# Dynare Reference Manual

***Release 8-unstable-2026-10-23-1545-ac3bae03***

**Dynare Team**

**Oct 23, 2026**

# CONTENTS


**1 Introduction** **3**
- 1.1 What is Dynare? . . . 3
- 1.2 Documentation sources . . . 4
- 1.3 Citing Dynare in your research . . . 4

**2 Installation and configuration** **5**
- 2.1 Software requirements . . . 5
- 2.2 Installation of Dynare . . . 5
  - 2.2.1 On Windows . . . 5
  - 2.2.2 On GNU/Linux . . . 6
  - 2.2.3 On macOS . . . 6
    - 2.2.3.1 With MATLAB . . . 6
    - 2.2.3.2 With Octave . . . 6
  - 2.2.4 On FreeBSD . . . 7
  - 2.2.5 For other systems . . . 7
- 2.3 Compiler installation . . . 7
  - 2.3.1 Prerequisites on Windows . . . 7
  - 2.3.2 Prerequisites on GNU/Linux . . . 7
  - 2.3.3 Prerequisites on macOS . . . 7
    - 2.3.3.1 With MATLAB . . . 7
    - 2.3.3.2 With Octave . . . 8
- 2.4 Optional installation of Panua PARDISO . . . 8
  - 2.4.1 On Windows . . . 8
- 2.5 Configuration . . . 8
  - 2.5.1 For MATLAB . . . 8
  - 2.5.2 For Octave . . . 9
  - 2.5.3 Some words of warning . . . 9

**3 Running Dynare** **11**
- 3.1 Dynare invocation . . . 11
- 3.2 Dynare hooks . . . 18
- 3.3 Understanding Preprocessor Error Messages . . . 18

**4 The model file** **21**
- 4.1 Conventions . . . 21
- 4.2 Variable declarations . . . 22
  - 4.2.1 On-the-fly Model Variable Declaration . . . 26
- 4.3 Expressions . . . 27
  - 4.3.1 Parameters and variables . . . 27
    - 4.3.1.1 Inside the model . . . 27
    - 4.3.1.2 Outside the model . . . 28
  - 4.3.2 Operators . . . 28
  - 4.3.3 Functions . . . 29
    - 4.3.3.1 Built-in functions . . . 29

4.3.3.2 External functions . . . 30
4.3.4 A few words of warning in stochastic context . . . 31
4.4 Parameter initialization . . . 31
4.5 Model declaration . . . 32
4.6 Auxiliary variables . . . 38
4.7 Initial and terminal conditions . . . 38
4.8 Shocks on exogenous variables . . . 52
4.9 Other general declarations . . . 57
4.10 Steady state . . . 57
4.10.1 Finding the steady state with Dynare nonlinear solver . . . 57
4.10.2 Providing the steady state to Dynare . . . 62
4.10.3 Replace some equations during steady-state computations . . . 63
4.11 Getting information about the model . . . 64
4.12 Deterministic simulation . . . 66
4.12.1 Perfect foresight . . . 66
4.12.2 Perfect foresight with expectation errors . . . 75
4.12.3 Controlling the path of endogenous variables . . . 81
4.12.4 Alternative syntax for specifying deterministic shocks . . . 83
4.13 Stochastic solution and simulation . . . 86
4.13.1 Computing the stochastic solution . . . 86
4.13.2 Typology and ordering of variables . . . 93
4.13.3 First-order approximation . . . 94
4.13.4 Second-order approximation . . . 95
4.13.5 Third-order approximation . . . 95
4.13.6 Higher-order approximation . . . 96
4.14 Extended path . . . 96
4.15 Occasionally binding constraints (OccBin) . . . 98
4.16 Estimation based on likelihood . . . 107
4.16.1 Specifying the observables, and trends . . . 107
4.16.2 Specifying the estimated parameters and priors . . . 108
4.16.3 The estimation command . . . 112
4.16.3.1 Data treatment options . . . 113
4.16.3.2 Graph output options . . . 114
4.16.3.3 Mode computing options . . . 115
4.16.3.4 Hessian/derivative options . . . 121
4.16.3.5 Mode-check diagnostics . . . 122
4.16.3.6 Kalman filtering and likelihood options . . . 122
4.16.3.7 Skewed Kalman filter options . . . 126
4.16.3.8 Posterior objects . . . 126
4.16.3.9 Variable selection for posterior objects . . . 128
4.16.3.10 Model solution options . . . 128
4.16.3.11 Posterior sampling options . . . 129
4.16.3.12 MCMC options . . . 135
4.16.3.13 Convergence diagnostics . . . 137
4.16.3.14 DSGE-VAR . . . 138
4.16.3.15 Non-linear filter options . . . 138
4.16.4 Estimation output variables . . . 141
4.16.5 Adding information about unobservable variables . . . 150
4.16.6 The data command . . . 151
4.16.7 Setting the initial state for the Kalman filter . . . 152
4.16.8 Bayesian VAR estimation . . . 152
4.17 Estimation based on moments . . . 152
4.17.1 Method of moments specific blocks . . . 153
4.17.2 method_of_moments command . . . 156
4.17.2.1 Necessary Options . . . 156
4.17.2.2 Common Options . . . 157
4.17.2.3 SMM specific options . . . 158
4.17.2.4 GMM specific options . . . 158

4.17.2.5 IRF matching specific options . . . 158
4.17.2.6 General options . . . 159
4.17.2.7 Data options . . . 159
4.17.2.8 Optimization options . . . 160
4.17.2.9 Bayesian estimation options . . . 160
4.17.2.10 Numerical algorithms options . . . 162
4.17.3 Method of moments specific outputs . . . 163
4.18 Model Comparison . . . 167
4.19 Shock Decomposition . . . 168
4.20 Calibrated Smoother . . . 176
4.21 Forecasting . . . 178
4.22 Optimal policy . . . 187
4.22.1 Optimal policy under commitment (Ramsey) . . . 188
4.22.2 Optimal policy under discretion . . . 191
4.22.3 Optimal Simple Rules (OSR) . . . 191
4.23 Sensitivity and identification analysis . . . 195
4.23.1 Performing sensitivity analysis . . . 195
4.23.2 IRF/Moment calibration . . . 198
4.23.3 Performing identification analysis . . . 200
4.23.4 Types of analysis and output files . . . 202
4.23.4.1 Sampling . . . 202
4.23.4.2 Stability Mapping . . . 203
4.23.4.3 IRF/Moment restrictions . . . 203
4.23.4.4 Reduced-Form Mapping . . . 204
4.23.4.5 RMSE . . . 205
4.23.4.6 Screening Analysis . . . 206
4.23.4.7 Identification Analysis . . . 206
4.24 Markov-switching SBVAR . . . 207
4.25 Epilogue Variables . . . 216
4.26 Heterogeneity . . . 217
4.26.1 Declaring Heterogeneous Agent Models . . . 217
4.26.1.1 Heterogeneity dimension . . . 217
4.26.1.2 Heterogeneous Variable Declarations . . . 217
4.26.1.3 Heterogeneous Agent Model Block . . . 218
4.26.1.4 Heterogeneous Shocks Block . . . 219
4.26.1.5 Aggregate Variable Declarations . . . 219
4.26.1.6 Aggregate Shocks Block . . . 219
4.26.1.7 Aggregate Model Block . . . 219
4.26.2 Solving Heterogeneous Agent Models . . . 220
4.26.2.1 Loading a pre-computed steady state . . . 220
4.26.2.2 Computing the steady state . . . 222
4.26.2.3 Helper functions . . . 226
4.26.2.4 Solving for the dynamics . . . 227
4.26.2.5 Simulating . . . 228
4.27 Semi-structural models . . . 229
4.27.1 Auxiliary models . . . 229
4.27.2 VAR expectations . . . 232
4.27.3 PAC equation . . . 234
4.27.4 Estimation of a PAC equation . . . 238
4.28 Displaying and saving results . . . 239
4.29 Macro processing language . . . 240
4.29.1 Macro expressions . . . 240
4.29.2 Macro directives . . . 244
4.29.3 Typical usages . . . 248
4.29.3.1 Modularization . . . 248
4.29.3.2 Indexed sums of products . . . 248
4.29.3.3 Multi-country models . . . 249
4.29.3.4 Endogeneizing parameters . . . 249

4.29.4 MATLAB/Octave loops versus macro processor loops . . . 250
4.30 Verbatim inclusion . . . 251
4.31 Misc commands . . . 251

**5 Parallel Execution and Configuration File** **255**
5.1 Parallel Computing Toolbox (PCT) . . . 255
5.2 Cluster-based Parallel Configuration . . . 255
5.2.1 The Configuration File . . . 256
5.2.2 Windows Step-by-Step Guide . . . 259
5.3 Dynare Configuration . . . 260

**6 Time Series** **263**
6.1 Dates . . . 263
6.1.1 Dates in a mod file . . . 263
6.1.2 The dates class . . . 265
6.2 The dseries class . . . 276
6.3 X-13 ARIMA-SEATS interface . . . 310
6.4 Miscellaneous . . . 314
6.4.1 Time aggregation . . . 314
6.4.2 Create time series with a univariate model . . . 315

**7 Reporting** **317**

**8 Examples** **329**
8.1 Stochastic simulations . . . 329
8.2 Estimation . . . 329
8.3 Perfect foresight . . . 330
8.4 Optimal policy . . . 330
8.5 Heterogeneity . . . 330
8.6 OccBin . . . 331
8.7 Macroprocessor . . . 331
8.8 Reporting . . . 331
8.9 Semistructural . . . 331

**9 Dynare misc commands** **333**

**Bibliography** **339**

**Index** **345**

Currently the development team of Dynare is composed of:

- Stéphane Adjemian (Le Mans Université, Gains)
- Michel Juillard (Banque de France)
- Sumudu Kankanamge (Le Mans Université and CEPREMAP)
- Frédéric Karamé (Le Mans Université, Gains and CEPREMAP)
- Willi Mutschler (University of Tübingen)
- Johannes Pfeifer (University of the Bundeswehr Munich)
- Marco Ratto (European Commission, Joint Research Centre - JRC)
- Normann Rion (CEPREMAP)
- Sébastien Villemot (CEPREMAP)

The following people used to be members of the team:

- Houtan Bastani
- Abdeljabar Benzougar
- Alejandro Buesa
- Fabrice Collard
- Assia Ezzeroug
- Dóra Kocsis
- Stéphane Lhuissier
- Junior Maih
- Ferhat Mihoubi
- George Perendia

Copyright © 1996-2026, Dynare Team.

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts.

A copy of the license can be found at https://www.gnu.org/licenses/fdl.txt.

CHAPTER

ONE

# INTRODUCTION

## 1.1 What is Dynare?

Dynare is a software platform for handling a wide class of economic models, in particular dynamic stochastic general equilibrium (DSGE) and overlapping generations (OLG) models. Its history is described in Cherrier *et al.* (2023). The models solved by Dynare include those relying on the *rational expectations* hypothesis, wherein agents form their expectations about the future in a way consistent with the model. But Dynare is also able to handle models where expectations are formed differently: on one extreme, models where agents perfectly anticipate the future; on the other extreme, models where agents have limited rationality or imperfect knowledge of the state of the economy and, hence, form their expectations through a learning process. In terms of types of agents, models solved by Dynare can incorporate consumers, productive firms, governments, monetary authorities, investors and financial intermediaries. Some degree of heterogeneity can be achieved by including several distinct classes of agents in each of the aforementioned agent categories.

Dynare offers a user-friendly and intuitive way of describing these models. It is able to perform simulations of the model given a calibration of the model parameters and is also able to estimate these parameters given a dataset. In practice, the user will write a text file containing the list of model variables, the dynamic equations linking these variables together, the computing tasks to be performed and the desired graphical or numerical outputs.

A large panel of applied mathematics and computer science techniques are internally employed by Dynare: multivariate nonlinear solving and optimization, matrix factorizations, local functional approximation, Kalman filters and smoothers, MCMC techniques for Bayesian estimation, graph algorithms, optimal control, ...

Various public bodies (central banks, ministries of economy and finance, international organisations) and some private financial institutions use Dynare for performing policy analysis exercises and as a support tool for forecasting exercises. In the academic world, Dynare is used for research and teaching purposes in postgraduate macroeconomics courses.

Dynare is a free software, which means that it can be downloaded free of charge, that its source code is freely available, and that it can be used for both non-profit and for-profit purposes. Most of the source files are covered by the GNU General Public Licence (GPL) version 3 or later (there are some exceptions to this, see the file license.txt in Dynare distribution). It is available for the Windows, macOS, and Linux platforms and is fully documented through a reference manual. Part of Dynare is programmed in C++, while the rest is written using the MATLAB programming language. The latter implies that commercially-available MATLAB software is required in order to run Dynare. However, as an alternative to MATLAB, Dynare is also able to run on top of GNU Octave (basically a free clone of MATLAB): this possibility is particularly interesting for students or institutions who cannot afford, or do not want to pay for, MATLAB and are willing to bear the concomitant performance loss.

The development of Dynare is mainly done at CEPREMAP by a core team of researchers who devote part of their time to software development. Increasingly, the developer base is expanding, as tools developed by researchers outside CEPREMAP are integrated into Dynare. Financial support is provided by CEPREMAP, Banque de France, and DSGE-net (an international research network for DSGE modeling).

Interaction between developers and users of Dynare is central to the project. A web forum is available for users who have questions about the usage of Dynare or who want to report bugs. Current known and fixed bugs are listed on the Dynare wiki. Issues or whishes can be reported on our Git repository. Training sessions are given through the Dynare Summer School, which is organized every year and is attended by about 40 people. Finally, priorities in terms of future developments and features to be added are decided in cooperation with the institutions providing financial support.

## 1.2 Documentation sources

The present document is the reference manual for Dynare. It documents all commands and features in a systematic fashion.

Other useful sources of information include the Dynare wiki and the Dynare forums.

## 1.3 Citing Dynare in your research

You should cite Dynare if you use it in your research. The recommended way to do this is to cite the present manual, as:

> Stéphane Adjemian, Michel Juillard, Frédéric Karamé, Willi Mutschler, Johannes Pfeifer, Marco Ratto, Normann Rion and Sébastien Villemot (2026), "Dynare: Reference Manual, Version 7," *Dynare Working Papers*, 87, CEPREMAP

For convenience, you can copy and paste the following into your BibTeX file:

```
@TechReport{Adjemianetal2026,
  author      = {Adjemian, St\'ephane and Juillard, Michel and
                 Karam\'e, Fr\'ed\'eric and Mutschler, Willi and
                 Pfeifer, Johannes and Ratto, Marco and
                 Rion, Normann and Villemot, S\'ebastien},
  title       = {Dynare: Reference Manual, Version 7},
  year        = {2026},
  institution = {CEPREMAP},
  type        = {Dynare Working Papers},
  number      = {87},
}
```

If you want to give a URL, use the address of the Dynare website: https://www.dynare.org.

CHAPTER

TWO

# INSTALLATION AND CONFIGURATION

## 2.1 Software requirements

Packaged versions of Dynare are available for Windows 11, several GNU/Linux distributions (Debian, Ubuntu, Linux Mint, openSUSE), macOS (26 "Tahoe"), and FreeBSD. Dynare should work on other systems, but some compilation steps are necessary in that case.

In order to run Dynare, you need one of the following:

- MATLAB, any version ranging from 9.8 (R2020a) to 26.2 (R2026b);
- GNU Octave, any version ranging from 8.4.0 to 11.3.0, with the statistics and datatypes packages. Note however that the Dynare installer for Windows requires a more specific version of Octave, as indicated on the download page.

The following optional extensions are also useful to benefit from extra features, but are in no way required:

- If under MATLAB: the
  - Optimization Toolbox (providing various optimizers like `fminsearch`, `fmincon`, or `fminunc`, used in e.g. `mode_compute`, `opt_algo` or `ALGO`),
  - Statistics Toolbox (for faster and sometimes more robust implementations of statistical distributions),
  - Global Optimization Toolbox (for `particleswarm` and `simulannealbnd`, used in e.g. `mode_compute`, `opt_algo` or `ALGO`)
  - Control System Toolbox (for the Lyapunov solver `dlyapchol` triggered with `lyapunov=square_root_solver`)
  - Parallel Computing Toolbox (to speed up the `dime` sampler)
- If under Octave, the following packages: control, io, optim.

## 2.2 Installation of Dynare

After installation, Dynare can be used in any directory on your computer. It is best practice to keep your model files in directories different from the one containing the Dynare toolbox. That way you can upgrade Dynare and discard the previous version without having to worry about your own files.

### 2.2.1 On Windows

Execute the automated installer called `dynare-x.y-win.exe` (where `x.y` is the version number), and follow the instructions. The default installation directory is `c:\dynare\x.y`.

After installation, this directory will contain several sub-directories, among which are `matlab`, `mex` and `doc`.

The installer will also add an entry in your Start Menu with a shortcut to the documentation files and uninstaller.

Note that you can have several versions of Dynare coexisting (for example in `c:\dynare`), as long as you correctly adjust your path settings (see *Some words of warning*).

Also note that it is possible to do a silent installation, by passing the `/S` flag to the installer on the command line. This can be useful when doing an unattended installation of Dynare on a computer pool.

### 2.2.2 On GNU/Linux

On Debian, Ubuntu and Linux Mint, the Dynare package can be installed with: `apt install dynare`. This will give a fully-functional Dynare installation usable with Octave. If you have MATLAB installed, you should also do: `apt install dynare-matlab` (under Debian, this package is in the `contrib` section). Documentation can be installed with `apt install dynare-doc`. The status of those packages can be checked at those pages:

- Package status in Debian
- Package status in Ubuntu
- Package status in Linux Mint

On Arch Linux, the Dynare package is not in the official repositories, but is available in the Arch User Repository. The needed sources can be downloaded from the package status in Arch Linux.

There is also a Dynare package for openSUSE, see the package status in openSUSE.

Dynare will be installed under `/usr/lib/dynare` (or `/usr/lib64/dynare` on openSUSE). Documentation will be under `/usr/share/doc/dynare` (only on Debian, Ubuntu and Linux Mint).

### 2.2.3 On macOS

**Warning**

Installing into `/Applications/dynare` might fail if you have older versions of Dynare already installed in `/Applications/Dynare`. To fix this, modify the ownership by executing the following command in Terminal.app:

```
sudo chown -R "$USER":staff /Applications/Dynare
```

Alternatively, you can modify the installation path in the automated installed using *Customize* and *Location*. After installation, the folder will contain several sub-directories, among which are `matlab`, `mex`, and `doc`. Several versions of Dynare can coexist (by default in `/Applications/Dynare`), as long as you correctly adjust your path settings (see *Some words of warning*).

#### 2.2.3.1 With MATLAB

To install Dynare for use with MATLAB, execute the automated installer called `dynare-x.y.pkg` (where *x.y* is the version number), and follow the instructions. Note that only the Apple Silicon version of MATLAB is supported.

This installation does not require administrative privileges. If for some reason admin rights are requested, use *Change Install Location* and select *Install for me only*. The default installation directory is `/Applications/Dynare/x.y-arch`.

It is recommended to install the Xcode Command Line Tools (this is an Apple product) and GCC via Homebrew (see *Prerequisites on macOS*).

To uninstall Dynare, simply delete the folder where you installed the program. The package installer does not put any files anywhere else in the system.

#### 2.2.3.2 With Octave

We don't provide Dynare packages for macOS with Octave support, but there is a Dynare package with Octave support in Homebrew.

Once Homebrew is installed, run a terminal and install Dynare (and Octave) by typing the following:

```
brew install dynare
```

Then open Octave by running the following in the same terminal:

```
octave --gui
```

Finally, at the Octave prompt, install some add-ons (you only have to do it once):

```
octave:1> pkg install io datatypes statistics control struct optim
```

If you want to use the *x13* functionality of *dseries*, you also need to build the *x13as* binary.[1]

### 2.2.4 On FreeBSD

A FreeBSD port for Dynare is available. It can be installed with:

```
pkg install dynare
```

### 2.2.5 For other systems

You need to download Dynare source code from the Dynare website and unpack it somewhere.

Then you will need to recompile the pre-processor and the dynamic loadable libraries. Please refer to README.md.

## 2.3 Compiler installation

### 2.3.1 Prerequisites on Windows

There are no prerequisites on Windows. Dynare now ships a compilation environment that can be used with the *use_dll* option.

### 2.3.2 Prerequisites on GNU/Linux

Users of MATLAB under GNU/Linux need a working compilation environment installed. Under Debian, Ubuntu or Linux Mint, it can be installed via `apt install build-essential`.

Users of Octave under GNU/Linux should install the package for MEX file compilation (under Debian, Ubuntu or Linux Mint, it can be done via `apt install liboctave-dev`).

