*(1-p.theta)^(-p.sigma)*D^(-p.varphi))^(1/(p.varphi+p.sigma));
18   C = (1-p.theta)*Y;
19   G1 = ((C^(-p.sigma)*W*Y)/((1-p.gamma*p.beta*p.PaiT^p.eta)));
20   G2 = ((C^(-p.sigma)*Y)/((1-p.gamma*p.beta*p.PaiT^(p.eta-1))));
21   R = p.nu*p.PaiT/p.beta;
22   N = D*Y;
23
24   ys = [Y, Pai, R, C, G1, G2, Pstar, Pf, Pb, D, W, N].'; % DO NOT INCLUDE IMPOSED VARIABLE HERE
25
26   % check the validity of the calculations
27   if ~utils.error.valid(ys) retcode = 1; else y_(id) = ys; end
28 end
29 end
```

A RISE steady-state file must follow a strict input–output structure. When called with a single input, it declares the list of variables to be computed (line 5) and the list of parameters to be updated (line 7). When called with all arguments, it computes the steady state (lines 11–22) and returns the corresponding values in the same order. The check in lines 26–27 validates the result and signals success or failure via the return code `retcode`.

Users can replace lines 5 and 10–22 with their own MATLAB code (ensuring that variable names do not conflict with internal parameters of the function) if no closed form exists. In models with composite parameters, the output `newp` can also update parameter values.

While the researcher may know that the inflation target `PaiT` defines the steady-state of gross inflation, RISE treats inflation like any other endogenous variable and, if requested, computes its steady-state within this file.

The steady-state file need not provide values for *all* variables. Partial solutions are accepted; in such cases, RISE initializes the remaining steady-state values using default guesses (typically zeros). It is therefore helpful to ensure that such variables are either declared with the `(log)` tag in the model file (Box 14, line 1) or genuinely admit zero steady states.

Once created, the steady-state file can be passed to the solver as an option (line 14 in Box 15). In this example, providing the steady-state file ensures that the model is solved successfully, and the solution is stored in the object `ms1`.

If the supplied steady-state file does not yield an exact solution, the function `resid` can be used to diagnose the problem. Executing line 15 of the driver prints the residuals of each equation in all regimes, evaluated at the imposed steady state.

Approach 2: Block decomposition solver. A lighter—and often effective—alternative is to let RISE exploit the algebraic structure of the model. The option `'sstate_blocks'` (line 17) instructs the solver to partition the steady-state system into blocks and attempt a recursive solution. This mirrors the manual computation in lines 11–22 of Box 16, but is handled automatically.

In our example, the command in line 17 fails, with diagnostics indicating that the solver has exhausted the maximum number of iterations. To assess convergence, one can enable diagnostic output using the option `'debug',true`.

Each of the calls in lines 18–19 and 20–21 produces a valid solution in our example. In both cases, we increase the maximum number of iterations and function evaluations. The first call applies these settings to the default solver, `lsqnonlin`, while the second uses the alternative solver `fsolve`.

Providing simple steady-state *bounds* can further stabilize and accelerate the block solver. Bounds also supply initial guesses. The syntax for defining bounds is shown in lines 23–27, and the resulting structure is passed together with the block decomposition option in line 28 of Box 15.

Combining `'sstate_bounds'` with `'sstate_blocks'` often enables convergence in nonlinear models that otherwise fail to solve, without requiring a custom steady-state file. With `'debug',true` enabled, one can also verify that the number of required iterations decreases substantially.

The success of this approach hinges on the use of the `(log)` tag in line 1 of Box 14. This transformation is essential: without it, neither the block decomposition solver nor the default steady-state routine produces a valid solution.

Practical advice. In practice, one typically tries the block decomposition option first; if that fails, a user-provided steady-state file is required. The two approaches are complementary: the block solver offers convenience, while a dedicated steady-state file provides greater control and robustness, especially for complex nonlinear models.

More advanced techniques—such as improving initial guesses, simplifying equations, or solving selected parameters within the steady-state routine—can further assist in challenging cases, but lie beyond the scope of this chapter.

7.2 Solving the Dynamics

Finding a first-order solution. By default, RISE computes a first–order (linear) solution to the DSGE model: once a deterministic steady-state has been obtained, it constructs the linear mapping that describes dynamics in a neighborhood of that steady-state. In practice, obtaining the dynamic solution is usually less problematic than finding the steady state. However, it is useful to know a few options when the default solver fails to converge.

The preceding subsection described how to control the steady-state step within `solve(m,...)`. In addition, some options affect the dynamic solution step. Besides `'debug',true`—which produces detailed diagnostics (iteration traces, residual magnitudes, and which equations remain unsatisfied)—you can request finer progress reports from the fixed-point–style solver with `'fix_point_verbose',true`. If the dynamic step stops too early, increase the iteration cap via `'fix_point_maxiter',5000` (or another suitable value), and adjust the residual tolerance via `'fix_point_TolFun',1e-6`. Although these flags resemble those used for steady-state computation, they govern the subsequent step that finds the linear solution around the (already computed) steady-state. They help the user diagnose whether non-convergence is due to too few iterations, tight tolerances, or weak starting values in the dynamic step.

The dynamic step can also switch among alternative internal solvers using `'solver'`—typical choices include `'mn'`, `'mnk'`, and `'sims'`. These differ primarily in numerical implementation and stability checks. In most applications, the default works well, but if it fails to converge, trying an alternative is

worthwhile.

In all cases, combining `'debug',true,` and `'fix_point_verbose',true,` with generous iteration limits often reveals which equations or variable blocks drive instability, guiding better initial conditions or modest analytical simplifications.

Higher degree of perturbation. The option `'solve_order',k,...` specifies the order of the Taylor (perturbation) expansion used to compute the model's decision rules around the stochastic steady-state. Setting `k=1` produces the standard first–order (linear) approximation; higher values request nonlinear perturbations that capture risk and volatility effects when the model structure supports them.

However, this option alone is not sufficient to compute higher–order solutions: one must also specify how the derivatives of policy functions around the steady-state are obtained.

When the model is parsed with `m = rise('nk_nonlinear.rs')`, RISE computes first–order derivatives analytically (symbolically) by default, but it does not automatically compute higher–order derivatives. To instruct RISE to do so analytically, use the following pair of commands:

```
m = rise('nk_nonlinear.rs','max_deriv_order',2); ... m = solve(m,'solve_order',2);
```

Alternatively, automatic (algorithmic) differentiation can be used instead:

```
m = rise('nk_nonlinear.rs'); ...
m = solve(m,'solve_order',2,'solve_derivatives_type','automatic');
```

In other words, when requesting a higher–order solution, the derivatives–type option must be explicitly specified.

Even for higher–order perturbations, RISE first computes the steady-state and the first–order structure within the same call. Therefore, it is good practice to verify and debug the first–order solution before moving to higher orders.

7.3 Accessing Information in RISE

Getting model information. This subsection summarizes key commands helpful in debugging, model documentation, and reusing RISE output in custom code.

Box 17: Quick queries

```
     % current parameter values (as fields)
p = get(m,'parameters');  % e.g., p.sigma, p.theta, ...
defs  = get(m,'definitions');
pmode = get(me,'mode');   % e.g., pmode.sigma, pmode.theta, ...
     % name lists (cell arrays of char)
pnames = get(m,'par_list'); % 'sigma','theta',...
defnames = get(m,'def_list'); % 'varkappa',...
endnames = get(m,'endo_list'); % 'Pai','Y','R',...
exnames = get(m,'exo_list'); % 'Ez','Emu',...
obsnames = get(m,'obs_list'); % 'Dy_obs', 'R_obs',...
chainnames = get(m,'chain_list'); % 'const', 'policy',...
     % Equations and related metadata
eqs   = get(m,'equations');    % cellstr of model equations
     % steady-state (after solve)
stst   =   get(m,'sstate');       % for @endogenous{log) these are logs
ststlev  = get(m,'sstatelevel');  % in levels for all variables
     % Priors and posterior mode (after estimate)
pmode  = get(me,'mode');    % structure of parameter values at the posterior mode
```

RISE allows users to *query* a model object for components frequently needed in user scripts—such as parameter values, equation strings, lists of variables, priors and posterior modes, or compact solution objects. The main interface is the method `get`, supplemented by several task–specific utilities. Box 17 lists a collection of commonly used queries.

Lines 2–11 show how to retrieve the current parameterization as a MATLAB `struct` together with the associated name lists. For documentation or debugging, it is often helpful to extract the parsed equations (line 13 in Box 17). Equations are available immediately after parsing the model with `rise`, while steady-state values become available only after the model has been solved. Lines 15-16 show how to obtain them, both in logs and in levels.

When the model has been estimated, the estimated object `me` can be queried for posterior modes (line 18), enabling reproducible reporting of estimation results.

Accessing model matrices. Once the model is solved, its linearized structure can be accessed directly using

```
[Aplus,A0,Aminus,B,Q,stst,growth] = extract_first_order_structure(m);
```

This command returns the matrices of the first–order (linear) system corresponding to the equilibrium conditions in (1.12). Specifically, A_{+1}, A_0, and A_{-1} contain the coefficients on forward–looking, contemporaneous, and lagged variables, respectively, while B holds the coefficients on shocks. The transition matrix Q governs regime shifts, and `stst` and `growth` return the regime–specific steady-states and deterministic growth components.

This function thus exposes the system (1.12) in matrix form, making the underlying linearized model directly available for inspection or manipulation. It is intended for users who wish to manipulate or analyze the linearized model outside RISE, for example, by using their own routines for solution, simulation, or filtering. The output, therefore, serves as a gateway for custom algorithms. Higher–order solution components—such as the arrays that define the quadratic or cubic terms in a nonlinear perturbation—are handled internally by RISE and are beyond the scope of this chapter; users interested in those extensions should consult the RISE manual.

For many purposes, it is convenient to extract not only the raw coefficient matrices but also the solved transition and shock matrices. After solving the model, they can be obtained using

```
[T,Qfunc,stst,growth,state_vars_location,retcode] = dsge_tools.solve.load_solution(m);
```

This command provides the model's *first–order law of motion* in the form of (1.11). Here `T` collects the transition matrices (one per regime, if applicable); `Qfunc` is a function handle for evaluating the grand transition matrix; `stst` stores steady-state values for all endogenous variables; `growth` contains deterministic growth rates when relevant; `state_vars_location` identifies which elements of the endogenous vector form the state vector; and `retcode` reports the solver status (0 indicates success).

7.4 Managing Long Estimation Runs

Empirical work in RISE often involves lengthy estimation procedures. While the underlying methods are standard, the key practical challenge is to ensure that intermediate results are preserved and that estimation can be safely resumed if interrupted. This subsection, therefore, focuses on handling estimation output rather than on estimation settings themselves.

Basic estimation call. The command

```
[me,filtration] = estimate(m,'data',db);
```

estimates the parameters of a solved model using the dataset db. The function accepts many optional arguments, but only a few are essential for most users. Some of these options were introduced earlier (see Box 7); here we emphasize the choice of optimizer.

The optimizer is selected via the `'optimizer'` option. The default is MATLAB's `fmincon`, but other solvers can be specified by name, for example:

```
[me,filtration] = estimate(m,'data',db,'optimizer','bee_gate');
```

The `bee_gate` algorithm (Karaboga and Basturk, 2007) implements a stochastic, population–based search that is particularly robust in high–dimensional likelihoods but does not terminate automatically. By default, it runs until a stopping condition is met (e.g., a time or iteration limit) and then halts. Users should decide whether to continue with another round or conclude that the objective function has stabilized and the output is final.

Saving and resuming estimation. Because estimation can take hours or even days, it is good practice to *save the output after each run* and restart estimation from the most recent mode. A simple approach is shown below.

```
rndName=['Estimation_NKUS5425_',replace(char(datetime("now")),...
    {'-',':',' '},{'_','_','_'})];
pmode=get(ms,'mode');
save(rndName,'pmode','priors','m','filtration','me','db')
% Inspect output here before continuing with additional iterations if required.
for i = 1:20
me = estimate(me,'estim_start_from_mode',true);
plot_probabilities(me); % optional: visualizes whether results have stabilized
rndName=['Estimation_NKUS5425_',replace(char(datetime("now")),...
    {'-',':',' '},{'_','_','_'})];
pmode=get(ms,'mode');
save(rndName,'pmode','priors','m','filtration','me','db')
end
```

Each iteration restarts the optimizer from the previous mode rather than from random initial draws, allowing for controlled continuation and ensuring that partial progress is not lost.

Choosing the scale for MCMC chains. Line 4 in Box 9 sets `scale = 0.15`. How should the scale be chosen? A practical approach is to run several short chains (e.g., 2,000 draws each) with different `scale` values and check that the acceptance rate—reported in `results.{1,n}.stats.accept_ratio` for each $n = 1, \ldots, K$, where K is the number of chains—lies between 20% and 40%.

Restarting MCMC chains. When running long MCMC simulations, it is often useful to stop and resume later. Box 9 shows how to run one block of MCMC simulations and save the results for later analysis. In our numerical example, we ran a single block of 100,000 draws. In practice, researchers usually run shorter blocks and save results more frequently—to keep file sizes manageable and to avoid losing work if computations are interrupted. Sampling can then be resumed from the previous output:

```
results = sample(rsamplers.rwmh(energy, results, lb, ub, myOpts));
```

Here `results` contains the saved sampler state from the previous run, ensuring that the new draws continue seamlessly from where the earlier block ended.

8 Conclusion

This chapter has outlined the core workflow for Markov–switching DSGE modeling in RISE, from solving a simple New Keynesian model to estimating regime-switching specifications with real data. We have shown how the toolbox integrates all the elements needed for empirical inference—simulation, filtering and smoothing, posterior mode estimation, and Bayesian analysis via MCMC. Notably, the same object-oriented framework applies to both single-regime and switching models, so once the model is specified, subsequent steps such as likelihood evaluation, diagnostics, and posterior analysis follow a unified logic.

By combining transparent model declaration with algorithms specifically designed for switching environments, RISE makes empirical work with Markov-switching DSGE models both accessible and reproducible. The examples illustrate that even compact models can recover meaningful regime dynamics and produce results consistent with the literature. With the ability to extend models, add additional Markov chains, and incorporate richer data, researchers and students can build directly on these foundations for applied analysis and policy evaluation.

Beyond the applications illustrated in this chapter, RISE provides a broader platform for dynamic modeling. It accommodates a range of structures beyond DSGE frameworks, including general state-space representations, vector autoregressions, and reduced-form models with switching parameters. The toolbox supports likelihood-based estimation, Bayesian inference using several alternative samplers, and several methods for evaluating the marginal data density, enabling systematic model comparison and robustness analysis. Its filtering and smoothing routines extend to nonlinear and regime-dependent systems, allowing users to extract latent states and compute regime probabilities even in partially observed environments. Additional tools support diagnostics and model validation—residual analysis, identification checks, and convergence assessment—as well as utilities for policy evaluation, forecasting, and simulation under user-defined constraints. In this way, RISE functions as a comprehensive environment for solving, estimating, and evaluating dynamic models under a broad spectrum of structural and statistical specifications.

The preceding chapters in this volume develop the theoretical and econometric foundations for modern macroeconomic analysis—from Bayesian inference and state-space methods to calibration, simulation, and policy evaluation within DSGE frameworks. This chapter complements those contributions by illustrating how the techniques discussed throughout the volume can be implemented and extended within a unified computational environment. In doing so, it connects the analytical methods presented in earlier chapters to practical applications using RISE, particularly in models with regime changes and nonlinear dynamics.

Notes

[1] Stochastic singularity occurs when the model implies that certain variables are deterministic functions of others, making the error covariance matrix singular and preventing estimation.

[2] Because of the factorization in (1.17), it is *optimal* to implement these computations sequentially, even when the entire sample is available.

[3] We use a slightly more general state–space form here than in our example NK model.

[4] See Cantore et al. (2026) in this volume.

[5]We use the FRED series CPIAUCSL_PC1, GDPC1_PC1, and FEDFUNDS.

[6]This example sets priors in terms of means and standard deviations. For an alternative way to set priors in terms of quantiles, see the manual.

[7]Because the data come from FRED, they could alternatively be imported directly using fetch_fred; we use an Excel file here to illustrate a more general workflow.

[8]Demeaning is required because the model is written in terms of log deviations from the steady state.

[9]For readability, we do not reproduce the full plotting code here. All scripts and data used to generate the figures in this chapter are provided as supplementary materials at https://github.com/jmaih/.

[10]For a rigorous treatment of this algorithm and Bayesian inference more broadly, see Bauwens and Korobilis (2026) in this volume.

[11]The diagnostic plots shown in Figure 3 are illustrative. When the code is rerun using the supplementary materials, the exact realizations may differ due to stochastic variation in the MCMC draws, although the qualitative patterns discussed in the text are the same.

[12]We omit plotting commands from the codebox, but they are done in the same way as in lines 31-49 in Box 12.