### 2.3.3 Prerequisites on macOS

#### 2.3.3.1 With MATLAB

Dynare now ships a compilation environment that can be used with the *use_dll* option. To install this environment correctly, the Dynare installer ensures that the Xcode Command Line Tools (an Apple product) have been installed on a system folder. To install the Xcode Command Line Tools yourself, simply type `xcode-select --install` into the terminal (`/Applications/Utilities/Terminal.app`) prompt. Additionally, to make MATLAB aware that you agree to the terms of Xcode, run the following two commands in the Terminal prompt:

```
CLT_VERSION=$(pkgutil --pkg-info=com.apple.pkg.CLTools_Executables | grep version |␣
↪awk '{print $2}' | cut -d'.' -f1-2)
defaults write com.apple.dt.Xcode IDEXcodeVersionForAgreedToGMLicense "${CLT_VERSION}"
defaults read com.apple.dt.Xcode IDEXcodeVersionForAgreedToGMLicense
```

Otherwise, you will see a warning that Xcode is installed, but its license has not been accepted. You can check this e.g. by running the following command in the MATLAB command window:

```
mex -setup
```

[1] See the instructions at https://forum.dynare.org/t/missing-installation-package/27350/4.

Moreover, we recommend making use of optimized compilation flags when using *use_dll* and for this you need to install GCC via Homebrew:

```
brew install gcc
```

If you have installed GCC via Homebrew, Dynare will automatically prefer it for *use_dll*, otherwise it will fall back to Apple Clang in `/usr/bin/clang`.

#### 2.3.3.2 With Octave

The compiler can be installed via Homebrew. In a terminal, run:

```
brew install gcc
```

## 2.4 Optional installation of Panua PARDISO

In order to use the values `9` or `10` of *stack_solve_algo*, the user must manually copy the PARDISO library into the directory containing Dynare MEX files, and also provide a license file. This is needed because Panua PARDISO is a proprietary software, and thus cannot be distributed alongside Dynare.

### 2.4.1 On Windows

The library file `libpardiso.dll` from the Panua PARDISO distribution should be copied under the MEX directory, which by default has the form `C:\dynare\<version>\mex\matlab\win64-<version_range>\` for MATLAB or `C:\dynare\<version>\mex\octave\win64\` for Octave. Then the license file `panua.lic` should go into the home directory (typically `C:\Users\<YOUR_LOGIN>\`).

## 2.5 Configuration

### 2.5.1 For MATLAB

You need to add the `matlab` subdirectory of your Dynare installation to MATLAB path. You have two options for doing that:

- Using the `addpath` command in the MATLAB command window:

  Under Windows, assuming that you have installed Dynare in the standard location, and replacing `x.y` with the correct version number, type:

  ```
  >> addpath c:/dynare/x.y/matlab
  ```

  Under GNU/Linux, type:

  ```
  >> addpath /usr/lib/dynare/matlab
  ```

  Under macOS, assuming that you have installed Dynare in the standard location, and replacing `x.y` with the correct version number, type:

  ```
  >> addpath /Applications/Dynare/x.y/matlab
  ```

  MATLAB will not remember this setting next time you run it, and you will have to do it again.

- Via the menu entries:

  Select the "Set Path" entry in the "File" menu, then click on "Add Folder…", and select the `matlab` subdirectory of 'your Dynare installation. Note that you *should not* use "Add with Subfolders…". Apply the settings by clicking on "Save". Note that MATLAB will remember this setting next time you run it.

### 2.5.2 For Octave

You need to add the `matlab` subdirectory of your Dynare installation to Octave path, using the `addpath` at the Octave command prompt.

Under Windows, assuming that you have installed Dynare in the standard location, and replacing "*x.y*" with the correct version number, type:

```
octave:1> addpath c:/dynare/x.y/matlab
```

Under Debian, Ubuntu or Linux Mint, there is no need to use the `addpath` command; the packaging does it for you. Under Arch Linux, you need to do:

```
octave:1> addpath /usr/lib/dynare/matlab
```

Under macOS, assuming you have installed Dynare via Homebrew:

```
octave:1> addpath /usr/local/lib/dynare/matlab
```

If you don't want to type this command every time you run Octave, you can put it in a file called `.octaverc` in your home directory (under Windows this will generally be `c:\Users\USERNAME` while under macOS it is `/Users/USERNAME/`). This file is run by Octave at every startup.

### 2.5.3 Some words of warning

You should be very careful about the content of your MATLAB or Octave path. You can display its content by simply typing `path` in the command window.

The path should normally contain system directories of MATLAB or Octave, and some subdirectories of your Dynare installation. You have to manually add the `matlab` subdirectory, and Dynare will automatically add a few other subdirectories at runtime (depending on your configuration). You must verify that there is no directory coming from another version of Dynare than the one you are planning to use.

You have to be aware that adding other directories (on top of the Dynare folders) to your MATLAB or Octave path can potentially create problems if any of your M-files have the same name as a Dynare file. Your routine would then override the Dynare routine, making Dynare unusable.

**Warning**

Never add all the subdirectories of the `matlab` folder to the MATLAB or Octave path. You must let Dynare decide which subdirectories have to be added to the MATLAB or Octave path. Otherwise, you may end up with a non-optimal or unusable installation of Dynare.

CHAPTER

# THREE

# RUNNING DYNARE

In order to give instructions to Dynare, the user has to write a *model file* whose filename extension must be `.mod` or `.dyn`. This file contains the description of the model and the computing tasks required by the user. Its contents are described in *The model file*.

## 3.1 Dynare invocation

Once the model file is written, Dynare is invoked using the `dynare` command at the MATLAB or Octave prompt (with the filename of the `.mod` given as argument).

In practice, the handling of the model file is done in two steps: in the first one, the model and the processing instructions written by the user in a *model file* are interpreted and the proper MATLAB or Octave instructions are generated; in the second step, the program actually runs the computations. Both steps are triggered automatically by the `dynare` command.

**MATLAB/Octave command: dynare** FILENAME[.mod] [OPTIONS...]

This command launches Dynare and executes the instructions included in `FILENAME.mod`. This user-supplied file contains the model and the processing instructions, as described in *The model file*. The options, listed below, can be passed on the command line, following the name of the `.mod` file or in the first line of the `.mod` file itself (see below).

Dynare begins by launching the preprocessor on the `.mod file`. By default (unless the `use_dll` option has been given to `model`), the preprocessor creates three intermediary files:

- `+FILENAME/driver.m`

  Contains variable declarations, and computing tasks.

- `+FILENAME/dynamic.m`

  Contains the dynamic model equations. Note that Dynare might introduce auxiliary equations and variables (see *Auxiliary variables*). Outputs are the residuals of the dynamic model equations in the order the equations were declared and the Jacobian of the dynamic model equations. For higher order approximations also the Hessian and the third-order derivatives are provided. When computing the Jacobian of the dynamic model, the order of the endogenous variables in the columns is stored in `M_.lead_lag_incidence`. The rows of this matrix represent time periods: the first row denotes a lagged (time t-1) variable, the second row a contemporaneous (time t) variable, and the third row a leaded (time t+1) variable. The columns of the matrix represent the endogenous variables in their order of declaration. A zero in the matrix means that this endogenous does not appear in the model in this time period. The value in the `M_.lead_lag_incidence` matrix corresponds to the column of that variable in the Jacobian of the dynamic model. Example: Let the second declared variable be `c` and the `(3,2)` entry of `M_.lead_lag_incidence` be 15. Then the 15th column of the Jacobian is the derivative with respect to `c(+1)`.

- `+FILENAME/static.m`

Contains the long run static model equations. Note that Dynare might introduce auxiliary equations and variables (see *Auxiliary variables*). Outputs are the residuals of the static model equations in the order the equations were declared and the Jacobian of the static equations. Entry `(i,j)` of the Jacobian represents the derivative of the $i$ -th static model equation with respect to the $j$ -th model variable in declaration order.

These files may be looked at to understand errors reported at the simulation stage.

`dynare` will then run the computing tasks by executing `+FILENAME/driver.m`. If a user needs to rerun the computing tasks without calling the preprocessor (or without calling the *`dynare`* command), for instance because he has modified the script, he just has to type the following on the command line:

```
>> FILENAME.driver
```

A few words of warning are warranted here: under Octave the filename of the `.mod` file should be chosen in such a way that the generated `.m` files described above do not conflict with `.m` files provided by Octave or by Dynare. Not respecting this rule could cause crashes or unexpected behaviour. In particular, it means that the `.mod` file cannot be given the name of an Octave or Dynare command. For instance, under Octave, it also means that the `.mod` file cannot be named `test.mod` or `example.mod`.

**Note**

Note on Quotes

When passing command line options that contain a space (or, under Octave, a double quote), you must surround the entire option (keyword and argument) with single quotes, as in the following example.

*Example*

Call Dynare with options containing spaces

```
>> dynare <<modfile.mod>> '-DA=[ i in [1,2,3] when i > 1 ]' 'conffile=C:\
↪User\My Documents\config.txt'
```

*Options*

**`noclearall`**

By default, `dynare` deletes all the global variables and the functions using persistent variables, in order to benefit from the JIT (just-in-time) compilation. This option instructs `dynare` not to clear those.

**`onlyclearglobals`**

By default, `dynare` deletes all the global variables and the functions using persistent variables, in order to benefit from the JIT (just-in-time) compilation. This option instructs `dynare` to clear only its own global variables (*i.e.* `M_`, `options_`, `oo_`, `estim_params_`, `dataset_`, `dataset_info` and `estimation_info`), leaving the other variables in the workspace, and not clearing functions using persistent variables.

**`debug`**

Instructs the preprocessor to write some debugging information about the scanning and parsing of the `.mod` file.

**`notmpterms`**

Instructs the preprocessor to omit temporary terms in the static and dynamic files; this generally decreases performance, but is used for debugging purposes since it makes the static and dynamic files more readable.

**`savemacro[=FILENAME]`**

Instructs `dynare` to save the intermediary file which is obtained after macro processing (see *Macro processing language*); the saved output will go in the file specified, or if no file is specified

in FILENAME_macroexp.mod. See the *note on quotes* for info on passing a FILENAME argument containing spaces.

**onlymacro**

Instructs the preprocessor to only perform the macro processing step, and stop just after. Useful for debugging purposes or for using the macro processor independently of the rest of Dynare toolbox.

**linemacro**

Instructs the macro preprocessor to include @#line directives specifying the line on which macro directives were encountered and expanded from. Only useful in conjunction with *savemacro*.

**onlymodel**

Instructs the preprocessor to print only information about the model in the driver file; no Dynare commands (other than the shocks statement and parameter initializations) are printed and hence no computational tasks performed. The same ancillary files are created as would otherwise be created (dynamic, static files, etc.).

**nolog**

Instructs Dynare to not create a log file of this run in FILENAME.log. The default is to create the log file.

**output=first|second|third**

Instructs the preprocessor to output derivatives of the dynamic model at least up to the given order. The *first* option is useful in larger models when debugging steady state computation, because it allows overriding the default computation and output of dynamic second order derivatives in case of the mod-file not containing commands for further computations.

**language=matlab|julia**

Instructs the preprocessor to write output for MATLAB or Julia. Default: MATLAB

**params_derivs_order=0|1|2**

When *identification*, *sensitivity* (with identification), or *estimation* are present, this option is used to limit the order of the derivatives with respect to the parameters that are calculated by the preprocessor. 0 means no derivatives, 1 means first derivatives, and 2 means second derivatives. Default: 2

**nowarn**

Suppresses all warnings.

**notime**

Do not print the total computing time at the end of the driver, and do not save that total computing time to oo_.time.

**transform_unary_ops**

Transform the following operators in the model block into auxiliary variables: exp, log, log10, cos, sin, tan, acos, asin, atan, cosh, sinh, tanh, acosh, asinh, atanh, sqrt, cbrt, abs, sign, erf. Default: no obligatory transformation

**json = parse|check|transform|compute**

Causes the preprocessor to output a version of the .mod file in JSON format to <<M_.fname>>/model/json/. When the JSON output is created depends on the value passed. These values represent various steps of processing in the preprocessor.

If parse is passed, the output will be written after the parsing of the .mod file to a file called FILENAME.json but before file has been checked (e.g. if there are unused exogenous in the model block, the JSON output will be created before the preprocessor exits).

If check is passed, the output will be written to a file called FILENAME.json after the model has been checked.

If `transform` is passed, the JSON output of the transformed model (maximum lead of 1, minimum lag of -1, expectation operators substituted, etc.) will be written to a file called `FILENAME.json` and the original, untransformed model will be written in `FILENAME_original.json`.

And if `compute` is passed, the output is written after the computing pass. In this case, the transformed model is written to `FILENAME.json`, the original model is written to `FILENAME_original.json`, and the dynamic and static files are written to `FILENAME_dynamic.json` and `FILENAME_static.json`.

**`jsonstdout`**

Instead of writing output requested by `json` to files, write to standard out, i.e. to the MATLAB/Octave command window (and the log-file).

**`onlyjson`**

Quit processing once the output requested by `json` has been written.

**`jsonderivsimple`**

Print a simplified version (excluding variable name(s) and lag information) of the static and dynamic files in `FILENAME_static.json` and `FILENAME_dynamic.`.

**`warn_uninit`**

Display a warning for each variable or parameter which is not initialized. See *Parameter initialization*, or `load_params_and_steady_state` for initialization of parameters. See *Initial and terminal conditions*, or `load_params_and_steady_state` for initialization of endogenous and exogenous variables.

**`console`**

Activate console mode. In addition to the behavior of `nodisplay`, Dynare will not use graphical wait bars for long computations.

**`nograph`**

Activate the `nograph` option (see `nograph`), so that Dynare will not produce any graph.

**`nointeractive`**

Instructs Dynare to not request user input.

**`nopathchange`**

By default, Dynare will change MATLAB/Octave's path if `dynare/matlab` directory is not on top and if Dynare's routines are overridden by routines provided in other toolboxes. If one wishes to override Dynare's routines, the `nopathchange` options can be used. Alternatively, the path can be temporarily modified by the user at the top of the `.mod` file (using MATLAB/Octave's `addpath` command).

**`nopreprocessoroutput`**

Prevent Dynare from printing the output of the steps leading up to the preprocessor as well as the preprocessor output itself.

**`mexext=mex|mexw64|mexmaci64|mexmaca64|mexa64`**

The mex extension associated with your platform to be used when compiling output associated with `use_dll`. Dynare is able to set this automatically, so you should not need to set it yourself.

**`matlabroot=<<path>>`**

The path to the MATLAB installation for use with `use_dll`. Dynare is able to set this automatically, so you should not need to set it yourself. See the *note on quotes* for info on passing a <<path>> argument containing spaces.

**`parallel[=CLUSTER_NAME]`**

Tells Dynare to perform computations in parallel. If CLUSTER_NAME is passed, Dynare will use the specified cluster to perform parallel computations. Otherwise, Dynare will use the first cluster specified in the configuration file. See *Parallel Execution and Configuration File*, for more information about the configuration file.

**conffile=FILENAME**

Specifies the location of the configuration file if it differs from the default. See *Parallel Execution and Configuration File*, for more information about the configuration file and its default location. See the *note on quotes* for info on passing a `FILENAME` argument containing spaces.

**parallel_follower_open_mode**

Instructs Dynare to leave the connection to the follower node open after computation is complete, closing this connection only when Dynare finishes processing.

**parallel_test**

Tests the parallel setup specified in the configuration file without executing the `.mod` file. See *Parallel Execution and Configuration File*, for more information about the configuration file.

**parallel_use_psexec=true|false**

For local execution under Windows operating system, set `parallel_use_psexec=false` to use `start` instead of `psexec`, to properly allocate affinity when there are more than 32 cores in the local machine. This option is also helpful if `psexec` cannot be executed due to missing administrator privileges. [default=true]

**-DMACRO_VARIABLE[=MACRO_EXPRESSION]**

Defines a macro-variable from the command line (the same effect as using the Macro directive `@#define` in a model file, see *Macro processing language*). See the *note on quotes* for info on passing a `MACRO_EXPRESSION` argument containing spaces. Note that an expression passed on the command line can reference variables defined before it. If `MACRO_EXPRESSION` is omitted, the variable is assigned the `true` logical value. Strings assigned to a macro variable need to be enclosed in double-quoted strings. This also allows for passing single quotes within the strings.

*Example*

Call Dynare with command line defines

```
>> dynare <<modfile.mod>> -DA=true '-DB="A string with space"' -
↪DC=[1,2,3] '-DD=[ i in C when i > 1 ]' -DE -Ddatafile_name="
↪'my_data_file.mat'"
```

**-I<<path>>**

Defines a path to search for files to be included by the macro processor (using the `@#include` command). Multiple `-I` flags can be passed on the command line. The paths will be searched in the order that the `-I` flags are passed, and the first matching file will be used. The flags passed here take priority over those passed to `@#includepath`. See the *note on quotes* for info on passing a <<path>> argument containing spaces.

**nostrict**

Allows Dynare to issue a warning and continue processing when

1. there are more endogenous variables than equations.
2. an undeclared symbol is assigned in `initval` or `endval`.
3. an undeclared symbol is found in the `model` block in this case, it is automatically declared exogenous.
4. exogenous variables were declared but not used in the `model` block.

**fast**

Don't rewrite the output files otherwise written to the disk by the preprocessor when re-running the same model file while the lists of variables and the equations haven't changed. Note that the whole model still needs to be preprocessed. This option is most useful with model option *use_dll*, because the time-consuming compilation of the MEX files will be skipped. We use a 32 bit checksum, stored in `<model filename>/checksum`. There is a very small probability that the preprocessor misses a change in the model. In case of doubt, re-run without the fast option.

`minimal_workspace`

Instructs Dynare not to write parameter assignments to parameter names in the `.m` file produced by the preprocessor. This is potentially useful when running `dynare` on a large `.mod` file that runs into workspace size limitations imposed by MATLAB.

`compute_xrefs`

Tells Dynare to compute the equation cross-references, writing them to the output `.m` file.

`stochastic`

By default, if no Dynare commands related to stochastic models (`stoch_simul`, `estimation`, `osr`, `discretionary_policy`, `calib_smoother`, `identification`, `method_of_moments`, `sensitivity`) are present in the `.mod` file, Dynare treats the model to be solved as deterministic. This option instructs Dynare that the model to be solved is stochastic and that there is an implicit conditional expectation operator in the model equations. It modifies the internal handling of nonlinear forward-looking terms to account for Jensen's Inequality. This option is useful when the user wants to work with a stochastic model but does not want to use any of the stochastic Dynare commands, e.g., due to calling Matlab functions directly.

`exclude_eqs=<<equation_tags_to_exclude>>`

Tells Dynare to exclude all equations specified by the argument. As a `.mod` file must have the same number of endogenous variables as equations, when *exclude_eqs* is passed, certain rules are followed for excluding endogenous variables. If the `endogenous` tag has been set for the excluded equation, the variable it specifies is excluded. Otherwise, if the left-hand side of the excluded equation is an expression that contains only one endogenous variable, that variable is excluded. If neither of these conditions hold, processing stops with an error. If an endogenous variable has been excluded by the *exclude_eqs* option, and it exists in an equation that has not been excluded, it is transformed into an exogenous variable.

To specify which equations to exclude, you must pass the argument `<<equation_tags_to_exclude>>`. This argument takes either a list of equation tags specifying the equations to be excluded or a filename that contains those tags.

If `<<equation_tags_to_exclude>>` is a list of equation tags, it can take one of the following forms:

1. Given a single argument, e.g. `exclude_eqs=eq1`, the equation with the tag `[name='eq1']` will be excluded. Note that if there is a file called `eq1` in the current directory, Dynare will instead try to open this and read equations to exclude from it (see info on filename argument to `exclude_eqs` below). Further note that if the tag value contains a space, you must use the variant specified in 2 below, i.e. `exclude_eqs=[eq 1]`.
2. Given two or more arguments, e.g. `exclude_eqs=[eq1, eq 2]`, the equations with the tags `[name='eq1']` and `[name='eq 2']` will be excluded.
3. If you'd like to exclude equations based on another tag name (as opposed to the default `name`), you can pass the argument as either e.g. `exclude_eqs=[tagname=a tag]` if a single equation with tag `[tagname='a tag']` is to be excluded or as e.g. `exclude_eqs=[tagname=(a tag, 'a tag with a, comma')]` if more than one equation with tags `[tagname='a tag']` and `[tagname='a tag with a, comma']` will be excluded (note the parenthesis, which are required when more than one equation is specified). Note that if the value of a tag contains a comma, it must be included inside single quotes.

If `<<equation_tags_to_exclude>>` is a filename, the file can take one of the following forms:

1. One equation per line of the file, where every line represents the value passed to the `name` tag. e.g., a file such as:

```
eq1
eq 2
```

would exclude equations with tags `[name='eq1']` and `[name='eq 2']`.

2. One equation per line of the file, where every line after the first line represents the value passed to the tag specified by the first line. e.g., a file such as:

```
tagname=
a tag
a tag with a, comma
```

would exclude equations with tags `[tagname='a tag']` and `[tagname='a tag with a, comma']`. Here note that the first line must end in an equal sign.

**include_eqs=<<equation_tags_to_include>>**

Tells Dynare to run with only those equations specified by the argument; in other words, Dynare will exclude all equations not specified by the argument. The argument <<equation_tags_to_include>> is specified in the same way as the argument to *exclude_eqs*. The functionality of `include_eqs` is to find which equations to exclude then take actions in accord with *exclude_eqs*.

**use_dll**

Instructs the preprocessor to create dynamic loadable libraries (DLL) containing the model equations and derivatives, instead of writing those in M-files. This is equivalent to the *use_dll* option of the `model` block.

**nocommutativity**

This option tells the preprocessor not to use the commutativity of addition and multiplication when looking for common subexpressions. As a consequence, when using this option, equations in various outputs (LaTeX, JSON...) will appear as the user entered them (without terms or factors swapped). Note that using this option may have a performance impact on the preprocessing stage, though it is likely to be small.

These options can be passed to the preprocessor by listing them after the name of the `.mod` file. They can alternatively be defined in the first line of the `.mod` file, this avoids typing them on the command line each time a `.mod` file is to be run. This line must be a Dynare one-line comment (i.e. must begin with `//`) and the options must be whitespace separated between `--+ options:` and `+--`. Note that any text after the `+--` will be discarded. As in the command line, if an option admits a value the equal symbol must not be surrounded by spaces. For instance `json = compute` is not correct, and should be written `json=compute`. The `nopathchange` option cannot be specified in this way, it must be passed on the command-line.

*Output*

Depending on the computing tasks requested in the `.mod` file, executing the `dynare` command will leave variables containing results in the workspace available for further processing. More details are given under the relevant computing tasks. The `M_`, `oo_`, and `options_` structures are saved in a file called `FILENAME_results.mat` located in the `MODFILENAME/Output` folder. If they exist, `estim_params_`, `dataset_`, `oo_recursive_` and `estimation_info` are saved in the same file. Note that MATLAB by default only allows `.mat` files up to 2GB. You can lift this restriction by enabling the `save -v7.3` option in `Preferences -> General -> MAT-Files`.

**MATLAB/Octave variable: M_**

Structure containing various information about the model.

**MATLAB/Octave variable: options_**

Structure contains the values of the various options used by Dynare during the computation.

**MATLAB/Octave variable: oo_**

Structure containing the various results of the computations.

**MATLAB/Octave variable: dataset_**

A `dseries` object containing the data used for estimation.

**MATLAB/Octave variable: oo_recursive_**

Cell array containing the oo_ structures obtained when estimating the model for the different samples when performing recursive estimation and forecasting. The oo_ structure obtained for the sample ranging to the $i$ -th observation is saved in the $i$ -th field. The fields for non-estimated endpoints are empty.

**MATLAB/Octave variable: oo_.time**

Total computing time of the Dynare run, in seconds. This field is not set if the *notime* option has been used.

*Example*

Call Dynare from the MATLAB or Octave prompt, without or with options:

```
>> dynare ramst
>> dynare ramst.mod savemacro
```

Alternatively the options can be passed in the first line of ramst.mod:

```
// --+ options: savemacro, json=compute +--
```

and then call Dynare without passing options on the command line:

```
>> dynare ramst
```

## 3.2 Dynare hooks

It is possible to call pre and post Dynare preprocessor hooks written as MATLAB scripts. The script MODFILENAME/hooks/priorprocessing.m is executed before the call to Dynare's preprocessor, and can be used to programmatically transform the mod file that will be read by the preprocessor. The script MODFILENAME/hooks/postprocessing.m is executed just after the call to Dynare's preprocessor, and can be used to programmatically transform the files generated by Dynare's preprocessor before actual computations start. The pre and/or post Dynare preprocessor hooks are executed if and only if the aforementioned scripts are detected in the same folder as the model file, FILENAME.mod.

## 3.3 Understanding Preprocessor Error Messages

If the preprocessor runs into an error while processing your .mod file, it will issue an error. Due to the way that a parser works, sometimes these errors can be misleading. Here, we aim to demystify these error messages.

The preprocessor issues error messages of the form:

1. ERROR: <<file.mod>>: line A, col B: <<error message>>
2. ERROR: <<file.mod>>: line A, cols B-C: <<error message>>
3. ERROR: <<file.mod>>: line A, col B - line C, col D: <<error message>>

The first two errors occur on a single line, with error two spanning multiple columns. Error three spans multiple rows.

Often, the line and column numbers are precise, leading you directly to the offending syntax. Infrequently however, because of the way the parser works, this is not the case. The most common example of misleading line and column numbers (and error message for that matter) is the case of a missing semicolon, as seen in the following example:

```
varexo a, b
parameters c, ...;
```

In this case, the parser doesn't know a semicolon is missing at the end of the varexo command until it begins parsing the second line and bumps into the parameters command. This is because we allow commands to span multiple lines and, hence, the parser cannot know that the second line will not have a semicolon on it until it gets

there. Once the parser begins parsing the second line, it realizes that it has encountered a keyword, `parameters`, which it did not expect. Hence, it throws an error of the form: `ERROR: <<file.mod>>: line 2, cols 0-9: syntax error, unexpected PARAMETERS`. In this case, you would simply place a semicolon at the end of line one and the parser would continue processing.

It is also helpful to keep in mind that any piece of code that does not violate Dynare syntax, but at the same time is not recognized by the parser, is interpreted as native MATLAB code. This code will be directly passed to the driver script. Investigating the `driver.m` file then helps with debugging. Such problems most often occur when defined variable or parameter names have been misspelled so that Dynare's parser is unable to recognize them.

CHAPTER

FOUR

# THE MODEL FILE

## 4.1 Conventions

A model file contains a list of commands and of blocks. Each command and each element of a block is terminated by a semicolon (;). Blocks are terminated by `end;`.

If Dynare encounters an unknown expression at the beginning of a line or after a semicolon, it will parse the rest of that line as native MATLAB code, even if there are more statements separated by semicolons present. To prevent cryptic error messages, it is strongly recommended to always only put one statement/command into each line and start a new line after each semicolon.[1]

Lines of codes can be commented out line by line or as a block. Single-line comments begin with `//` and stop at the end of the line. Multiline comments are introduced by `/*` and terminated by `*/`.

*Examples*

```
// This is a single line comment
```

```
var x; // This is a comment about x
```

```
/* This is another inline comment about alpha */  alpha = 0.3;
```

```
/*
 This comment is spanning
 two lines.
*/
```

Note that these comment marks should not be used in native MATLAB code regions where the `%` should be preferred instead to introduce a comment. In a `verbatim` block, see *Verbatim inclusion*, this would result in a crash since `//` is not a valid MATLAB statement.

Most Dynare commands have arguments and several accept options, indicated in parentheses after the command keyword. Several options are separated by commas.

In the description of Dynare commands, the following conventions are observed:

- Optional arguments or options are indicated between square brackets: '[]';
- Repeated arguments are indicated by ellipses: "…";
- Mutually exclusive arguments are separated by vertical bars: '|';
- INTEGER indicates an integer number;
- INTEGER_VECTOR indicates a vector of integer numbers separated by spaces, enclosed by square brackets;

[1] A `.mod` file must have lines that end with a line feed character, which is not commonly visible in text editors. Files created on Windows and Unix-based systems have always conformed to this requirement, as have files created on OS X and macOS. Files created on old, pre-OS X Macs used carriage returns as end of line characters. If you get a Dynare parsing error of the form `ERROR: <<mod file>>: line 1, cols 341-347: syntax error,...` and there's more than one line in your `.mod` file, know that it uses the carriage return as an end of line character. To get more helpful error messages, the carriage returns should be changed to line feeds.

- DOUBLE indicates a double precision number. The following syntaxes are valid: `1.1e3`, `1.1E3`, `1.1d3`, `1.1D3`. In some places, infinite Values `Inf` and `-Inf` are also allowed;
- NUMERICAL_VECTOR indicates a vector of numbers separated by spaces, enclosed by square brackets;
- EXPRESSION indicates a mathematical expression valid outside the model description (see *Expressions*);
- MODEL_EXPRESSION (sometimes MODEL_EXP) indicates a mathematical expression valid in the model description (see *Expressions* and *Model declaration*);
- MACRO_EXPRESSION designates an expression of the macro processor (see *Macro expressions*);
- VARIABLE_NAME (sometimes VAR_NAME) indicates a variable name starting with an alphabetical character and can't contain: `()+-\*/^=!;:@#.` or accentuated characters;
- ENDOGENOUS_NAME indicates an endogenous variable name
- EXOGENOUS_NAME indicates an exogenous variable name
- PARAMETER_NAME (sometimes PARAM_NAME) indicates a parameter name starting with an alphabetical character and can't contain: `()+-\*/^=!;:@#.` or accentuated characters;
- DATABASE_NAME indicates the name of a variable in the MATLAB/Octave workspace that can be used as a source of numeric values for model variables;
- LATEX_NAME (sometimes TEX_NAME) indicates a valid LaTeX expression in math mode (not including the dollar signs);
- FUNCTION_NAME indicates a valid MATLAB function name;
- FILENAME indicates a filename valid in the underlying operating system; it is necessary to put it between quotes when specifying the extension or if the filename contains a non-alphanumeric character;
- QUOTED_STRING indicates an arbitrary string enclosed between (single) quotes;
- DATE indicates a time period which can be either a year (e.g. `2024Y` or `2024A`), a half-year (`2024S1` or `2024H1`), a quarter (`2024Q2`) or a month (`2024M3`) (see *Dates in a mod file*). Optionally, the time period can be followed by a plus sign and a number of periods, in which case the date is shifted accordingly (e.g. `2023Q1+6` is accepted and is equivalent to `2024Q3`).

## 4.2 Variable declarations

While Dynare allows the user to choose their own variable names, there are some restrictions to be kept in mind. First, variables and parameters must not have the same name as Dynare commands or built-in functions. In this respect, Dynare is not case-sensitive. For example, do not use `Ln` or `shocks` to name your variable. Not conforming to this rule might yield hard-to-debug error messages or crashes. Second, when employing user-defined steady-state files it is recommended to avoid using the name of MATLAB functions as this may cause conflicts. In particular, when working with user-defined steady-state files, do not use correctly-spelled Greek names like `alpha`, because there are MATLAB functions of the same name. Rather go for `alppha` or `alph`. Lastly, please do not name a variable or parameter `i`. This may interfere with the imaginary number i and the index in many loops. Rather, name investment `invest`. Using `inv` is also not recommended as it already denotes the inverse operator. Commands for declaring variables and parameters are described below.

**Command:** `var VAR_NAME [$TEX_NAME$] [(long_name=QUOTED_STRING|NAME=QUOTED_STRING)]...;`

**Command:**
`var(log) VAR_NAME [$TEX_NAME$] [(long_name=QUOTED_STRING|NAME=QUOTED_STRING)]...;`

**Command:** `var(deflator=MODEL_EXPR) VAR_NAME (... same options apply)`

**Command:** `var(log, deflator=MODEL_EXPR) VAR_NAME (... same options apply)`

**Command:** `var(log_deflator=MODEL_EXPR) VAR_NAME (... same options apply)`

This required command declares the endogenous variables in the model. See *Conventions* for the syntax of *VAR_NAME* and *MODEL_EXPR*. Optionally it is possible to give a LaTeX name to the variable or, if it is nonstationary, provide information regarding its deflator. The variables in the list can be separated by spaces or by commas. `var` commands can appear several times in the file and Dynare will concatenate them. Dynare stores the list of declared parameters, in the order of declaration, in a column cell array `M_.endo_names`.

If the model is nonstationary and is to be written as such in the `model` block, Dynare will need the trend deflator for the appropriate endogenous variables in order to stationarize the model. The trend deflator must be provided alongside the variables that follow this trend.

*Options*

**`log`**

In addition to the endogenous variable(s) thus declared, this option also triggers the creation of auxiliary variable(s) equal to the log of the corresponding endogenous variable(s). For example, given a `var(log) y` statement, two endogenous will be created (`y` and `LOG_y`), and an auxiliary equation linking the two will also be added (equal to `LOG_y = log(y)`). Moreover, every occurrence of `y` in the model will be replaced by `exp(LOG_y)`. This option is for example useful when one wants to perform a loglinear approximation of some variable(s) in the context of a first-order stochastic approximation; or when one wants to ensure the variable(s) stay(s) in the definition domain of the function defining the steady state or the dynamic residuals when the nonlinear solver is used.

**`deflator = MODEL_EXPR`**

The expression used to detrend an endogenous variable. All trend variables, endogenous variables and parameters referenced in MODEL_EXPR must already have been declared by the `trend_var`, `log_trend_var`, `var` and `parameters` commands. The deflator is assumed to be multiplicative; for an additive deflator, use `log_deflator`. This option can be used together with the `log` option (the latter must come first).

**`log_deflator = MODEL_EXPR`**

Same as `deflator`, except that the deflator is assumed to be additive instead of multiplicative (or, to put it otherwise, the declared variable is equal to the log of a variable with a multiplicative trend). This option cannot be used together with the `log` option, because it would not make much sense from an economic point of view (the corresponding auxiliary variable would correspond to the log taken two times on a variable with a multiplicative trend).

**`long_name = QUOTED_STRING`**

This is the long version of the variable name. Its value is stored in `M_.endo_names_long` (a column cell array, in the same order as `M_.endo_names`). In case multiple `long_name` options are provided, the last one will be used. Default: `VAR_NAME`.

**`NAME = QUOTED_STRING`**

This is used to create a partitioning of variables. It results in the direct output in the `.m` file analogous to: `M_.endo_partitions.NAME = QUOTED_STRING;`.

*Example (variable partitioning)*

```
var c gnp cva (country='US', state='VA')
          cca (country='US', state='CA', long_name='Consumption CA');
var(deflator=A) i b;
var c $C$ (long_name='Consumption');
```

**Command:**

**`varexo`** `VAR_NAME [$TEX_NAME$] [(long_name=QUOTED_STRING|NAME=QUOTED_STRING)...];`

This optional command declares the exogenous variables in the model. See *Conventions* for the syntax of `VAR_NAME`. Optionally it is possible to give a LaTeX name to the variable. Exogenous variables are required if the user wants to be able to apply shocks to her model. The variables in the list can be separated by spaces or by commas. `varexo` commands can appear several times in the file and Dynare will concatenate them.

*Options*

**`long_name = QUOTED_STRING`**

Like *long_name* but value stored in `M_.exo_names_long`.

**`NAME = QUOTED_STRING`**

Like *partitioning* but QUOTED_STRING stored in `M_.exo_partitions.NAME`.

*Example*

```
varexo m gov;
```

*Remarks*

An exogenous variable is an innovation, in the sense that this variable cannot be predicted from the knowledge of the current state of the economy. For instance, if logged TFP is a first order autoregressive process:

$$a_t = \rho a_{t-1} + \varepsilon_t$$

then logged TFP $a_t$ is an endogenous variable to be declared with `var`, its best prediction is $\rho a_{t-1}$, while the innovation $\varepsilon_t$ is to be declared with `varexo`.

**Command:**
**varexo_det** VAR_NAME [$TEX_NAME$] [(long_name=QUOTED_STRING|NAME=QUOTED_STRING)...];

This optional command declares exogenous deterministic variables in a stochastic model. See *Conventions* for the syntax of VARIABLE_NAME. Optionally it is possible to give a LaTeX name to the variable. The variables in the list can be separated by spaces or by commas. `varexo_det` commands can appear several times in the file and Dynare will concatenate them.

It is possible to mix deterministic and stochastic shocks to build models where agents know from the start of the simulation about future exogenous changes. In that case *stoch_simul* will compute the rational expectation solution adding future information to the state space (nothing is shown in the output of *stoch_simul*) and forecast will compute a simulation conditional on initial conditions and future information.

Note that exogenous deterministic variables cannot appear with a lead or a lag in the model.

*Options*

**long_name = QUOTED_STRING**

Like *long_name* but value stored in `M_.exo_det_names_long`.

**NAME = QUOTED_STRING**

Like *partitioning* but QUOTED_STRING stored in `M_.exo_det_partitions.NAME`.

*Example*

```
varexo m gov;
varexo_det tau;
```

**Command:**
**parameters** PARAM_NAME [$TEX_NAME$] [(long_name=QUOTED_STRING|NAME=QUOTED_STRING)...];

This command declares parameters used in the model, in variable initialization or in shocks declarations. See *Conventions* for the syntax of `PARAM_NAME`. Optionally it is possible to give a LaTeX name to the parameter.

The parameters must subsequently be assigned values (see *Parameter initialization*).

The parameters in the list can be separated by spaces or by commas. `parameters` commands can appear several times in the file and Dynare will concatenate them.

*Options*

**long_name = QUOTED_STRING**

Like *long_name* but value stored in `M_.param_names_long`.

**NAME = QUOTED_STRING**

Like *partitioning* but QUOTED_STRING stored in `M_.param_partitions.NAME`.

*Example*

```
parameters alpha, bet;
```

**Command:** **change_type**(var|varexo|varexo_det|parameters) VAR_NAME | PARAM_NAME...;

Changes the types of the specified variables/parameters to another type: endogenous, exogenous, exogenous deterministic or parameter. It is important to understand that this command has a global effect on the `.mod` file: the type change is effective after, but also before, the `change_type` command. This command is typically used when flipping some variables for steady-state calibration: typically a separate model file is used for calibration, which includes the list of variable declarations with the macro processor, and flips some variable.

*Example*

```
var y, w;
parameters alpha, beta;
...
change_type(var) alpha, beta;
change_type(parameters) y, w;
```

Here, in the whole model file, `alpha` and `beta` will be endogenous and `y` and `w` will be parameters.

**Command:** **var_remove** VAR_NAME | PARAM_NAME...;

Removes the listed variables (or parameters) from the model. Removing a variable that has already been used in a model equation or elsewhere will lead to an error.

**Command:** **predetermined_variables** VAR_NAME...;

In Dynare, the default convention is that the timing of a variable reflects when this variable is decided. The typical example is for capital stock: since the capital stock used at current period is actually decided at the previous period, then the capital stock entering the production function is `k(-1)`, and the law of motion of capital must be written:

```
k = i + (1-delta)*k(-1)
```

Put another way, for stock variables, the default in Dynare is to use a "stock at the end of the period" concept, instead of a "stock at the beginning of the period" convention.

The `predetermined_variables` is used to change that convention. The endogenous variables declared as predetermined variables are supposed to be decided one period ahead of all other endogenous variables. For stock variables, they are supposed to follow a "stock at the beginning of the period" convention.

Note that Dynare internally always uses the "stock at the end of the period" concept, even when the model has been entered using the `predetermined_variables` command. Thus, when plotting, computing or simulating variables, Dynare will follow the convention to use variables that are decided in the current period. For example, when generating impulse response functions for capital, Dynare will plot `k`, which is the capital stock decided upon by investment today (and which will be used in tomorrow's production function). This is the reason that capital is shown to be moving on impact, because it is `k` and not the predetermined `k(-1)` that is displayed. It is important to remember that this also affects simulated time series and output from smoother routines for predetermined variables. Compared to non-predetermined variables they might otherwise appear to be falsely shifted to the future by one period.

*Example*

The following two program snippets are strictly equivalent.

Using default Dynare timing convention:

```
var y, k, i;
...
model;
y = k(-1)^alpha;
k = i + (1-delta)*k(-1);
...
end;
```

Using the alternative timing convention:

```
var y, k, i;
predetermined_variables k;
...
model;
y = k^alpha;
k(+1) = i + (1-delta)*k;
...
end;
```

**Command:** `trend_var(growth_factor = MODEL_EXPR) VAR_NAME [$LATEX_NAME$]...;`

This optional command declares the trend variables in the model. See *Conventions* for the syntax of MODEL_EXPR and VAR_NAME. Optionally it is possible to give a LaTeX name to the variable.

The variable is assumed to have a multiplicative growth trend. For an additive growth trend, use `log_trend_var` instead.

Trend variables are required if the user wants to be able to write a nonstationary model in the `model` block. The `trend_var` command must appear before the var command that references the trend variable.

`trend_var` commands can appear several times in the file and Dynare will concatenate them.

If the model is nonstationary and is to be written as such in the `model` block, Dynare will need the growth factor of every trend variable in order to stationarize the model. The growth factor must be provided within the declaration of the trend variable, using the `growth_factor` keyword. All endogenous variables and parameters referenced in MODEL_EXPR must already have been declared by the var and parameters commands.

*Example*

```
trend_var (growth_factor=gA) A;
```

**Command:** `log_trend_var(log_growth_factor = MODEL_EXPR) VAR_NAME [$LATEX_NAME$]...;`

Same as `trend_var`, except that the variable is supposed to have an additive trend (or, to put it otherwise, to be equal to the log of a variable with a multiplicative trend).

**Command:** `model_local_variable VARIABLE_NAME [LATEX_NAME]... ;`

This optional command declares a model local variable. See *Conventions* for the syntax of VARIABLE_NAME. As you can create model local variables on the fly in the model block (see *Model declaration*), the interest of this command is primarily to assign a LATEX_NAME to the model local variable.

*Example*

```
model_local_variable GDP_US $GDPUS$;
```

### 4.2.1 On-the-fly Model Variable Declaration

Endogenous variables, exogenous variables, and parameters can also be declared inside the model block. You can do this in two different ways: either via the equation tag (only for endogenous variables) or directly in an equation (for endogenous, exogenous or parameters).

To declare an endogenous variable on-the-fly in an equation tag, simply write `endogenous` followed by an equal sign and the variable name in single quotes. Hence, to declare a variable `c` as endogenous in an equation tag, you can type `[endogenous='c']`.

To perform on-the-fly variable declaration in an equation, simply follow the symbol name with a vertical line (`|`, pipe character) and either an `e` (for endogenous), an `x` (for exogenous), or a `p` (for parameter). For example, to declare a parameter named `alphaa` in the model block, you could write `alphaa|p` directly in an equation where it appears. Similarly, to declare an endogenous variable `c` in the model block you could write `c|e`. Note that in-equation on-the-fly variable declarations must be made on contemporaneous variables.