[13]The model is stationarized; for an example with non-stationary variables, see the manual.

References

An, S. and F. Schorfheide (2007). Bayesian Analysis of DSGE Models. *Econometric Reviews 26*(2-4), 113–172.

Bauwens, L. and D. Korobilis (2026). Bayesian methods. In N. Hashimzade and M. A. Thornton (Eds.), *Handbook of Research Methods and Applications in Empirical Macroeconomics*, pp. ??–?? Cheltenham, UK and Northampton, MA, USA: Edward Elgar Publishing.

Blanchard, O. and C. Kahn (1980). The Solution of Linear Difference Models Under Rational Expectations. *Econometrica 48*, 1305–1311.

Blom, H. A. and Y. Bar-Shalom (1988). The interacting multiple model algorithm for systems with Markovian switching coefficients. *IEEE transactions on Automatic Control 33*(8), 780–783.

Blom, H. A. P. (1984). An efficient filter for abruptly changing systems. In *The 23rd IEEE conference on decision and control*, pp. 656–658. IEEE.

Calvo, G. (1983). Staggered Prices in a Utility-Maximising Framework. *Journal of Monetary Economics 12*, 383–398.

Cantore, C., V. J. Gabriel, P. Levine, J. Pearlman, and B. Yang (2026). The science and art of dsge modelling: II – model comparisons, model validation, policy analysis and general discussion. In N. Hashimzade and M. A. Thornton (Eds.), *Handbook of Research Methods and Applications in Empirical Macroeconomics*, pp. ??–?? Cheltenham, UK and Northampton, MA, USA: Edward Elgar Publishing.

Chen, X., T. Kirsanova, and C. Leith (2017). How Optimal is US Monetary Policy? *Journal of Monetary Economics 92*, 96–111.

do Valle Costa, O. L., M. D. Fragoso, and R. P. Marques (2005). *Discrete-Time Markov Jump Linear Systems*. London: Springer-Verlag.

Galí, J. and M. Gertler (1999). Inflation Dynamics: A Structural Econometric Analysis. *Journal of Monetary Economics 44*, 195–222.

Guerrón-Quintana, P. A. and J. M. Nason (2026). Bayesian estimation of dsge models. In N. Hashimzade and M. A. Thornton (Eds.), *Handbook of Research Methods and Applications in Empirical Macroeconomics*, Chapter 21, pp. ??–?? Cheltenham, UK and Northampton, MA, USA: Edward Elgar Publishing.

Hamilton, J. D. (1989). A new approach to the economic analysis of nonstationary time series and the business cycle. *Econometrica 57*(2), 357–384.

Hashimzade, N., O. Kirsanov, T. Kirsanova, and J. Maih (2024). On Bayesian Filtering for Markov Regime Switching Models. Norges Bank Working Paper 8/2024.

Judd, K. L. (1996). Approximation, perturbation, and projection methods in economic analysis. *Handbook of computational economics 1*, 509–585.

Kalman, R. E. (1960). A new approach to linear filtering and prediction problems. *Journal of Basic Engineering 82*(1), 35–45.

Karaboga, D. and B. Basturk (2007). A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm. *Journal of Global Optimization 39*, 459–471.

Kim, C. and C. Nelson (1999). *State-space Models with Regime Switching: Classical and Gibbs-sampling Approaches with Applications*. MIT Press.

Kim, C.-J. (1994). Dynamic linear models with Markov-switching. *Journal of Econometrics 60*(1-2), 1–22.

Klein, P. (2000). Using the Generalized Schur Form to Solve a Multivariate Linear Rational Expectations Model. *Journal of Economic Dynamics and Control 24*(10), 1405–1423.

Leith, C., T. Kirsanova, C. Machado, and A. P. Ribeiro (2025). (Re)Evaluating recent macroeconomic policy in the US. *European Economic Review 178*, 105091.

Lubik, T. and F. Schorfheide (2005). "A Bayesian Look at New Open Economy Macroeconomics". In *NBER Macroeconomics Annual 20*, pp. 313–366.

Maih, J. (2015). Efficient perturbation methods for solving regime-switching DSGE models. Working Paper 2015/01, Norges Bank.

Meng, X.-L. and W. H. Wong (1996). Simulating ratios of normalizing constants via a simple identity: a theoretical exploration. *Statistica Sinica 6*(4), 831–860.

Pollock, D. S. G. (2026). Filtering macroeconomic data. In N. Hashimzade and M. A. Thornton (Eds.), *Handbook of Research Methods and Applications in Empirical Macroeconomics*, Chapter 5, pp. ??–?? Cheltenham, UK and Northampton, MA, USA: Edward Elgar Publishing.

Proietti, T. and A. Luati (2026). Maximum likelihood estimation of time series models: The kalman filter and beyond. In N. Hashimzade and M. A. Thornton (Eds.), *Handbook of Research Methods and Applications in Empirical Macroeconomics*, Chapter 15, pp. ??–?? Cheltenham, UK and Northampton, MA, USA: Edward Elgar Publishing.

Saito, Y. and T. Mitsui (1996). Stability analysis of numerical schemes for stochastic differential equations. *SIAM Journal on Numerical Analysis 33*(6), 2254–2267.

Sims, C. A. and T. Zha (2006). Were There Regime Switches in US Monetary Policy. *American Economic Review 96*(1), 54–81.