On-the-fly variable declarations do not have to appear in the first place where this variable is encountered.

*Example*

The following two snippets are equivalent:

```
model;
  [endogenous='k',name='law of motion of capital']
  k(+1) = i|e + (1-delta|p)*k;
  y|e = k^alpha|p;
  ...
end;
delta = 0.025;
alpha = 0.36;
```

```
var k, i, y;
parameters delta, alpha;
delta = 0.025;
alpha = 0.36;
...
model;
  [name='law of motion of capital']
  k(1) = i|e + (1-delta|p)*k;
  y|e = k|e^alpha|p;
  ...
end;
```

## 4.3 Expressions

Dynare distinguishes between two types of mathematical expressions: those that are used to describe the model, and those that are used outside the model block (e.g. for initializing parameters or variables, or as command options). In this manual, those two types of expressions are respectively denoted by MODEL_EXPRESSION and EXPRESSION.

Unlike MATLAB or Octave expressions, Dynare expressions are necessarily scalar ones: they cannot contain matrices or evaluate to matrices.[2]

Expressions can be constructed using integers (INTEGER), floating point numbers (DOUBLE), parameter names (PARAMETER_NAME), variable names (VARIABLE_NAME), operators and functions.

The following special constants are also accepted in some contexts:

**Constant: inf**

Represents infinity.

**Constant: nan**

"Not a number": represents an undefined or unrepresentable value.

### 4.3.1 Parameters and variables

Parameters and variables can be introduced in expressions by simply typing their names. The semantics of parameters and variables is quite different whether they are used inside or outside the model block.

#### 4.3.1.1 Inside the model

Parameters used inside the model refer to the value given through parameter initialization (see *Parameter initialization*) or `homotopy_setup` when doing a simulation, or are the estimated variables when doing an estimation.

Variables used in a MODEL_EXPRESSION denote current period values when neither a lead nor a lag is given. A lead or a lag can be given by enclosing an integer between parenthesis just after the variable name: a positive integer means a lead, a negative one means a lag. Leads or lags of more than one period are allowed. For example,

[2] Note that arbitrary MATLAB or Octave expressions can be put in a `.mod` file, but those expressions have to be on separate lines, generally at the end of the file for post-processing purposes. They are not interpreted by Dynare, and are simply passed on unmodified to MATLAB or Octave. Those constructions are not addresses in this section.

if `c` is an endogenous variable, then `c(+1)` is the variable one period ahead, and `c(-2)` is the variable two periods before.

When specifying the leads and lags of endogenous variables, it is important to respect the following convention: in Dynare, the timing of a variable reflects when that variable is decided. A control variable — which by definition is decided in the current period — must have no lead. A predetermined variable — which by definition has been decided in a previous period — must have a lag. A consequence of this is that all stock variables must use the "stock at the end of the period" convention.

Leads and lags are primarily used for endogenous variables, but can be used for exogenous variables. They have no effect on parameters and are forbidden for local model variables (see Model declaration).

#### 4.3.1.2 Outside the model

When used in an expression outside the model block, a parameter or a variable simply refers to the last value given to that variable. More precisely, for a parameter it refers to the value given in the corresponding parameter initialization (see *Parameter initialization*); for an endogenous or exogenous variable, it refers to the value given in the most recent `initval` or `endval` block.

### 4.3.2 Operators

The following operators are allowed in both MODEL_EXPRESSION and EXPRESSION:

- Binary arithmetic operators: `+`, `-`, `*`, `/`, `^`
- Unary arithmetic operators: `+`, `-`
- Binary comparison operators (which evaluate to either 0 or 1): `<`, `>`, `<=`, `>=`, `==`, `!=`

Note the binary comparison operators are differentiable everywhere except on a line of the 2-dimensional real plane. However for facilitating convergence of Newton-type methods, Dynare assumes that, at the points of non-differentiability, the partial derivatives of these operators with respect to both arguments is equal to 0 (since this is the value of the partial derivatives everywhere else).

The following special operators are accepted in MODEL_EXPRESSION (but not in EXPRESSION):

**`Operator: STEADY_STATE (MODEL_EXPRESSION)`**

This operator is used to take the value of the enclosed expression at the steady state. A typical usage is in the Taylor rule, where you may want to use the value of GDP at steady state to compute the output gap.

Exogenous and exogenous deterministic variables may not appear in MODEL_EXPRESSION.

> **Warning**
>
> The concept of a steady state is ambiguous in a perfect foresight context with permanent and potentially anticipated shocks occurring. Dynare will use the contents of `oo_.steady_state` as its reference for calls to the `STEADY_STATE()` operator. In the presence of `endval`, this implies that the terminal state provided by the user is used. This may be a steady state computed by Dynare (if `endval` is followed by `steady`) or simply the terminal state provided by the user (if `endval` is not followed by `steady`). Put differently, Dynare will not automatically compute the steady state conditional on the specified value of the exogenous variables in the respective periods.

**`Operator: EXPECTATION (INTEGER) (MODEL_EXPRESSION)`**

This operator is used to take the expectation of some expression using a different information set than the information available at current period. For example, `EXPECTATION(-1)(x(+1))` is equal to the expected value of variable x at next period, using the information set available at the previous period. See *Auxiliary variables* for an explanation of how this operator is handled internally and how this affects the output.

### 4.3.3 Functions

#### 4.3.3.1 Built-in functions

The following standard functions are supported internally for both MODEL_EXPRESSION and EXPRESSION:

**Function:** **exp**(x)

Natural exponential.

**Function:** **log**(x)

**Function:** **ln**(x)

Natural logarithm.

**Function:** **log10**(x)

Base 10 logarithm.

**Function:** **sqrt**(x)

Square root.

**Function:** **cbrt**(x)

Cube root.

**Function:** **sign**(x)

Signum function, defined as:

$$\operatorname{sign}(x) = \begin{cases} -1 & \text{if } x < 0 \\ 0 & \text{if } x = 0 \\ 1 & \text{if } x > 0 \end{cases}$$

Note that this function is not continuous, hence not differentiable, at $x = 0$. However, for facilitating convergence of Newton-type methods, Dynare assumes that the derivative at $x = 0$ is equal to 0. This assumption comes from the observation that both the right- and left-derivatives at this point exist and are equal to 0, so we can remove the singularity by postulating that the derivative at $x = 0$ is 0.

**Function:** **abs**(x)

Absolute value.

Note that this continuous function is not differentiable at $x = 0$. However, for facilitating convergence of Newton-type methods, Dynare assumes that the derivative at $x = 0$ is equal to 0 (even if the derivative does not exist). The rationale for this mathematically unfounded definition, rely on the observation that the derivative of $\operatorname{abs}(x)$ is equal to $\operatorname{sign}(x)$ for any $x \neq 0$ in $\mathbb{R}$ and from the convention for the value of $\operatorname{sign}(x)$ at $x = 0$).

**Function:** **sin**(x)

**Function:** **cos**(x)

**Function:** **tan**(x)

**Function:** **asin**(x)

**Function:** **acos**(x)

**Function:** **atan**(x)

Trigonometric functions.

**Function:** **sinh**(x)

**Function:** **cosh**(x)

**Function:** **tanh**(x)

**Function:** **asinh**(x)

**Function:** **acosh**(x)

**Function:** **atanh**(x)

Hyperbolic functions.

**Function:** **max**(a, b)

**Function:** **min**(a, b)

Maximum and minimum of two reals.

Note that these functions are differentiable everywhere except on a line of the 2-dimensional real plane defined by $a = b$. However for facilitating convergence of Newton-type methods, Dynare assumes that, at the points of non-differentiability, the partial derivative of these functions with respect to the first (resp. the second) argument is equal to 1 (resp. to 0) (i.e. the derivatives at the kink are equal to the derivatives observed on the half-plane where the function is equal to its first argument).

**Function:** **normcdf**(x)

**Function:** **normcdf**(x, mu, sigma)

Gaussian cumulative density function, with mean *mu* and standard deviation *sigma*. Note that `normcdf(x)` is equivalent to `normcdf(x,0,1)`.

**Function:** **normpdf**(x)

**Function:** **normpdf**(x, mu, sigma)

Gaussian probability density function, with mean *mu* and standard deviation *sigma*. Note that `normpdf(x)` is equivalent to `normpdf(x,0,1)`.

**Function:** **erf**(x)

Gauss error function.

**Function:** **erfc**(x)

Complementary error function, *i.e.* $\mathrm{erfc}(x) = 1 - \mathrm{erf}(x)$.

#### 4.3.3.2 External functions

Any other user-defined (or built-in) MATLAB or Octave function may be used in both a MODEL_EXPRESSION and an EXPRESSION, provided that this function has a scalar argument as a return value.

To use an external function in a MODEL_EXPRESSION, one must declare the function using the `external_function` statement. This is not required for external functions used in an EXPRESSION outside a `model` block or *`steady_state_model`* block.

**Command:** **external_function**(OPTIONS...);

This command declares the external functions used in the model block. It is required for every unique function used in the model block.

`external_function` commands can appear several times in the file and must come before the model block.

*Options*

**name = NAME**

The name of the function, which must also be the name of the M-/MEX file implementing it. This option is mandatory.

**nargs = INTEGER**

The number of arguments of the function. If this option is not provided, Dynare assumes `nargs = 1`.

**first_deriv_provided [= NAME]**

If NAME is provided, this tells Dynare that the Jacobian is provided as the only output of the M-/MEX file given as the option argument. If NAME is not provided, this tells Dynare that the M-/MEX file specified by the argument passed to NAME returns the Jacobian as its second output argument. When this option is not provided, Dynare will use finite difference approximations for computing the derivatives of the function, whenever needed.

**second_deriv_provided [= NAME]**

If NAME is provided, this tells Dynare that the Hessian is provided as the only output of the M-/MEX file given as the option argument. If NAME is not provided, this tells Dynare that the M-/MEX file specified by the argument passed to NAME returns the Hessian as its third output argument. NB: This option can only be used if the `first_deriv_provided` option is used in the same `external_function` command. When this option is not provided, Dynare will use finite difference approximations for computing the Hessian derivatives of the function, whenever needed.

*Example*

```
external_function(name = funcname);
external_function(name = otherfuncname, nargs = 2, first_deriv_provided,␣
↪second_deriv_provided);
external_function(name = yetotherfuncname, nargs = 3, first_deriv_
↪provided = funcname_deriv);
```

### 4.3.4 A few words of warning in stochastic context

The use of the following functions and operators is strongly discouraged in a stochastic context: `max`, `min`, `abs`, `sign`, <, >, <=, >=, ==, !=.

The reason is that the local approximation used by *stoch_simul* or *estimation* will by nature ignore the non-linearities introduced by these functions if the steady state is away from the kink. And, if the steady state is exactly at the kink, then the approximation will be bogus because the derivative of these functions at the kink is bogus (as explained in the respective documentations of these functions and operators).

Note that *extended_path* is not affected by this problem, because it does not rely on a local approximation of the mode.

## 4.4 Parameter initialization

When using Dynare for computing simulations, it is necessary to calibrate the parameters of the model. This is done through parameter initialization.

The syntax is the following:

```
PARAMETER_NAME = EXPRESSION;
```

Here is an example of calibration:

```
parameters alpha, beta;

beta = 0.99;
alpha = 0.36;
A = 1-alpha*beta;
```

Internally, the parameter values are stored in `M_.params`:

**MATLAB/Octave variable: M_.params**

Contains the values of model parameters. The parameters are in the order that was used in the `parameters` command, hence ordered as in `M_.param_names`.

The parameter names are stored in `M_.param_names`:

**MATLAB/Octave variable:** `M_.param_names`

Cell array containing the names of the model parameters.

**MATLAB/Octave command:** `get_param_by_name('PARAMETER_NAME');`

Given the name of a parameter, returns its calibrated value as it is stored in `M_.params`.

**MATLAB/Octave command:** `set_param_value('PARAMETER_NAME', MATLAB_EXPRESSION);`

Sets the calibrated value of a parameter to the provided expression. This does essentially the same as the parameter initialization syntax described above, except that it accepts arbitrary MATLAB/Octave expressions, and that it works from MATLAB/Octave scripts.

## 4.5 Model declaration

The model is declared inside a `model` block:

**Block:** `model ;`

**Block:** `model(OPTIONS...);`

The equations of the model are written in a block delimited by `model` and `end` keywords.

There must be as many equations as there are endogenous variables in the model, except when computing the unconstrained optimal policy with *ramsey_model*, *ramsey_policy* or *discretionary_policy*.

The syntax of equations must follow the conventions for MODEL_EXPRESSION as described in *Expressions*. Each equation must be terminated by a semicolon (';'). A normal equation looks like:

MODEL_EXPRESSION = MODEL_EXPRESSION;

When the equations are written in homogenous form, it is possible to omit the '=0' part and write only the left-hand side of the equation. A homogenous equation looks like:

MODEL_EXPRESSION;

**Warning**

In Dynare, only equality signs can delineate the left and right-hand side of an equation. If Dynare encounters an expression like `a>=b`, this will therefore not define an inequality constraint. Rather, it is interpreted as the homogenous equation `(a>=b)=0;`, i.e., the Boolean `(a>=b)` must evaluate to 0. Inequality constraints in Dynare instead need to be set up either via OccBin or as mixed complementarity problems.

Inside the model block, Dynare allows the creation of *model-local variables*, which constitute a simple way to share a common expression between several equations. The syntax consists of a pound sign (#) followed by the name of the new model local variable (which must **not** be declared as in *Variable declarations*, but may have been declared by *model_local_variable*), an equal sign, and the expression for which this new variable will stand. Later on, every time this variable appears in the model, Dynare will substitute it by the expression assigned to the variable (if the model-local variable appears with a lead or a lag attached to it between parenthesis, the substitution will be done by shifting the expression accordingly). Note that the scope of this variable is restricted to the model block; it cannot be used outside. To assign a LaTeX name to the model local variable, use the declaration syntax outlined by *model_local_variable*. A model local variable declaration looks like:

#VARIABLE_NAME = MODEL_EXPRESSION;

It is possible to tag equations written in the model block. A tag can serve different purposes by allowing the user to attach arbitrary information to each equation and to recover them at runtime. For instance, it is possible to name the equations with a `name` tag, using a syntax like:

```
model;

[name = 'Budget constraint'];
c + k = k^theta*A;

end;
```

Here, `name` is the keyword indicating that the tag names the equation. If an equation of the model is tagged with a name, the *`resid`* command will display the name of the equations (which may be more informative than the equation numbers) in addition to the equation number. Several tags for one equation can be separated using a comma:

```
model;

[name='Taylor rule', endogenous='r']
r = rho*r(-1) + (1-rho)*(gpi*Infl+gy*YGap) + e;

end;
```

More information on tags is available at https://git.dynare.org/Dynare/dynare/-/wikis/Equations-Tags.

There can be several `model` blocks, in which case they are simply concatenated. The set of effective options is also the concatenation of the options declared in all the blocks, but in that case you may rather want to use the *`model_options`* command.

*Options*

**`linear`**

Declares the model as being linear. It spares oneself from having to declare initial values for computing the steady state of a stationary linear model. This option can't be used with non-linear models, it will NOT trigger linearization of the model.

**`use_dll`**

Instructs the preprocessor to create dynamic loadable libraries (DLL) containing the model equations and derivatives, instead of writing those in M-files. You need a working compilation environment, (see *Compiler installation* for more details). Using this option can result in faster simulations or estimations, at the expense of some initial compilation time. Alternatively, this option can be given to the `dynare` command (see *Dynare invocation*).[3]

**`block`**

Perform the block decomposition of the model, and exploit it in computations (steady state, deterministic simulation, stochastic simulation with first-order approximation and estimation). See https://git.dynare.org/Dynare/dynare/-/wikis/BlockDecomposition for details on the algorithms used in deterministic simulation and steady-state computation.

**`bytecode`**

Instead of M-files, use a bytecode representation of the model, i.e. a binary file containing a compact representation of all the equations.

**`cutoff = DOUBLE`**

Threshold under which a Jacobian element is considered as null during the model normalization. Only available with option `block`. Default: `1e-15`

**`mfs = INTEGER`**

Controls the handling of minimum feedback set of endogenous variables for the dynamic model. Only available with option `block`. Possible values:

`0`

[3] In particular, for big models, the compilation step can be very time-consuming, and use of this option may be counter-productive in those cases.

All the endogenous variables are considered as feedback variables.

1
The endogenous variables assigned to equation naturally normalized (i.e. of the form $x = f(Y)$ where $x$ does not appear in $Y$) are potentially recursive variables. All the other variables are forced to belong to the set of feedback variables.

2
In addition to variables with `mfs = 1`. the endogenous variables related to linear equations which could be normalized are potential recursive variables. All the other variables are forced to belong to the set of feedback variables.

3
In addition to variables with `mfs = 2`, the endogenous variables related to non-linear equations which could be normalized are potential recursive variables. All the other variables are forced to belong to the set of feedback variables.

Default value is `1`.

**static_mfs**

Controls the handling of minimum feedback set of endogenous variables for the static model. Only available with option `block`. See the `mfs` option for the possible values. Default value is `0`.

**no_static**

Don't create the static model file. This can be useful for models which don't have a steady state.

**differentiate_forward_vars**

**differentiate_forward_vars = ( VARIABLE_NAME [VARIABLE_NAME ...] )**

Tells Dynare to create a new auxiliary variable for each endogenous variable that appears with a lead, such that the new variable is the time differentiate of the original one. More precisely, if the model contains `x(+1)`, then a variable `AUX_DIFF_FWRD_nn` (where `nn` is some number) will be created such that `AUX_DIFF_FWRD_nn=x-x(-1)`, and `x(+1)` will be replaced with `x+AUX_DIFF_FWRD_nn(+1)`.

The transformation is applied to all endogenous variables with a lead if the option is given without a list of variables. If there is a list, the transformation is restricted to endogenous with a lead that also appear in the list.

This option can be useful for some deterministic simulations where convergence is hard to obtain. Bad values for terminal conditions in the case of very persistent dynamics or permanent shocks can hinder correct solutions or any convergence. The new differentiated variables have obvious zero terminal conditions (if the terminal condition is a steady state) and this in many cases helps convergence of simulations.

**parallel_local_files = ( FILENAME [, FILENAME]... )**

Declares a list of extra files that should be transferred to follower nodes when doing a parallel computation (see *Cluster-based Parallel Configuration*).

**balanced_growth_test_tol = DOUBLE**

Tolerance used for determining whether cross-derivatives are zero in the test for balanced growth path (the latter is documented on https://git.dynare.org/Dynare/dynare/-/wikis/RemovingTrends). Default: `1e-6`

*Example* (Elementary RBC model)

```
var c k;
varexo x;
parameters aa alph bet delt gam;

model;
c =  - k + aa*x*k(-1)^alph + (1-delt)*k(-1);
```

(continues on next page)

(continued from previous page)

```
c^(-gam) = (aa*alph*x(+1)*k^(alph-1) + 1 - delt)*c(+1)^(-gam)/
↪(1+bet);
end;
```

*Example* (Use of model local variables)

The following program:

```
model;
# gamma = 1 - 1/sigma;
u1 = c1^gamma/gamma;
u2 = c2^gamma/gamma;
end;
```

...is formally equivalent to:

```
model;
u1 = c1^(1-1/sigma)/(1-1/sigma);
u2 = c2^(1-1/sigma)/(1-1/sigma);
end;
```

*Example* (A linear model)

```
model(linear);
x = a*x(-1)+b*y(+1)+e_x;
y = d*y(-1)+e_y;
end;
```

**Command:** **model_options**(OPTIONS...);

This command accepts the same options as the *model* block.

The purpose of this statement is to specify the options that apply to the whole model, when there are several `model` blocks, so as to restore the symmetry between those blocks (since otherwise one `model` block would typically bear the options, while the other ones would typically have no option).

**Command:** **model_remove**(TAGS...);

This command removes equations that appeared in a previous *model* block.

The equations must be specified by a list of tag values, separated by commas. Each element of the list is either a simple quoted string, in which case it designates an equation by its `name` tag; or a tag name (without quotes), followed by an equal sign, then by the tag value (within quotes); or a list of tag-equals-value pairs separated by commas and enclosed within brackets, in which case this element removes the equation(s) that has all these tags with the corresponding values.

Each removed equation must either have an `endogenous` tag, or have a left-hand side containing a single endogenous variable. The corresponding endogenous variable will be either turned into an exogenous (if it is still used in somewhere in the model at that point), otherwise it will be removed from the model.

*Example*

```
var c k dummy1 dummy2 dummy3;

model;
  c + k - aa*x*k(-1)^alph - (1-delt)*k(-1) + dummy1;
  c^(-gam) - (1+bet)^(-1)*(aa*alph*x(+1)*k^(alph-1) + 1 - delt)*c(+1)^(-
↪gam);
  [ name = 'eq:dummy1', endogenous = 'dummy1' ]
  c*k = dummy1;
  [ foo = 'eq:dummy2' ]
```

(continues on next page)

(continued from previous page)

```
  log(dummy2) = k + 2;
  [ name = 'eq:dummy3', bar = 'baz' ]
  dummy3 = c + 3;
end;

model_remove('eq:dummy1', foo = 'eq:dummy2', [ name = 'eq:dummy3', bar =
↪'baz' ]);
```

In the above example, the last three equations will be removed, `dummy1` will be turned into an exogenous, and `dummy2` and `dummy3` will be removed.

**Block:** **model_replace**(TAGS...);

This block replaces several equations in the model. It removes the equations given by the tags list (with the same syntax as in *model_remove*), and it adds equations given within the block (with the same syntax as *model*).

No variable is removed or has its type changed in the process.

*Example*

```
var c k;

model;
  c + k - aa*x*k(-1)^alph - (1-delt)*k(-1);
  [ name = 'dummy' ]
  c*k = 1;
end;

model_replace('dummy');
  c^(-gam) = (1+bet)^(-1)*(aa*alph*x(+1)*k^(alph-1) + 1 - delt)*c(+1)^(-
↪gam);
end;
```

In the above example, the dummy equation is replaced by a proper Euler equation.

Dynare has the ability to output the original list of model equations to a LaTeX file, using the `write_latex_original_model` command, the list of transformed model equations using the `write_latex_dynamic_model` command, and the list of static model equations using the `write_latex_static_model` command.

**Command:** **write_latex_original_model** ;

**Command:** **write_latex_original_model**(OPTIONS);

This command creates two LaTeX files: one containing the model as defined in the model block and one containing the LaTeX document header information.

If your `.mod` file is `FILENAME.mod`, then Dynare will create a file called `FILENAME/latex/original.tex`, which includes a file called `FILENAME/latex/original_content.tex` (also created by Dynare) containing the list of all the original model equations.

If LaTeX names were given for variables and parameters (see *Variable declarations*), then those will be used; otherwise, the plain text names will be used.

Time subscripts (`t`, `t+1`, `t-1`, ...) will be appended to the variable names, as LaTeX subscripts.

Compiling the TeX file requires the following LaTeX packages: `geometry`, `fullpage`, `breqn`.

*Options*

**write_equation_tags**

Write the equation tags in the LaTeX output. The equation tags will be interpreted with LaTeX markups.

**Command:** **write_latex_dynamic_model** ;

**Command:** **write_latex_dynamic_model**(OPTIONS);

This command creates two LaTeX files: one containing the dynamic model and one containing the LaTeX document header information.

If your `.mod` file is `FILENAME.mod`, then Dynare will create a file called `FILENAME/latex/dynamic.tex`, which includes a file called `FILENAME/latex/dynamic_content.tex` (also created by Dynare) containing the list of all the dynamic model equations.

If LaTeX names were given for variables and parameters (see *Variable declarations*), then those will be used; otherwise, the plain text names will be used.

Time subscripts (`t`, `t+1`, `t-1`, ...) will be appended to the variable names, as LaTeX subscripts.

Note that the model written in the TeX file will differ from the model declared by the user in the following dimensions:

- The timing convention of predetermined variables (see *predetermined_variables*) will have been changed to the default Dynare timing convention; in other words, variables declared as predetermined will be lagged on period back,
- The `EXPECTATION` operators will have been removed, replaced by auxiliary variables and new equations (as explained in the documentation of *EXPECTATION*),
- Endogenous variables with leads or lags greater or equal than two will have been removed, replaced by new auxiliary variables and equations,
- Exogenous variables with leads or lags will also have been replaced by new auxiliary variables and equations.

For the required LaTeX packages, see *write_latex_original_model*.

*Options*

**write_equation_tags**

See *write_equation_tags*

**Command:** **write_latex_static_model** ;

**Command:** **write_latex_static_model**(OPTIONS);

This command creates two LaTeX files: one containing the static model and one containing the LaTeX document header information.

If your `.mod` file is `FILENAME.mod`, then Dynare will create a file called `FILENAME/latex/static.tex`, which includes a file called `FILENAME/latex/static_content.tex` (also created by Dynare) containing the list of all the steady-state model equations.

If LaTeX names were given for variables and parameters (see *Variable declarations*), then those will be used; otherwise, the plain text names will be used.

Note that the model written in the TeX file will differ from the model declared by the user in some dimensions (see *write_latex_dynamic_model* for details).

Also note that this command will not output the contents of the optional *steady_state_model* block; it will rather output a static version (i.e. without leads and lags) of the dynamic `model` declared in the model block. To write the LaTeX contents of the *steady_state_model* block, see *write_latex_steady_state_model*.

For the required LaTeX packages, see *write_latex_original_model*.

*Options*

**write_equation_tags**

See *write_equation_tags*.

**Command:** **write_latex_steady_state_model** ;

This command creates two LaTeX files: one containing the steady-state model and one containing the LaTeX document header information.

If your `.mod` file is `FILENAME.mod`, then Dynare will create a file called `FILENAME/latex/steady_state.tex`, which includes a file called `FILENAME/latex/steady_state_content.tex` (also created by Dynare) containing the list of all the steady-state model equations.

If LaTeX names were given for variables and parameters (see *Variable declarations*), then those will be used; otherwise, the plain text names will be used.

Note that the model written in the `.tex` file will differ from the model declared by the user in some dimensions (see *write_latex_dynamic_model* for details).

For the required LaTeX packages, see *write_latex_original_model*.

## 4.6 Auxiliary variables

The model which is solved internally by Dynare is not exactly the model declared by the user. In some cases, Dynare will introduce auxiliary endogenous variables—along with corresponding auxiliary equations—which will appear in the final output.

The main transformation concerns leads and lags. Dynare will perform a transformation of the model so that there is only one lead and one lag on endogenous variables and no leads/lags on exogenous variables.

This transformation is achieved by the creation of auxiliary variables and corresponding equations. For example, if `x(+2)` exists in the model, Dynare will create one auxiliary variable `AUX_ENDO_LEAD = x(+1)`, and replace `x(+2)` by `AUX_ENDO_LEAD(+1)`.

A similar transformation is done for lags greater than 2 on endogenous (auxiliary variables will have a name beginning with `AUX_ENDO_LAG`), and for exogenous with leads and lags (auxiliary variables will have a name beginning with `AUX_EXO_LEAD` or `AUX_EXO_LAG` respectively).

Another transformation is done for the `EXPECTATION` operator. For each occurrence of this operator, Dynare creates an auxiliary variable defined by a new equation, and replaces the expectation operator by a reference to the new auxiliary variable. For example, the expression `EXPECTATION(-1)(x(+1))` is replaced by `AUX_EXPECT_LAG_1(-1)`, and the new auxiliary variable is declared as `AUX_EXPECT_LAG_1 = x(+2)`.

Auxiliary variables are also introduced by the preprocessor for the *ramsey_model* and `ramsey_policy` commands. In this case, they are used to represent the Lagrange multipliers when first-order conditions of the Ramsey problem are computed. The new variables take the form `MULT_i`, where *i* represents the constraint with which the multiplier is associated (counted from the order of declaration in the model block).

Auxiliary variables are also introduced by the `differentiate_forward_vars` option of the model block. The new variables take the form `AUX_DIFF_FWRD_i`, and are equal to `x-x(-1)` for some endogenous variable `x`.

Finally, auxiliary variables will arise in the context of employing the `diff` operator.

Once created, all auxiliary variables are included in the set of endogenous variables. The output of decision rules (see below) is such that auxiliary variable names are replaced by the original variables they refer to.

The number of endogenous variables before the creation of auxiliary variables is stored in `M_.orig_endo_nbr`, and the number of endogenous variables after the creation of auxiliary variables is stored in `M_.endo_nbr`.

See https://git.dynare.org/Dynare/dynare/-/wikis/Auxiliary-variables for more technical details on auxiliary variables.

## 4.7 Initial and terminal conditions

For most simulation exercises, it is necessary to provide initial (and possibly terminal) conditions. It is also necessary to provide initial guess values for non-linear solvers. This section describes the statements used for those purposes.

In many contexts (deterministic or stochastic), it is necessary to compute the steady state of a non-linear model: `initval` then specifies numerical initial values for the non-linear solver. The command *resid* can be used to compute the equation residuals for the given initial values.

Used in perfect foresight mode, the types of forward-looking models for which Dynare was designed require both initial and terminal conditions. Most often these initial and terminal conditions are static equilibria, but not necessarily.

One typical application is to consider an economy at the equilibrium at time 0, trigger a shock in first period, and study the trajectory of return to the initial equilibrium. To do that, one needs `initval` and `shocks` (see *Shocks on exogenous variables*).

Another one is to study how an economy, starting from arbitrary initial conditions at time 0 converges towards equilibrium. In this case models, the command `histval` permits to specify different historical initial values for variables with lags for the periods before the beginning of the simulation. Due to the design of Dynare, in this case `initval` is used to specify the terminal conditions.

**Block:** **`initval ;`**

**Block:** **`initval`**`(OPTIONS...);`

The `initval` block has two main purposes: providing guess values for non-linear solvers in the context of perfect foresight simulations and providing guess values for steady-state computations in both perfect foresight and stochastic simulations. Depending on the presence of `histval` and `endval` blocks it is also used for declaring the initial and terminal conditions in a perfect foresight simulation exercise. Because of this interaction of the meaning of an `initval` block with the presence of `histval` and `endval` blocks in perfect foresight simulations, it is strongly recommended to check that the constructed `oo_.endo_simul` and `oo_.exo_simul` variables contain the desired values after running *perfect_foresight_setup* and before running *perfect_foresight_solver*. In the presence of leads and lags, these subfields of the results structure will store the historical values for the lags in the first column/row and the terminal values for the leads in the last column/row.

The `initval` block is terminated by `end;` and contains lines of the form:

VARIABLE_NAME = EXPRESSION;

*In a deterministic (i.e. perfect foresight) model*

First, both the `oo_.endo_simul` and `oo_.exo_simul` variables storing the endogenous and exogenous variables will be filled with the values provided by this block. If there are no other blocks present, it will therefore provide the initial and terminal conditions for all the endogenous and exogenous variables, because it will also fill the last column/row of these matrices. For the intermediate simulation periods it thereby provides the starting values for the solver. In the presence of a `histval` block (and therefore absence of an `endval` block), this `histval` block will provide/overwrite the historical values for the state variables (lags) by setting the first column/row of `oo_.endo_simul` and `oo_.exo_simul`. This implies that the `initval` block in the presence of `histval` only sets the terminal values for the variables with leads and provides initial values for the perfect foresight solver.

Because of these various functions of `initval` it is often necessary to provide values for all the endogenous variables in an `initval` block. Initial and terminal conditions are strictly necessary for lagged/leaded variables, while feasible starting values are required for the solver. It is important to be aware that if some variables, endogenous or exogenous, are not mentioned in the `initval` block, a zero value is assumed. It is particularly important to keep this in mind when specifying exogenous variables using `varexo` that are not allowed to take on the value of zero, like e.g. TFP.

Note that if the `initval` block is immediately followed by a *steady* command, its semantics are slightly changed. The *steady* command will compute the steady state of the model for all the endogenous variables, assuming that exogenous variables are kept constant at the value declared in the `initval` block. These steady-state values conditional on the declared exogenous variables are then written into `oo_.endo_simul` and take up the potential roles as historical and terminal conditions as well as starting values for the solver. An `initval` block followed by `steady` is therefore formally equivalent to an `initval` block with the specified values for the exogenous variables, and the endogenous variables set to the associated steady-state values conditional on the exogenous variables.

*In a stochastic model*

The main purpose of `initval` is to provide initial guess values for the non-linear solver in the steady-state computation. Note that if the `initval` block is not followed by *steady*, the steady-state computation will still be triggered by subsequent commands (*stoch_simul*, *estimation*...).

As such, `initval` allows specifying the initial instrument value for steady state finding when providing an analytical conditional steady-state file for *ramsey_model* computations.

It is not necessary to declare 0 as initial value for exogenous stochastic variables, since it is the only possible value.

The subsequently computed steady state (not the initial values, use `histval` for this) will be used as the initial condition at all the periods preceding the first simulation period for the three possible types of simulations in stochastic mode:

- *stoch_simul*, if the *periods* option is specified.
- *forecast* as the initial point at which the forecasts are computed.
- *conditional_forecast* as the initial point at which the conditional forecasts are computed.

To start simulations at a particular set of starting values that are not a computed steady state, use *histval*.

*Options*

**all_values_required**

Issues an error and stops processing the .mod file if there is at least one endogenous or exogenous variable that has not been set in the initval block.

***Example***

```
initval;
c = 1.2;
k = 12;
x = 1;
end;

steady;
```

**Block:** **endval** ;

**Block:** **endval**(OPTIONS...);

This block is terminated by `end;` and contains lines of the form:

VARIABLE_NAME = EXPRESSION;

The `endval` block makes only sense in a deterministic model and cannot be used together with `histval`. Similar to the `initval` command, it will fill both the `oo_.endo_simul` and `oo_.exo_simul` variables storing the endogenous and exogenous variables with the values provided by this block. If no `initval` block is present, it will fill the whole matrices, therefore providing the initial and terminal conditions for all the endogenous and exogenous variables, because it will also fill the first and last column/row of these matrices. Due to also filling the intermediate simulation periods it will provide the starting values for the solver as well.

If an `initval` block is present, `initval` will provide the historical values for the variables (if there are states/lags), while `endval` will fill the remainder of the matrices, thereby still providing *i*) the terminal conditions for variables entering the model with a lead and *ii*) the initial guess values for all endogenous variables at all the simulation dates for the perfect foresight solver.

Note that if some variables, endogenous or exogenous, are NOT mentioned in the `endval` block, the value assumed is that of the last `initval` block or *steady* command (if present). Therefore, in contrast to `initval`, omitted variables are not automatically assumed to be 0 in this case. Again, it is strongly recommended to check the constructed `oo_.endo_simul` and `oo_.exo_simul` variables after running *perfect_foresight_setup* and before running *perfect_foresight_solver* to see whether the desired outcome has been achieved.

Like `initval`, if the `endval` block is immediately followed by a *steady* command, its semantics are slightly changed. The *steady* command will compute the steady state of the model for all the endogenous variables, assuming that exogenous variables are kept constant to the value declared in the `endval` block. These steady-state values conditional on the declared exogenous variables are then written into `oo_.endo_simul` and

therefore take up the potential roles as historical and terminal conditions as well as starting values for the solver. An `endval` block followed by *steady* is therefore formally equivalent to an `endval` block with the specified values for the exogenous variables, and the endogenous variables set to the associated steady-state values.

It is possible to have several `endval` blocks, in which case their contents will be concatenated. Any *steady* command should then follow the last `endval` block.

*Options*

**all_values_required**

See *all_values_required*.

*Example*

```
var c k;
varexo x;

model;
c + k - aa*x*k(-1)^alph - (1-delt)*k(-1);
c^(-gam) - (1+bet)^(-1)*(aa*alph*x(+1)*k^(alph-1) + 1 - delt)*c(+1)^(-
↪gam);
end;

initval;
c = 1.2;
k = 12;
x = 1;
end;

steady;

endval;
c = 2;
k = 20;
x = 2;
end;

steady;

perfect_foresight_setup(periods=200);
perfect_foresight_solver;
```

In this example, the problem is finding the optimal path for consumption and capital for the periods $t = 1$ to $T = 200$, given the path of the exogenous technology level `x`. `c` is a forward-looking variable and the exogenous variable `x` appears with a lead in the expected return of physical capital, while `k` is a purely backward-looking (state) variable.

The initial equilibrium is computed by *steady* conditional on `x=1`, and the terminal one conditional on `x=2`. The `initval` block sets the initial condition for `k` (since it is the only backward-looking variable), while the `endval` block sets the terminal condition for `c` (since it is the only forward-looking endogenous variable). The starting values for the perfect foresight solver are given by the `endval` block. See below for more details.

*Example*

```
var c k;
varexo x;

model;
c + k - aa*x*k(-1)^alph - (1-delt)*k(-1);
```

(continues on next page)

(continued from previous page)

```
c^(-gam) - (1+bet)^(-1)*(aa*alph*x(+1)*k^(alph-1) + 1 - delt)*c(+1)^(-
↪gam);
end;

initval;
k = 12;
end;

endval;
c = 2;
x = 1.1;
end;

perfect_foresight_setup(periods=200);
perfect_foresight_solver;
```

In this example, there is no *steady* command, hence the conditions are exactly those specified in the *initval* and *endval* blocks. We need terminal conditions for `c` and `x`, since both appear with a lead, and an initial condition for `k`, since it appears with a lag.

Setting `x=1.1` in the `endval` block without a `shocks` block implies that technology is at $1.1$ in $t = 1$ and stays there forever, because `endval` is filling all entries of `oo_.endo_simul` and `oo_.exo_simul` except for the very first one, which stores the initial conditions and was set to $0$ by the `initval` block when not explicitly specifying a value for it.

Because the law of motion for capital is backward-looking, we need an initial condition for `k` at time $0$. Due to the presence of `endval`, this cannot be done via a `histval` block, but rather must be specified in the `initval` block. Similarly, because the Euler equation is forward-looking, we need a terminal condition for `c` at $t = 201$, which is specified in the `endval` block.

As can be seen, it is not necessary to specify `c` and `x` in the `initval` block and `k` in the `endval` block, because they have no impact on the results. Due to the optimization problem in the first period being to choose `c,k` at $t = 1$ given the predetermined capital stock `k` inherited from $t = 0$ as well as the current and future values for technology `x`, the values for `c` and `x` at time $t = 0$ play no role. The same applies to the choice of `c,``k`` at time $t = 200$, which does not depend on `k` at $t = 201$. As the Euler equation shows, that choice only depends on current capital as well as future consumption `c` and technology `x`, but not on future capital `k`. The intuitive reason is that those variables are the consequence of optimization problems taking place in at periods $t = 0$ and $t = 201$, respectively, which are not modeled here.

*Example*

```
initval;
c = 1.2;
k = 12;
x = 1;
end;

endval;
c = 2;
k = 20;
x = 1.1;
end;
```

In this example, initial conditions for the forward-looking variables `x` and `c` are provided, together with a terminal condition for the backward-looking variable `k`. As shown in the previous example, these values will not affect the simulation results. Dynare simply takes them as given and basically assumes that there were realizations of exogenous variables and states that make those choices equilibrium values (basically initial/terminal conditions at the unspecified time periods $t < 0$ and

$t > 201$).

The above example suggests another way of looking at the use of *steady* after `initval` and `endval`. Instead of saying that the implicit unspecified conditions before and after the simulation range have to fit the initial/terminal conditions of the endogenous variables in those blocks, steady specifies that those conditions at $t < 0$ and $t > 201$ are equal to being at the steady state given the exogenous variables in the `initval` and `endval` blocks. The endogenous variables at $t = 0$ and $t = 201$ are then set to the corresponding steady-state equilibrium values.

The fact that `c` at $t = 0$ and `k` at $t = 201$ specified in `initval` and `endval` are taken as given has an important implication for plotting the simulated vector for the endogenous variables, i.e. the rows of `oo_.endo_simul`: this vector will also contain the initial and terminal conditions and thus is 202 periods long in the example. When you specify arbitrary values for the initial and terminal conditions for forward- and backward-looking variables, respectively, these values can be very far away from the endogenously determined values at $t = 1$ and $t = 200$. While the values at $t = 0$ and $t = 201$ are unrelated to the dynamics for $0 < t < 201$, they may result in strange-looking large jumps. In the example above, consumption will display a large jump from $t = 0$ to $t = 1$ and capital will jump from $t = 200$ to $t = 201$ when using *rplot* or manually plotting `oo_.endo_simul`.

**Block:** `histval ;`

**Block:** `histval(OPTIONS...);`

*In a deterministic perfect foresight context*

In models with lags on more than one period, the `histval` block permits to specify different historical initial values for different periods of the state variables. In this case, the `initval` block takes over the role of specifying terminal conditions and starting values for the solver. Note that the `histval` block does not take non-state variables.

This block is terminated by `end;` and contains lines of the form:

VARIABLE_NAME(INTEGER) = EXPRESSION;

EXPRESSION is any valid expression returning a numerical value and can contain already initialized variable names.

By convention in Dynare, period 1 is the first period of the simulation. Going backward in time, the first period before the start of the simulation is period 0, then period -1, and so on.

State variables not initialized in the `histval` block are assumed to have a value of zero at period 0 and before. Note that `histval` cannot be followed by *steady*.

*Example*

```
model;
x=1.5*x(-1)-0.6*x(-2)+epsilon;
log(c)=0.5*x+0.5*log(c(+1));
end;

histval;
x(0)=-1;
x(-1)=0.2;
end;

initval;
c=1;
x=1;
end;
```

In this example, `histval` is used to set the historical conditions for the two lags of the endogenous variable `x`, stored in the first column of `oo_.endo_simul`. The `initval` block is used to set the terminal condition for the forward-looking variable `c`, stored in the last column of `oo_.`

endo_simul. Moreover, the initval block defines the starting values for the perfect foresight solver for both endogenous variables c and x.

*In a stochastic simulation context*

In the context of stochastic simulations, histval allows setting the starting point of those simulations in the state space. As for the case of perfect foresight simulations, all not explicitly specified variables are set to 0. Moreover, as only states enter the recursive policy functions, all values specified for control variables will be ignored. This can be used

- In *stoch_simul*, if the periods option is specified. Note that this only affects the starting point for the simulation, but not for the impulse response functions. When using the *loglinear* option, the histval block nevertheless takes the unlogged starting values.
- In *forecast* as the initial point at which the forecasts are computed. When using the *loglinear* option, the histval block nevertheless takes the unlogged starting values.
- In *conditional_forecast* for a calibrated model as the initial point at which the conditional forecasts are computed. When using the *loglinear* option, the histval block nevertheless takes the unlogged starting values.
- In *Ramsey policy*, where it also specifies the values of the endogenous states (including lagged exogenous) at which the objective function of the planner is computed. Note that the initial values of the Lagrange multipliers associated with the planner's problem cannot be set (see *evaluate_planner_objective*).

*Options*

**all_values_required**

See *all_values_required*.

*Example*

```
var x y;
varexo e;

model;
x = y(-1)^alpha*y(-2)^(1-alpha)+e;

end;

initval;
x = 1;
y = 1;
e = 0.5;
end;

steady;

histval;
y(0) = 1.1;
y(-1) = 0.9;
end;

stoch_simul(periods=100);
```

**Command:** **resid** ;

**Command:** **resid**(OPTIONS...);

This command will display the residuals of the static equations of the model, using the values given for the endogenous in the last initval or endval block (or the steady-state file if you provided one, see *Steady state*).

*Options*

**non_zero**

Only display non-zero residuals.

**Command:** `initval_file(OPTIONS...);`

In a deterministic setup, this command is used to specify a path for all endogenous and exogenous variables. The length of these paths must be equal to the number of simulation periods, plus the number of leads and the number of lags of the model (for example, with 50 simulation periods, in a model with 2 lags and 1 lead, the paths must have a length of 53). Note that these paths cover two different things:

- The constraints of the problem, which are given by the path for exogenous and the initial and terminal values for endogenous
- The initial guess for the non-linear solver, which is given by the path for endogenous variables for the simulation periods (excluding initial and terminal conditions)

In perfect foresight and stochastic contexts, *steady* uses the first observation loaded by `initval_file` as guess value to solve for the steady state of the model. This first observation is determined by the `first_obs` option when it is used.

Don't mix `initval_file` with `initval` statements. However, after `initval_file`, you can modify the historical initial values with `histval` or `histval_file` statement.

There can be several `initval_file` statements in a model file. Each statement resets `oo_.initval_series`.

*Options*

**datafile = FILENAME**

**filename = FILENAME (deprecated)**

The name of the file containing the data. It must be included in quotes if the filename contains a path or an extension. The command accepts the following file formats:

- M-file (extension `.m`): for each endogenous and exogenous variable, the file must contain a row or column vector of the same name.
- MAT-file (extension `.mat`): same as for M-files.
- Excel file (extension `.xls` or `.xlsx`): for each endogenous and exogenous variable, the file must contain a column of the same name. NB: Octave only supports the `.xlsx` file extension and must have the io package installed. The first column may contain the date of each observation. The whole first sheet of the file is read.
- CSV files (extension `.csv`): for each endogenous and exogenous variable, the file must contain a column of the same name. The first column may contain the date of each observation.

**first_obs = {INTEGER | DATE}**

The observation number or the date (see *The dates class*) of the first observation to be used in the file

**first_simulation_period = {INTEGER | DATE}**

The observation number in the file or the date (see *dates*) at which the simulation (or the forecast) is starting. This option avoids to have to compute the maximum number of lags in the model. The observation corresponding to the first period of simulation doesn't need to exist in the file as the only dates necessary for initialization are before that date.

**last_simulation_period = {INTEGER | DATE}**

The observation number in the file or the date (see *dates*) at which the simulation (or the forecast) is ending. This option avoids having to compute the maximum number of leads in the model.

**last_obs = {INTEGER | DATE}**

The observation number or the date (see *The dates class*) of the last observation to be used in the file.

**nobs = INTEGER**

The number of observations to be used in the file (starting with first of `first_obs` observation).

**series = DSERIES NAME**

The name of a DSERIES containing the data (see *The dseries class*)

*Example 1*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

initval_file(datafile=mydata.csv);

perfect_foresight_setup(periods=200);
perfect_foresight_solver;
```

The initial and terminal values are taken from file `mydata.csv` (nothing guarantees that these vales are the steady state of the model). The guess value for the trajectories is also taken from the file. The file must contain at least 203 observations of variables `c`, `x` and `e`. If there are more than 203 observations available in the file, the first 203 are used by `perfect_foresight_setup(periods=200)`. Note that the values for the auxiliary variable corresponding to `x(-2)` are automatically computed by `initval_file`.

*Example 2*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

initval_file(datafile=mydata.csv,
             first_obs=10);

perfect_foresight_setup(periods=200);
perfect_foresight_solver;
```

The initial and terminal values are taken from file `mydata.csv` starting with the 10th observation in the file, for a total of 203 observations used.

*Example 3*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

ds = dseries(mydata.csv);
lds = log(ds);

initval_file(series=lds,
             first_obs=2010Q1);

perfect_foresight_setup(periods=200);
perfect_foresight_solver;
```

The initial and terminal values are taken from dseries `lds`. All observations are loaded starting with the 1st quarter of 2010 until the end of the file. There must be data available at least until 2050Q3.

*Example 4*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

initval_file(datafile=mydata.csv,
             first_simulation_period=2010Q1);

perfect_foresight_setup(periods=200);
perfect_foresight_solver;
```

The initial and terminal values are taken from file `mydata.csv`. The observations in the file must have dates. All observations are loaded from the 3rd quarter of 2009 until the end of the file. There must be data available in the file at least until 2050Q1.

*Example 5*

```
var c x;
varexo e;
parameters a b c d;
```

(continues on next page)

(continued from previous page)

```
a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

initval_file(datafile=mydata.csv,
             last_obs = 212);

perfect_foresight_setup(periods=200);
perfect_foresight_solver;
```

The initial and terminal values are taken from file `mydata.csv`. Observations from 10 to 212 will be used by `perfect_foresight_setup(periods=200)`.

*Example 6*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

initval_file(datafile=mydata.csv,
             first_obs = 10,
             nobs = 203);

perfect_foresight_setup(periods=200);
perfect_foresight_solver;
```

The initial and terminal values are taken from file `mydata.csv`. Observations 10 to 212 are loaded.

*Example 7*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
```

(continues on next page)

(continued from previous page)

```
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

initval_file(datafile=mydata.csv,
             first_obs = 10);

steady;
```

The values of the 10th observation of `mydata.csv` are used as guess value to compute the steady state. The exogenous variables are set to values found in the file or zero if these variables aren't present.

**Command:** **histval_file**(OPTIONS...);

This command is equivalent to `histval`, except that it reads its input from a file. The number of observations that will be used in the file is equal to the maximum number of lags in the model.

*Options*

**datafile = FILENAME**

**filename = FILENAME (deprecated)**

The name of the file containing the data. The command accepts the following file formats:

- M-file (extension `.m`): for each endogenous and exogenous variable, the file must contain a row or column vector of the same name.
- MAT-file (extension `.mat`): same as for M-files.
- Excel file (extension `.xls` or `.xlsx`): for each endogenous and exogenous variable, the file must contain a column of the same name. NB: Octave only supports the `.xlsx` file extension and must have the io package installed. The first column may contain the date of each observation. The whole first sheet of the file is read.
- CSV files (extension `.csv`): for each endogenous and exogenous variable, the file must contain a column of the same name. The first column may contain the date of each observation.

**first_obs = {INTEGER | DATE}**

The observation number or the date (see *The dates class*) of the first observation to be used in the file.

**first_simulation_period = {INTEGER | DATE}**

The observation number in the file or the date (see *The dates class*) at which the simulation (or the forecast) is starting. This option avoids having to compute the maximum number of lags in the model. The observation corresponding to the first period of simulation doesn't need to exist in the file as the only dates necessary for initialization are before that date.

**last_obs = {INTEGER | DATE}**

The observation number or the date (see *The dates class*) of the last observation to be used in the file.

**series = DSERIES NAME**

The name of a DSERIES containing the data (see *The dseries class*)

*Example 1*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;
```

(continues on next page)

(continued from previous page)

```
model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

steady_state_model;
x = 0;
c = exp(c*x/(1 - d));
end;

histval_file(datafile=mydata.csv);

stoch_simul(order=1,periods=100);
```

The initial values for the stochastic simulation are taken from the two first rows of file `mydata.csv`.

*Example 2*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

histval_file(datafile=mydata.csv,
             first_obs=10);

stoch_simul(order=1,periods=100);
```

The initial values for the stochastic simulation are taken from rows 10 and 11 of file `mydata.csv`.

*Example 3*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

histval_file(datafile=mydata.csv,
```

(continues on next page)

(continued from previous page)

```
            first_obs=2010Q1);

stoch_simul(order=1,periods=100);
```

The initial values for the stochastic simulation are taken from observations 2010Q1 and 2010Q2 of file `mydata.csv`. The observations in the file must have dates.

*Example 4*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

histval_file(datafile=mydata.csv,
             first_simulation_period=2010Q1)

stoch_simul(order=1,periods=100);
```

The initial values for the stochastic simulation are taken from observations 2009Q3 and 2009Q4 of file `mydata.csv`. The observations in the file must have dates.

*Example 5*

```
var c x;
varexo e;
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

histval_file(datafile=mydata.csv,
             last_obs = 4);

stoch_simul(order=1,periods=100);
```

The initial values for the stochastic simulation are taken from rows 3 and 4 of file `mydata.csv`.

*Example 6*

```
var c x;
varexo e;
```

(continues on next page)

(continued from previous page)

```
parameters a b c d;

a = 1.5;
b = -0,6;
c = 0.5;
d = 0.5;

model;
x = a*x(-1) + b*x(-2) + e;
log(c) = c*x + d*log(c(+1));
end;

initval_file(datafile=mydata.csv,
             first_obs=10);

histval_file(datafile=myotherdata.csv);

perfect_foresight_setup(periods=200);
perfect_foresight_solver;
```

Historical initial values for the simulation are taken from the two first rows of file `myotherdata.csv`.

Terminal values and guess values for the simulation are taken from file `mydata.csv` starting with the 12th observation in the file, for a total of 201 observations used.

## 4.8 Shocks on exogenous variables

In a deterministic context, when one wants to study the transition of one equilibrium position to another, it is equivalent to analyze the consequences of a permanent shock and this in done in Dynare through the proper use of `initval` and `endval`.

Another typical experiment is to study the effects of a temporary shock after which the system goes back to the original equilibrium (if the model is stable...). A temporary shock is a temporary change of value of one or several exogenous variables in the model. Temporary shocks are specified with the command `shocks`.

In a stochastic framework, the exogenous variables take random values in each period. In Dynare, these random values follow a skew normal distribution with zero mean, but the user must specify the variability and skewness of these shocks. The non-zero elements of the covariance matrix and the coskewness tensor can be entered with the `shocks` command.

If the variance of an exogenous variable is set to zero, this variable will appear in the report on policy and transition functions, but isn't used in the computation of moments and of Impulse Response Functions. Setting a variance to zero is an easy way of removing an exogenous shock. If not specified, all skewness coefficients are assumed to be zero, and thus the shocks follow a Gaussian distribution.

Note that, by default, if there are several `shocks` or `mshocks` blocks in the same `.mod` file, then they are cumulative: all the shocks declared in all the blocks are considered; however, if a `shocks` or `mshocks` block is declared with the `overwrite` option, then it replaces all the previous `shocks` and `mshocks` blocks.

**Block:** `shocks ;`

**Block:** `shocks(overwrite);`

See above for the meaning of the `overwrite` option.

*In deterministic context*

For deterministic simulations, the `shocks` block specifies temporary changes in the value of exogenous variables. For permanent shocks, use an `endval` block.

The block should contain one or more occurrences of the following group of three lines:

```
var EXOGENOUS_NAME;
periods INTEGER[:INTEGER] | DATE[:DATE] [[,] INTEGER[:INTEGER] | DATE[:DATE]]...;
values DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
```

It is possible to specify shocks which last several periods and which can vary over time. The `periods` keyword accepts a list of several dates or date ranges, which must be matched by as many shock values in the `values` keyword. Note that a range in the `periods` keyword can be matched by only one value in the `values` keyword. If `values` represents a scalar, the same value applies to the whole range. If `values` represents a vector, it must have as many elements as there are periods in the range.

Note that shock values are not restricted to numerical constants: arbitrary expressions are also allowed, but you have to enclose them inside parentheses.

The feasible range of `periods`, when specified as integers, is from 0 to the number of `periods` specified in *perfect_foresight_setup* or *perfect_foresight_with_expectation_errors_setup*. Alternatively, it is possible to use real dates if the `first_simulation_period` and/or `last_simulation_period` option has been passed to the aforementioned commands.

**Warning**

Note that the first endogenous simulation period is period 1 (when specified as an integer). Thus, a shock value specified for the initial period 0 may conflict with (i.e. may overwrite or be overwritten by) values for the initial period specified with `initval` or `endval` (depending on the exact context). Users should always verify the correct setting of `oo_.exo_simul` after *perfect_foresight_setup* or *perfect_foresight_with_expectation_errors_setup*.

*Example* (with scalar values)

```
shocks;

var e;
periods 1;
values 0.5;
var u;
periods 4:5;
values 0;
var v;
periods 4:5 6 7:9;
values 1 1.1 0.9;
var w;
periods 1 2;
values (1+p) (exp(z));

end;
```

*Example* (with vector values)

```
xx = [1.2; 1.3; 1];

shocks;
var e;
periods 1:3;
values (xx);
end;
```

*Example* (with dates)

```
shocks;
  var e;
  periods 2023Q1;
  values 0.5;

  var u;
  periods 2023Q2:2023Q4 2024Q1;
  values 0 0.1;
end;
```

*In stochastic context*

For stochastic simulations, the `shocks` block specifies the non-zero elements of the covariance matrix and coskewnes tensor of the shocks of exogenous variables.

You can use the following types of entries in the block:

- Specification of the standard error of an exogenous variable.

  ```
  var VARIABLE_NAME; stderr EXPRESSION;
  ```

- Specification of the variance of an exogenous variable.

  ```
  var VARIABLE_NAME = EXPRESSION;
  ```

- Specification the covariance of two exogenous variables.

  ```
  var VARIABLE_NAME, VARIABLE_NAME = EXPRESSION;
  ```

- Specification of the correlation of two exogenous variables.

  ```
  corr VARIABLE_NAME, VARIABLE_NAME = EXPRESSION;
  ```

- Specification of the skewness of an exogenous variable.

  ```
  skew VARIABLE_NAME = EXPRESSION;
  ```

- Specification of the co-skewness of three exogenous variables.

  ```
  skew VARIABLE_NAME, VARIABLE_NAME, VARIABLE_NAME = EXPRESSION;
  ```

In an estimation context, it is also possible to specify variances and covariances (but not skewness) on endogenous variables: in that case, these values are interpreted as the calibration of the measurement errors on these variables. This requires the `varobs` command to be specified before the `shocks` block.

*Example*

```
shocks;
var e = 0.000081;
var u; stderr 0.009;
corr e, u = 0.8;
var v, w = 2;
skew e = 1;
skew u, v, w = 2;
end;
```

*In stochastic optimal policy context*

When computing conditional welfare in a *ramsey_model* or *discretionary_policy* context, welfare is conditional on the state values inherited by planner when making choices in the first period. The information set of the first period includes the respective exogenous shock realizations. Thus, their known value can be

specified using the perfect foresight syntax. Note that i) all other values specified for periods than period 1 will be ignored and ii) the value of lagged shocks (e.g. in the case of news shocks) is specified with `histval`.

*Example*

```
shocks;
 var u; stderr 0.008;
 var u;
 periods 1;
 values 1;
end;
```

*Mixing deterministic and stochastic shocks*

It is possible to mix deterministic and stochastic shocks to build models where agents know from the start of the simulation about future exogenous changes. In that case *stoch_simul* will compute the rational expectation solution adding future information to the state space (nothing is shown in the output of *stoch_simul*) and `forecast` will compute a simulation conditional on initial conditions and future information.

*Example*

```
varexo_det tau;
varexo e;
...
shocks;
var e; stderr 0.01;
var tau;
periods 1:9;
values -0.15;
end;

stoch_simul(irf=0);

forecast;
```

**Block:** **mshocks** ;

**Block:** **mshocks**(OPTIONS...);

The purpose of this block is similar to that of the `shocks` block for deterministic shocks, except that the numeric values given will be interpreted in a multiplicative way. For example, if a value of `1.05` is given as shock value for some exogenous at some date, it means 5% above its steady-state value.

If no `endval` block is present, the steady state as specified in the `initval` block is used as the basis for the multiplication. If an `endval` block is present, the terminal steady state as specified in the `endval` block will be used as the basis for the multiplication (unless the `relative_to_initval` option is passed).

The syntax is the same as `shocks` in a deterministic context.

This command is only meaningful in two situations:

- on exogenous variables with a non-zero steady state, in a deterministic setup,
- on deterministic exogenous variables with a non-zero steady state, in a stochastic setup.

*Options*

**overwrite**

Same meaning as in the *shocks* block.

**relative_to_initval**

If an `endval` block is present, the initial steady state as specified in the `initval` block will be used as the basis for multiplication (instead of the terminal steady state).

**Block:** **heteroskedastic_shocks** ;

**Block:** `heteroskedastic_shocks(overwrite);`

In *estimation context*, it implements heteroskedastic filters, where the standard error of shocks or measurement errors may unexpectedly change in every period. The standard deviation of shocks or measurement errors may be either provided directly or set/modified in each observed period by a scale factor. If `std0` is the usual standard error for `shock1`, then:

- using a scale factor in period `t` implies: `std(shock1|t)=std0(shock1)*scale(t)`
- using a provided value in period `t` implies: `std(shock1|t)=value(t)`.

The block has a similar syntax as the `shocks` block in a perfect foresight context. It should contain one or more occurrences of the following group of three lines (for setting values):

```
var VARIABLE_NAME;
periods INTEGER[:INTEGER] | DATE[:DATE] [[,] INTEGER[:INTEGER] | DATE[:DATE]]...;
values DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
```

or (for setting scale factors):

```
var VARIABLE_NAME;
periods INTEGER[:INTEGER] | DATE[:DATE] [[,] INTEGER[:INTEGER] | DATE[:DATE]]...;
scales DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
```

*Notes*

- Integer period indices refer to the original dataset (*e.g.* if `first_obs=10`, then a period index of 11 will refer to the second observation used for estimation).
- Period indices can only be specified as dates if the dataset has either been specified as a `.csv` file with dates in the `datafile` or `dataseries` option of *estimation*, or with the `series` option of the *data* command.
- `scales` and `values` cannot be simultaneously set for the same shock in the same period, but it is possible to set `values` for some periods and `scales` for other periods for the same shock.
- There can be only one `scales` and `values` directive each for a given shock, so all affected periods must be set in one statement.
- This block is not compatible with *analytic_derivation*.

*Example*

```
heteroskedastic_shocks;

var e1;
periods 86:87, 89:97;
scales 0.5, 0;

var e1;
periods 88;
values 0.1;

var e2;
periods 86:87 88:97;
values 0.04 0.01;

end;
```

**MATLAB/Octave command:** `get_shock_stderr_by_name('EXOGENOUS_NAME');`

Given the name of an exogenous variable, returns its standard deviation, as set by a previous `shocks` block.

**MATLAB/Octave command:** **set_shock_stderr_value**('EXOGENOUS_NAME', MATLAB_EXPRESSION);

Sets the standard deviation of an exogenous variable. This does essentially the same as setting the standard error via a `shocks` block, except that it accepts arbitrary MATLAB/Octave expressions, and that it works from MATLAB/Octave scripts.

**MATLAB/Octave command:**
**get_shock_skew_by_name**('EXOGENOUS_NAME1', 'EXOGENOUS_NAME2', 'EXOGENOUS_NAME3');

Given the name of a single exogenous variable, returns its skewness coefficient, as set by a previous `shocks` block. Given the names of three exogenous variables, returns the co-skewness coefficient, as set by a previous `shocks` block.

**MATLAB/Octave command:** **set_shock_skew_value**('EXOGENOUS_NAME1', 'EXOGENOUS_NAME2',␣
'EXOGENOUS_NAME3', MATLAB_EXPRESSION);

Sets the skewness coefficient of an exogenous variable, or the co-skewness coefficient of three exogenous variables, as set by a previous `shocks` block.

## 4.9 Other general declarations

**Command:** **dsample** INTEGER [INTEGER];

Reduces the number of periods considered in subsequent output commands.

## 4.10 Steady state

There are two ways of computing the steady state (i.e. the static equilibrium) of a model. The first way is to let Dynare compute the steady state using a nonlinear Newton-type solver; this should work for most models, and is relatively simple to use. The second way is to give more guidance to Dynare, using your knowledge of the model, by providing it with a method to compute the steady state, either using a *steady_state_model* block or writing a MATLAB routine.

### 4.10.1 Finding the steady state with Dynare nonlinear solver

**Command:** **steady** ;

**Command:** **steady**(OPTIONS...);

This command computes the steady state of a model using a nonlinear Newton-type solver and displays it. When a steady-state file is used *steady* displays the steady state and checks that it is a solution of the static model.

More precisely, it computes the equilibrium value of the endogenous variables for the value of the exogenous variables specified in the previous `initval` or `endval` block.

*steady* uses an iterative procedure and takes as initial guess the value of the endogenous variables set in the previous `initval` or `endval` block.

For complicated models, finding good numerical initial values for the endogenous variables is the trickiest part of finding the equilibrium of that model. Often, it is better to start with a smaller model and add new variables one by one.

*Options*

**maxit = INTEGER**

Determines the maximum number of iterations used in the non-linear solver. The default value of `maxit` is 50.

**tolf = DOUBLE**

Convergence criterion for termination based on the function value. Iteration will cease when the residuals are smaller than `tolf`. Default: `eps^(1/3)`

**tolx = DOUBLE**

Convergence criterion for termination based on the step tolerance along. Iteration will cease when the attempted step size is smaller than `tolx`. Default: `eps^(2/3)`

**non_zero**

See *non_zero*.

**solve_algo = INTEGER**

Determines the non-linear solver to use. Possible values for the option are:

`0`

Use `fsolve` (under MATLAB, only available if you have the Optimization Toolbox; always available under Octave).

`1`

Use a Newton-like algorithm with line-search.

`2`

Splits the model into recursive blocks and solves each block in turn using the same solver as value `1`.

`3`

Use Chris Sims' solver.

`4`

Splits the model into recursive blocks and solves each block in turn using a trust-region solver with autoscaling.

`5`

Newton algorithm with a sparse Gaussian elimination (SPE) solver at each iteration. This algorithm requires the *bytecode* option. The *markowitz* option can be used to control the behaviour of the algorithm.

`6`

Newton algorithm with a sparse LU solver at each iteration.

`7`

Newton algorithm with a Generalized Minimal Residual (GMRES) solver at each iteration.

`8`

Newton algorithm with a Stabilized Bi-Conjugate Gradient (BiCGStab) solver at each iteration.

`9`

Trust-region algorithm with autoscaling (same as value `4`, but applied to the entire model, without splitting).

`10`

Levenberg-Marquardt mixed complementarity problem (LMMCP) solver (Kanzow and Petra, 2004). The complementarity conditions are specified using the perpendicular symbol, see *lmmcp*.

`11`

PATH mixed complementarity problem solver of Ferris and Munson (1999). The complementarity conditions are specified using the perpendicular symbol, see *lmmcp*. Dynare only provides the interface for using the solver. Due to licence restrictions, you have to download the solver's most current version yourself from http://pages.cs.wisc.edu/~ferris/path.html and place it in MATLAB's search path.

12

Computes a block decomposition and then applies a Newton-type solver on those smaller blocks rather than on the full nonlinear system. This is similar to 2, but is typically more efficient. The block decomposition is done at the preprocessor level, which brings two benefits: it identifies blocks that can be evaluated rather than solved; and evaluations of the residual and Jacobian of the model are more efficient because only the relevant elements are recomputed at every iteration. This option is typically used with the *perfect_foresight_solver* command with purely backward, forward or static models, or with routines for semi-structural models, and it must *not* be combined with option block of the *model* block or *model_options* command. Also note that for those models, the block decomposition is performed as if mfs=3 had been passed to the *model* block or *model_options* command, and the decomposition is slightly different because it is computed in a time-recursive fashion (*i.e.* in such a way that the simulation is meant to be done with the outer loop on periods and the inner loop on blocks; while for models with both leads and lags, the outer loop is on blocks and the inner loop is on periods).

14

Same as 12, except that it applies a trust region solver (similar to 4) to the blocks.

Default value is 4.

**homotopy_mode = INTEGER**

Use a homotopy (or divide-and-conquer) technique to solve for the steady state. If you use this option, you must specify a homotopy_setup block. This option can take three possible values:

0

Do not use homotopy.

1

In this mode, all the parameters are changed simultaneously, and the distance between the boundaries for each parameter is divided in as many intervals as there are steps (as defined by the homotopy_steps option); the problem is solved as many times as there are steps.

2

Same as mode 1, except that only one parameter is changed at a time; the problem is solved as many times as steps times number of parameters.

3

Dynare tries first the most extreme values. If it fails to compute the steady state, the interval between initial and desired values is divided by two for all parameters. Every time that it is impossible to find a steady state, the previous interval is divided by two. When it succeeds to find a steady state, the previous interval is multiplied by two. In that last case homotopy_steps contains the maximum number of computations attempted before giving up.

Default value is 0.

**homotopy_steps = INTEGER**

Defines the number of steps when performing a homotopy. See homotopy_mode option for more details. Default is 10.

**homotopy_force_continue = INTEGER**

This option controls what happens when homotopy fails.

0

*steady* fails with an error message

1

*steady* keeps the values of the last homotopy step that was successful and continues. **BE CAREFUL**: parameters and/or exogenous variables are NOT at the value expected by the user

Default is `0`.

**nocheck**

Don't check the steady-state values when they are provided explicitly either by a steady-state file or a *steady_state_model* block. This is useful for models with unit roots as, in this case, the steady state is not unique or doesn't exist.

**noprint**

Don't print anything. Useful for loops.

**markowitz = DOUBLE**

Value of the Markowitz criterion (in the interval $(0, \infty)$) used to select the pivot with sparse Gaussian elimination (`solve_algo = 5`). This criterion governs the tradeoff between selecting the pivot resulting in the most accurate solution (low `markowitz` values) and the one that preserves maximum sparsity (high `markowitz` values). See https://git.dynare.org/Dynare/dynare/-/wikis/MarkowitzCriterion for more mathematical details. Default: 0.5.

**fsolve_options = (NAME, VALUE, ...)**

A list of NAME and VALUE pairs. Can be used to set options for the `fsolve` routine, which is selected when `solve_algo = 0` (this option has no effect for other values of `solve_algo`). For the list of available name/value pairs, see the documentation of `fsolve` in the MATLAB or Octave manual. Note that Dynare already uses the values of the `maxit`, `tolf` and `tolx` options of the *steady* command for initializing the corresponding options passed to `fsolve`, so you should not need to override those. Also note that you should not try to override the value of the `Jacobian` or `SpecifyObjectiveGradient` option.

*Example*

See *Initial and terminal conditions*.

After computation, the steady state is available in the following variable:

**MATLAB/Octave variable: oo_.steady_state**

Contains the computed steady state. Endogenous variables are ordered in the order of declaration used in the `var` command (which is also the order used in `M_.endo_names`).

**MATLAB/Octave variable: oo_.exo_steady_state**

Contains the steady state of the exogenous variables, as declared by the previous `initval` or `endval` block. Exogenous variables are ordered in the order of declaration used in the `varexo` command (which is also the order used in `M_.exo_names`).

**MATLAB/Octave command: get_mean('ENDOGENOUS_NAME' [, 'ENDOGENOUS_NAME']... );**

Returns the steady of state of the given endogenous variable(s), as it is stored in `oo_.steady_state`. Note that, if the steady state has not yet been computed with *steady*, it will first try to compute it.

**Block: homotopy_setup ;**

**Block: homotopy_setup(from_initval_to_endval) ;**

This block is used to declare initial and final values when using a homotopy method. It is used in conjunction with the option `homotopy_mode` of the *steady* command.

The idea of homotopy (also called divide-and-conquer by some authors) is to subdivide the problem of finding the steady state into smaller problems. It assumes that you know how to compute the steady state for a given set of parameters, and it helps you to find the steady state for another set of parameters, by incrementally moving from one to another set of parameters.

The purpose of the `homotopy_setup` block is to declare the final (and possibly also the initial) values for the parameters or exogenous that will be changed during the homotopy. It should contain lines of the form:

```
VARIABLE_NAME, EXPRESSION, EXPRESSION;
```

This syntax specifies the initial and final values of a given parameter/exogenous.

There is an alternative syntax:

```
VARIABLE_NAME, EXPRESSION;
```

Here only the final value is specified for a given parameter/exogenous; the initial value is taken from the preceding `initval` block (or from the preceding `endval` block if there is one before the `homotopy_setup` block).

A necessary condition for a successful homotopy is that Dynare must be able to solve the steady state for the initial parameters/exogenous without additional help (using the guess values given in the `initval` or `endval` block).

The `from_initval_to_endval` option can be used in the context of a permanent shock, when the initial steady state has already been computed. This option can be used following the `endval` block that describes the terminal steady state. In that case, in the subsequent *steady* command, Dynare will perform a homotopy from the initial to the terminal steady state (technically, using this option is equivalent to writing a `homotopy_setup` block where all exogenous variables are asked to transition from their values in the `initval` to their values in the `endval` block). When this option is used, the `homotopy_setup` block is typically empty (but it's nevertheless possible to add explicit directives for moving exogenous or parameters; these will be added on top of those implicitly generated by the `from_initval_to_endval` option).

If the homotopy fails, a possible solution is to increase the number of steps (given in `homotopy_steps` option of *steady*).

*Example*

In the following example, Dynare will first compute the steady state for the initial values (`gam=0.5` and `x=1`), and then subdivide the problem into 50 smaller problems to find the steady state for the final values (`gam=2` and `x=2`):

```
var c k;
varexo x;

parameters alph gam delt bet aa;
alph=0.5;
delt=0.02;
aa=0.5;
bet=0.05;

model;
c + k - aa*x*k(-1)^alph - (1-delt)*k(-1);
c^(-gam) - (1+bet)^(-1)*(aa*alph*x(+1)*k^(alph-1) + 1 - delt)*c(+1)^(-gam);
end;

initval;
x = 1;
k = ((delt+bet)/(aa*x*alph))^(1/(alph-1));
c = aa*x*k^alph-delt*k;
end;

homotopy_setup;
gam, 0.5, 2;
x, 2;
end;

steady(homotopy_mode = 1, homotopy_steps = 50);
```

### 4.10.2 Providing the steady state to Dynare

If you know how to compute the steady state for your model, you can provide a MATLAB/Octave function doing the computation instead of using *steady*. Again, there are two options for doing that:

- The easiest way is to write a *steady_state_model* block, which is described below in more details. See also `fs2000.mod` in the `examples` directory for an example. The steady state file generated by Dynare will be called `+FILENAME/steadystate.m`.
- You can write the corresponding MATLAB function by hand. If your `.mod` file is called `FILENAME.mod`, the steady-state file must be called `FILENAME_steadystate.m`. See `NK_baseline_steadystate.m` in the `examples` directory for an example. This option gives a bit more flexibility (loops and conditional structures can be used), at the expense of a heavier programming burden and a lesser efficiency.

Note that both files allow to update parameters in each call of the function. This allows for example to calibrate a model to a labor supply of 0.2 in steady state by setting the labor disutility parameter to a corresponding value (see `NK_baseline_steadystate.m` in the `examples` directory). They can also be used in estimation where some parameter may be a function of an estimated parameter and needs to be updated for every parameter draw. For example, one might want to set the capital utilization cost parameter as a function of the discount rate to ensure that capacity utilization is 1 in steady state. Treating both parameters as independent or not updating one as a function of the other would lead to wrong results. But this also means that care is required. Do not accidentally overwrite your parameters with new values as it will lead to wrong results.

**Block: steady_state_model ;**

When the analytical solution of the model is known, this command can be used to help Dynare find the steady state in a more efficient and reliable way, especially during estimation where the steady state has to be recomputed for every point in the parameter space.

Each line of this block consists of a variable (either an endogenous, a temporary variable or a parameter) which is assigned an expression (which can contain parameters, exogenous at the steady state, or any endogenous or temporary variable already declared above). Each line therefore looks like:

```
VARIABLE_NAME = EXPRESSION;
```

Note that it is also possible to assign several variables at the same time, if the main function in the right-hand side is a MATLAB/Octave function returning several arguments:

```
[ VARIABLE_NAME, VARIABLE_NAME... ] = EXPRESSION;
```

There is also the possibility of writing assignments using the following special form:

```
VARIABLE_NAME = solve_from_equation('EQUATION_NAME');
```

In this case, Dynare will first look for an equation named *EQUATION_NAME* (*i.e.* tagged with `[name = 'EQUATION_NAME']`, see *model*), then will solve for the value of *VARIABLE_NAME* that is implied by this equation (given the values of other variables appearing in that equation, which should have been assigned earlier in the *steady_state_model* block). A restriction is that the variable must appear only once in the equation. The equation can contain leads or lags, they will be ignored for the purpose of this computation. Note that if two equations have the same name, one marked with the `[static]` tag and another with the `[dynamic]` tag (see *Replace some equations during steady-state computations*), then the `[static]` one is used.

Dynare will automatically generate a steady-state file (of the form `+FILENAME/steadystate.m`) using the information provided in this block.

*Steady state file for deterministic models*

The *steady_state_model* block also works with deterministic models. An `initval` block and, when necessary, an `endval` block, is used to set the value of the exogenous variables. Each `initval` or `endval` block must be followed by *steady* to execute the function created by *steady_state_model* and set the initial, respectively terminal, steady state.

*Example*

```
var m P c e W R k d n l gy_obs gp_obs y dA;
varexo e_a e_m;

parameters alp bet gam mst rho psi del;

...
model;
...
  [name = 'Intratemporal labour market condition']
  -(psi/(1-psi))*(c*P/(1-n))+l/n = 0;
...
end;

...
// parameter calibration, shock calibration...
...

steady_state_model;
  dA = exp(gam);
  gst = 1/dA; // A temporary variable
  m = mst;

  // Three other temporary variables
  khst = ( (1-gst*bet*(1-del)) / (alp*gst^alp*bet) )^(1/(alp-1));
  xist = ( ((khst*gst)^alp - (1-gst*(1-del))*khst)/mst )^(-1);
  nust = psi*mst^2/( (1-alp)*(1-psi)*bet*gst^alp*khst^alp );

  n  = xist/(nust+xist);
  P  = xist + nust;
  k  = khst*n;

  c  = mst/P;
  l = solve_from_equation('Intratemporal labour market condition');
  d  = l - mst + 1;
  y  = k^alp*n^(1-alp)*gst^alp;
  R  = mst/bet;

  // You can use MATLAB functions which return several arguments
  [W, e] = my_function(l, n);

  gp_obs = m/dA;
  gy_obs = dA;
end;

steady;
```

In particular, the statement `l = solve_from_equation('Intratemporal labour market condition');` is equivalent to writing `l = (psi/(1-psi))*(c*P/(1-n))*n;`.

### 4.10.3 Replace some equations during steady-state computations

When there is no steady-state file, Dynare computes the steady state by solving the static model, i.e. the model from the `.mod` file from which leads and lags have been removed.

In some specific cases, one may want to have more control over the way this static model is created. Dynare therefore offers the possibility to explicitly give the form of equations that should be in the static model.

More precisely, if an equation is prepended by a `[static]` tag, then it will appear in the static model used for steady-state computation, but that equation will not be used for other computations. For every equation tagged in

this way, you must tag another equation with `[dynamic]`: that equation will not be used for steady-state computation, but will be used for other computations.

This functionality can be useful on models with a unit root, where there is an infinity of steady states. An equation (tagged `[dynamic]`) would give the law of motion of the nonstationary variable (like a random walk). To pin down one specific steady state, an equation tagged `[static]` would affect a constant value to the nonstationary variable. Another situation where the `[static]` tag can be useful is when one has only a partial closed form solution for the steady state.

*Example*

This is a trivial example with two endogenous variables. The second equation takes a different form in the static model:

```
var c k;
varexo x;
...
model;
c + k - aa*x*k(-1)^alph - (1-delt)*k(-1);
[dynamic] c^(-gam) - (1+bet)^(-1)*(aa*alph*x(+1)*k^(alph-1) + 1 - delt)*c(+1)^(-gam);
[static] k = ((delt+bet)/(x*aa*alph))^(1/(alph-1));
end;
```

## 4.11 Getting information about the model

**Command:** **check** ;

**Command:** **check**(OPTIONS...);

Computes the eigenvalues of the model linearized around the values specified by the last `initval`, `endval` or *steady* statement. Generally, the eigenvalues are only meaningful if the linearization is done around a steady state of the model. It is a device for local analysis in the neighborhood of this steady state.

A necessary condition for the uniqueness of a stable equilibrium in the neighborhood of the steady state is that there are as many eigenvalues larger than one in modulus as there are forward-looking variables in the system. An additional rank condition requires that the square submatrix of the right Schur vectors corresponding to the forward looking variables (jumpers) and to the explosive eigenvalues must have full rank.

Note that the outcome may be different from what would be suggested by `sum(abs(oo_.dr.eigval))` when eigenvalues are very close to *qz_criterium*.

*Options*

**solve_algo = INTEGER**

See *solve_algo*, for the possible values and their meaning.

**qz_zero_threshold = DOUBLE**

Value used to test if a generalized eigenvalue is $0/0$ in the generalized Schur decomposition (in which case the model does not admit a unique solution). Default: `1e-6`.

*Output*

`check` returns the eigenvalues in the global variable `oo_.dr.eigval`.

**MATLAB/Octave variable:** **oo_.dr.eigval**

Contains the eigenvalues of the model, as computed by the `check` command.

**Command:** **model_diagnostics** ;

This command performs various sanity checks on the model, and prints a message if a problem is detected (missing variables at current period, invalid steady state, singular Jacobian of static model).

**Command:** **model_info** ;

**Command:** **model_info**(OPTIONS...);

This command provides information about the model. By default, it will provide a list of predetermined state variables, forward-looking variables, and purely static variables.

The command also allows to display information on the dynamic and static versions of the block decomposition of the model:

- The normalization of the model: an endogenous variable is attributed to each equation of the model (the dependent variable);
- The block structure of the model: for each block `model_info` indicates its type, size as well as the equation number(s) or name tags and endogenous variables belonging to this block.

There are five different types of blocks depending on the simulation method used:

- `EVALUATE FORWARD`

  In this case the block contains only equations where the dependent variable $j$ attributed to the equation appears contemporaneously on the left hand side and where no forward looking endogenous variables appear. The block has the form: $y_{j,t} = f_j(y_t, y_{t-1}, \ldots, y_{t-k})$.

- `EVALUATE BACKWARD`

  The block contains only equations where the dependent variable $j$ attributed to the equation appears contemporaneously on the left hand side and where no backward looking endogenous variables appear. The block has the form: $y_{j,t} = f_j(y_t, y_{t+1}, \ldots, y_{t+k})$.

- `SOLVE BACKWARD x`

  The block contains only equations where the dependent variable $j$ attributed to the equation does not appear contemporaneously on the left hand side and where no forward looking endogenous variables appear. The block has the form: $g_j(y_{j,t}, y_t, y_{t-1}, \ldots, y_{t-k}) = 0$. Here, `x` denotes the subtype of the block. `x` is equal to `SIMPLE` if the block has only one equation. If several equations appear in the block, `x` is equal to `COMPLETE`.

- `SOLVE FORWARD x`

  The block contains only equations where the dependent variable $j$ attributed to the equation does not appear contemporaneously on the left hand side and where no backward looking endogenous variables appear. The block has the form: $g_j(y_{j,t}, y_t, y_{t+1}, \ldots, y_{t+k}) = 0$. Here, `x` denotes the subtype of the block. `x` is equal to `SIMPLE` if the block has only one equation. If several equations appear in the block, `x` is equal to `COMPLETE`.

- `SOLVE TWO BOUNDARIES x`

  The block contains equations depending on both forward and backward variables. The block looks like: $g_j(y_{j,t}, y_t, y_{t-1}, \ldots, y_{t-k}, y_t, y_{t+1}, \ldots, y_{t+k}) = 0$. Here, `x` denotes the subtype of the block. `x` is equal to `SIMPLE` if the block has only one equation. If several equations appear in the block, `x` is equal to `COMPLETE`.

*Options*

**block_static**

Prints out the block decomposition of the static model.

**block_dynamic**

Prints out the block decomposition of the dynamic model.

**incidence**

Displays the gross incidence matrix and the reordered incidence matrix of the block decomposed model for the `block_dynamic` or `block_static` options. This option requires one of `block_dynamic` or `block_static`.

**Command:** **print_bytecode_dynamic_model ;**

Prints the equations and the Jacobian matrix of the dynamic model stored in the bytecode binary format file. Can only be used in conjunction with the `bytecode` option of the *model* block or *model_options* command.

**Command:** **print_bytecode_static_model ;**

Prints the equations and the Jacobian matrix of the static model stored in the bytecode binary format file. Can only be used in conjunction with the `bytecode` option of the *model* block or *model_options* command.

## 4.12 Deterministic simulation

### 4.12.1 Perfect foresight

When the framework is deterministic, Dynare can be used for models with the assumption of perfect foresight. Typically, the system is supposed to be in a state of equilibrium before a period 1 when the news of a contemporaneous or of a future shock is learned by the agents in the model. The purpose of the simulation is to describe the reaction in anticipation of, then in reaction to the shock, until the system returns to the old or to a new state of equilibrium. In most models, this return to equilibrium is only an asymptotic phenomenon, which one must approximate by an horizon of simulation far enough in the future. Another exercise for which Dynare is well suited is to study the transition path to a new equilibrium following a permanent shock. For deterministic simulations, the numerical problem consists of solving a nonlinear system of simultaneous equations in `n` endogenous variables in `T` periods. Dynare offers several algorithms for solving this problem, which can be chosen via the `stack_solve_algo` option. By default (`stack_solve_algo=0`), Dynare uses a Newton-type method to solve the simultaneous equation system. Because the resulting Jacobian is in the order of `n` by `T` and hence will be very large for long simulations with many variables, Dynare makes use of the sparse matrix capacities of MATLAB/Octave. A slower but potentially less memory consuming alternative (`stack_solve_algo=1`) is based on a Newton-type algorithm first proposed by Laffargue (1990) and Boucekkine (1995), which avoids ever storing the full Jacobian. The details of the algorithm can be found in Juillard (1996). The third type of algorithms makes use of block decomposition techniques (divide-and-conquer methods) that exploit the structure of the model. The principle is to identify recursive and simultaneous blocks in the model structure and use this information to aid the solution process. These solution algorithms can provide a significant speed-up on large models.

**Warning**

Be careful when employing auxiliary variables in the context of perfect foresight computations. The same model may work for stochastic simulations, but fail for perfect foresight simulations. The issue arises when an equation suddenly only contains variables dated `t+1` (or `t-1` for that matter). In this case, the derivative in the last (first) period with respect to all variables will be 0, rendering the stacked Jacobian singular.

*Example*

Consider the following specification of an Euler equation with log utility:

```
Lambda = beta*C(-1)/C;
Lambda(+1)*R(+1)= 1;
```

Clearly, the derivative of the second equation with respect to all endogenous variables at time `t` is zero, causing *perfect_foresight_solver* to generally fail. This is due to the use of the Lagrange multiplier `Lambda` as an auxiliary variable. Instead, employing the identical

```
beta*C/C(+1)*R(+1)= 1;
```

will work.

**Command:** **perfect_foresight_setup ;**

**Command:** **perfect_foresight_setup**`(OPTIONS...);`

Prepares a perfect foresight simulation, by extracting the information in the *initval*, *endval*, *shocks*, *mshocks*, *perfect_foresight_controlled_paths* and *shock_paths* blocks and converting them into simulation paths for exogenous and endogenous variables.

This command must always be called before running the simulation with *perfect_foresight_solver*.

*Options*

**periods = INTEGER**

Number of periods of the simulation. This option is mandatory, unless both `first_simulation_period` and `last_simulation_period` options are given.

**first_simulation_period = DATE**

Assign a date to the first simulation period, i.e. the first period in which endogenous variables are solved for. When this option is set, it becomes possible to declare shocks using dates, and *perfect_foresight_solver* returns the result of the simulation as a time series object in *Simulated_time_series*.

**last_simulation_period = DATE**

Assign a date to the last simulation period, i.e. the last period in which endogenous variables are solved for. When this option is set, it becomes possible to declare shocks using dates, and *perfect_foresight_solver* returns the result of the simulation as a time series object in *Simulated_time_series*.

**datafile = FILENAME**

Used to specify path for all endogenous and exogenous variables. Strictly equivalent to *initval_file*.

**endval_steady**

In scenarios with a permanent shock, specifies that the terminal condition is a steady state, even if the *steady* command has not been called after the `endval` block. As a consequence, the subsequent *perfect_foresight_solver* command will compute the terminal steady state itself (given the value of the exogenous variables given in the `endval` block). In practice, this option is useful when the permanent shock is very large, in which case the homotopy procedure inside *perfect_foresight_solver* will find both the terminal steady state and the transitional dynamics within the same loop (which is less costly than first computing the terminal steady state by homotopy, then computing the transitional dynamics by homotopy). Note that this option is implicitly enabled if the terminal condition is already a steady state at the point of execution of the *perfect_foresight_solver* command. It is also implicitly enabled if shocks are described using the *shock_paths* block.

**endval_steady_nocheck**

Similar to the *nocheck* option applied to the terminal steady state computation when used in conjuction with the `endval_steady` option.

*Output*

The paths for the exogenous variables are stored into `oo_.exo_simul`.

The initial and terminal conditions for the endogenous variables and the initial guess for the path of endogenous variables are stored into `oo_.endo_simul`.

**Command: perfect_foresight_solver ;**

**Command: perfect_foresight_solver**(OPTIONS...);

Computes the perfect foresight (or deterministic) simulation of the model.

Note that *perfect_foresight_setup* must be called before this command, in order to setup the environment for the simulation.

If the perfect foresight solver cannot directly find the solution of the problem, it subsequently tries a homotopy technique (unless the `no_homotopy` option is given). Concretely, this technique consists in dividing the problem into smaller steps by diminishing the size of shocks and increasing them progressively until the problem converges.

*Options*

**maxit = INTEGER**

Determines the maximum number of iterations used in the non-linear solver. The default value of `maxit` is `50`.

**tolf = DOUBLE**

Convergence criterion for termination based on the function value. Iteration will cease when it proves impossible to improve the function value by more than `tolf`. Default: `1e-5`

**tolx = DOUBLE**

Convergence criterion for termination based on the change in the function argument. Iteration will cease when the solver attempts to take a step that is smaller than `tolx`. Default: `1e-5`

**noprint**

See *noprint*.

**print**

Print results (opposite of `noprint`).

**stack_solve_algo = INTEGER**

Algorithm used for computing the solution. Possible values are:

`0`

Use a Newton algorithm with a direct sparse LU solver at each iteration, applied to the stacked system of all equations in all periods. The direct sparse LU solver is the `mldivide` MATLAB/Octave function (unless the *bytecode* option is used, in which case the sparse LU solver is UMFPACK from SuiteSparse). Termination criterion: `tolf`. (Default)

`1`

Use the Laffargue-Boucekkine-Juillard (LBJ) algorithm proposed in Juillard (1996) on top of a LU solver. It is slower than `stack_solve_algo=0`, but may be less memory consuming on big models. Note that if the `block` option is used (see *Model declaration*), a simple Newton algorithm with sparse matrices, applied to the stacked system of all block equations in all periods, is used for blocks which are purely backward or forward (of type `SOLVE BACKWARD` or `SOLVE FORWARD`, see *model_info*), since LBJ only makes sense on blocks with both leads and lags (of type `SOLVE TWO BOUNDARIES`). Termination criterion: `tolx`.

`2`

Use a Newton algorithm with a Generalized Minimal Residual (GMRES) solver at each iteration, applied on the stacked system of all equations in all periods. The following options can be used to control the behaviour of the algorithm: *preconditioner*, *iter_tol*, *iter_maxit*, *gmres_restart*. Termination criterion: `tolf`.

`3`

Use a Newton algorithm with a Biconjugate Gradient Stabilized (BiCGStab) solver at each iteration, applied on the stacked system of all equations in all periods. The following options can be used to control the behaviour of the algorithm: *preconditioner*, *iter_tol*, *iter_maxit*. Termination criterion: `tolf`.

`4`

Use a Newton algorithm with a direct sparse LU solver and an optimal path length at each iteration, applied on the stacked system of all equations in all periods (requires `bytecode` and/or `block` option, see *Model declaration*).

`5`

Use the Laffargue-Boucekkine-Juillard (LBJ) algorithm proposed in Juillard (1996) on top of a sparse Gaussian elimination (SPE) solver. The latter takes advantage of the similarity of the Jacobian across periods when searching for the pivots. This algorithm requires the *bytecode* option. The following options can be used to control the behaviour of the algorithm: *markowitz*, *minimal_solving_periods*.

`6`

Synonymous for `stack_solve_algo=1`. Kept for backward compatibility.

`7`

Allows the user to solve the perfect foresight model with the solvers available through option `solve_algo`, applied on the stacked system of all equations in all periods (See *solve_algo* for a list of possible values, note that values 5, 6, 7 and 8, which require `bytecode` and/or `block` options, are not allowed). For instance, the following commands:

```
perfect_foresight_setup(periods=400);
perfect_foresight_solver(stack_solve_algo=7, solve_algo=9)
```

trigger the computation of the solution with a trust region algorithm.

8

Use a Newton algorithm with a direct sparse LU solver at each iteration, applied to the stacked system of all equations in all periods. The sparse LU solver is ParU from SuiteSparse. Termination criterion: `tolf`. Not available with the *bytecode* option.

9

Use a Newton algorithm with a direct sparse LU solver at each iteration, applied to the stacked system of all equations in all periods. The direct sparse LU solver is PARDISO. Termination criterion: `tolf`. See *Optional installation of Panua PARDISO* for PARDISO installation instructions. Not available with the *bytecode* option.

10

Use a Newton algorithm applied on the stacked system of all equations in all periods. The direct linear solver in PARDISO is used to compute the solution in the first iteration; in further iterations, use a Conjugate Gradient Squared (CGS) iterative linear solver as implemented in PARDISO, with the LU decomposition from the first iteration as preconditioner. The *iter_tol* option can be used to control the termination criterion of the iterative linear solver. Termination criterion: `tolf`. See *Optional installation of Panua PARDISO* for PARDISO installation instructions. Not available with the *bytecode* option.

**preconditioner = OPTION**

When *stack_solve_algo* is equal to 2 or 3, this option specifies which preconditioner will be used in combination with the iterative sparse linear solver (either GMRES or BiCGStab). Possible values for OPTION are:

`first_iter_lu`

At the first iteration of the nonlinear Newton solver, compute the full LU decomposition with complete pivoting of the linear system (and use it to solve that first iteration rather than using the iterative solver). This LU decomposition is then used as the preconditioner in further Newton iterations.

`block_diagonal_lu`

Compute the LU decomposition with complete pivoting for only a few simulation periods within the stacked Jacobian (which is a block tridiagonal matrix). Then repeat that LU decomposition over the block diagonal to construct a preconditioner for the linear system with all simulation periods. If the total number of simulations periods is not a multiple of the number of periods used for the small LU, then an additional LU is computed for the remainder. The following options can be used to control the construction of this preconditioner: *block_diagonal_lu_maxlu*, *block_diagonal_lu_nperiods*, *block_diagonal_lu_nlu*, *block_diagonal_lu_relu*.

`incomplete_lu`

Use an incomple LU decomposition as the preconditioner, recomputed at every iteration of the nonlinear Newton solver.

Default value is `first_iter_lu`.

**iter_tol = DOUBLE**

When *stack_solve_algo* is equal to 2, 3 or 10, this option controls the relative tolerance of the iterative linear solver (either GMRES, BiCGStab or CGS). It corresponds to the `tol` option of the `gmres` and `bicgstab` MATLAB/Octave functions when `stack_solve_algo` is equal to 2 or 3, or to the stopping criterion of the Krylov-Subspace iteration as documented in `IPARM(4)` of the PARDISO routine. Note that the perfect foresight solver uses an *absolute* tolerance for determining convergence, so this option should be used with care, and the default is meant to suit most situations. Default: the value of the *tolf* option, divided by 10 times the infinite norm of the right-hand side of the linear system.

**iter_maxit = INTEGER**

When *stack_solve_algo* is equal to 2 or 3, this option controls the maximum number of iterations of the iterative linear solver (either GMRES or BiCGStab). It corresponds to the `maxit` option of the `gmres` and `bicgstab` MATLAB/Octave functions. It should not be confused with the *maxit* option, which controls the (outer) nonlinear Newton loop, while `iter_maxit` controls the (inner) linear loop. Default: `500`.

**gmres_restart = INTEGER**

When *stack_solve_algo* is equal to 2, this option controls the number of iterations before restart of the GMRES algorithm. It corresponds to the `restart` option of the `gmres` MATLAB/Octave function. Default: `100`.

**block_diagonal_lu_maxlu = INTEGER**

When *preconditioner* is equal to `block_diagonal_lu`, controls the maximum size of the matrix for which the small LU will computed. The actual size of the matrix will be determined by the largest number of periods that, multiplied by the number of equations, is less than the value of the option. Note that when combined with block decomposition (see *block*), blocks for which the whole stacked system is less than this option will be solved using a regular LU decomposition instead of the iterative linear solver. Default: `20000`.

**block_diagonal_lu_nperiods = INTEGER**

When *preconditioner* is equal to `block_diagonal_lu`, controls the number of periods used for the small LU. If nonzero, this option overrides the *block_diagonal_lu_maxlu* and *block_diagonal_lu_nlu* options. Default: `0`.

**block_diagonal_lu_nlu = INTEGER**

When *preconditioner* is equal to `block_diagonal_lu`, specifies the number of times that the small LU should be repeated in the large preconditioner. If nonzero, this option overrides the *block_diagonal_lu_maxlu* option. Default: `0`.

**block_diagonal_lu_relu = DOUBLE**

When *preconditioner* is equal to `block_diagonal_lu`, controls the relative position of the small LU within the whole stacked system. Must be a number between `0` and `1`. Default: `0.5`.

**robust_lin_solve**

Triggers the use of a robust linear solver for the default `stack_solve_algo=0`.

**allow_nonfinite_values**

By default, Dynare sets all NaN and Inf encountered during iterations to 0 and tries to continue solving the model. This approach regularly prevents the usually terminal propation of non-finite values. The current option allows keeping non-finite values, which may be useful for debugging purposes.

**check_jacobian_singularity**

Triggers a check of the dynamic Jacobian for singularity during the first iteration. Useful for detecting pathologies in the model specification. Available only for `stack_solve_algo` equal to `0`, `2` or `3`. Neither compatible with the i) `block`, `bytecode`, and `linear` options of the `model` block or `model_options` command nor ii) the `linear_approximation` and `lmmcp` options, nor iii) with purely forward or backward models. Due to computational intensity and high memory requirements, it is strongly recommended to conduct this test with a small value for the `periods` option.

**solve_algo**

See *solve_algo*. Allows selecting the solver used with `stack_solve_algo=7`. Also used for purely backward, forward and static models (when neither the `block` nor the `bytecode` option of the *model* block or *model_options* command is specified); for those models, the values `12` and `14` are especially relevant.

**no_homotopy**

This option tells Dynare to not try a homotopy technique (as described above) if the problem cannot be solved directly.

**homotopy_initial_step_size = DOUBLE**

Specifies which share of the shock should be applied in the first iteration of the homotopy procedure. This option is useful when it is known that immediately trying 100% of the shock will fail, so as to save computing time. Must be between `0` and `1`. Default: `1`.

**homotopy_min_step_size = DOUBLE**

The homotopy procedure halves the size of the step whenever there is a failure. This option specifies the minimum step size under which the homotopy procedure is considered to have failed. Default: `0.001`.

**homotopy_step_size_increase_success_count = INTEGER**

Specifies after how many consecutive successful iterations the homotopy procedure should double the size of the step. A zero value means that the step size should never be increased. Default: `3`.

**homotopy_linearization_fallback**

Whenever the homotopy procedure is not able to find a solution for 100% of the shock, but is able to find one for a smaller share, instructs Dynare to compute an approximate solution by rescaling the solution obtained for a fraction of the shock, as if the reaction of the model to the shock was a linear function of the size of that shock. More formally, if $s$ is the share of the shock applied (between 0 and 1), $y(s)$ is the value of a given endogenous variable at a given period as a function of $s$ (in particular, $y(1)$ corresponds to the exact solution of the problem), and $s^*$ is the greatest share of the shock for which the homotopy procedure has been able to find a solution, then the approximate solution returned is $\frac{y(s^*)-y(0)}{s^*}$.

If linearization is triggered, the variable *oo_.deterministic_simulation.homotopy_linearization* is set, and the simulation corresponding to share $s^*$ is stored in *oo_.deterministic_simulation.sim1*.

**homotopy_marginal_linearization_fallback [= DOUBLE]**

Whenever the homotopy procedure is not able to find a solution for 100% of the shock, but is able to find one for a smaller share, instructs Dynare to compute an approximate solution obtained by rescaling the solution obtained for a fraction of the shock, obtained as if the reaction of the model to the shock was, at the margin, a linear function of the size of that shock. More formally, if $s$ is the share of the shock applied (between 0 and 1), $y(s)$ is the value of a given endogenous variable at a given period as a function of $s$ (in particular, $y(1)$ corresponds to the exact solution of the problem), $s^*$ is the greatest share of the shock for which the homotopy procedure has been able to find a solution, and $\epsilon$ is a small step size, then the approximate solution returned is $y(s^*) + (1 - s^*)\frac{y(s^*)-y(s^*-\epsilon)}{\epsilon}$. The value of $\epsilon$ is `0.01` by default, but can be modified by passing some other value to the option.

If marginal linearization is triggered, the variable *oo_.deterministic_simulation.homotopy_marginal_linearization* is set. Moreover, the simulation corresponding to share $s^*$ is stored in *oo_.deterministic_simulation.sim1*, and the one corresponding to share $s^* - \epsilon$ is stored in *oo_.deterministic_simulation.sim2*.

**homotopy_max_completion_share = DOUBLE**

Instructs Dynare, within the homotopy procedure, to not try to compute the solution for a greater share than the one given as the option value. This option only makes sense when used in conjunction with either the `homotopy_linearization_fallback` or the `homotopy_marginal_linearization_fallback` option. It is typically used in situations where it is known that homotopy will fail to go beyond a certain point, so as to save computing time, while at the same time getting an approximate solution. Default: `1`.

**homotopy_exclude_varexo = (VARIABLE_NAME...)**

A list of exogenous variables which are to be excluded from the homotopy procedure, *i.e.* which must be kept at their value corresponding to 100% of the shock during all homotopy iterations.

**markowitz = DOUBLE**

Value of the Markowitz criterion, used to select the pivot (see *markowitz* for more details). Only used when `stack_solve_algo = 5`. Default: `0.5`.

**minimal_solving_periods = INTEGER**

Specify the minimal number of periods where the model has to be solved, before using a constant set of operations for the remaining periods. Only used when `stack_solve_algo = 5`. Default: `1`.

**lmmcp**

Solves the perfect foresight model with a Levenberg-Marquardt mixed complementarity problem (LMMCP) solver (Kanzow and Petra, 2004), which allows to consider inequality constraints on the endogenous variables (such as a zero lower bound, henceforth ZLB, on the nominal interest rate or a model with irreversible investment). This option is equivalent to `stack_solve_algo=7` **and** `solve_algo=10`. Using the LMMCP solver avoids the need for min/max operators and explicit complementary slackness conditions in the model as they will typically introduce a singularity into the Jacobian. This is done by setting the problem up as a mixed complementarity problem (MCP) of the form:

$$LB = X \Rightarrow F(X) > 0$$

$$LB < X < UB \Rightarrow F(X) = 0$$

$$X = UB \Rightarrow F(X) < 0.$$

where $X$ denotes the vector of endogenous variables, $F(X)$ the equations of the model, $LB$ denotes a lower bound, and $UB$ an upper bound. Such a setup is implemented by specifying the complementarity condition after the equation to which it is attached, the two being separated by the perpendicular symbol (the latter can be input either in UTF-8, as ⊥, corresponding to Unicode codepoint U+27C2; or alternatively as pure ASCII, as _|_, *i.e.* a vertical bar enclosed within two underscores).

For instance, a ZLB on the nominal interest rate would be specified as follows in the model block:

```
model;
   ...
   r = rho*r(-1) + (1-rho)*(gpi*Infl+gy*YGap) + e  ⊥  r > -1.94478;
   ...
end;
```

where `1.94478` is the steady-state level of the nominal interest rate and `r` is the nominal interest rate in deviation from the steady state. This construct implies that the Taylor rule is operative, unless the implied interest rate `r<=-1.94478`, in which case the `r` is fixed at `-1.94478` (thereby being equivalent to a complementary slackness condition). By restricting the value of `r` coming out of this equation, the complementarity condition also avoids using `max(r,-1.94478)` for other occurrences of `r` in the rest of the model. Two things are important to keep in mind. First, the complementary slackness condition cannot be simply attached to any equation; it must be attached to the correct affected equation as otherwise the solver will solve a different problem than originally intended. Second, the sign of the residual of the dynamic equation must conform to the MCP setup outlined above. In case of the ZLB, we are dealing with a lower bound. Consequently, the dynamic equation needs to return a positive residual. Dynare by default computes the residual of an equation `LHS=RHS` as `residual=LHS-RHS`, while an implicit equation `LHS` is interpreted as `LHS=0`. For the above equation this implies

```
residual= r - (rho*r(-1) + (1-rho)*(gpi*Infl+gy*YGap) + e);
```

which is correct, since it will be positive if the implied interest rate `rho*r(-1) + (1-rho)*(gpi*Infl+gy*YGap) + e` is below `r=-1.94478`. In contrast, specifying the equation as

```
rho*r(-1) + (1-rho)*(gpi*Infl+gy*YGap) + e = r;
```

would be wrong.

Note that both the lower and the upper bounds can be specified at the same time in a given complementarity condition. Moreover, arbitrary functions of parameters can appear in the bounds. As an example, the following complementarity condition is syntactically correct (assuming that `alpha` is a parameter):

```
... ⊥ -1.94478 < r < 1+2*alpha;
```

**endogenous_terminal_period**

The number of periods is not constant across Newton iterations when solving the perfect foresight model. The size of the nonlinear system of equations is reduced by removing the portion of the paths (and associated equations) for which the solution has already been identified (up to the tolerance parameter). This strategy can be interpreted as a mix of the shooting and relaxation approaches. Note that round off errors are more important with this mixed strategy (user should check the reported value of the maximum absolute error). Only available with `stack_solve_algo` option equal to `0`.

**linear_approximation**

Solves the linearized version of the perfect foresight model. The model must be stationary and a steady state needs to be provided. Linearization is conducted about the last defined steady state, which can derive from `initval`, `endval` or a subsequent *steady*. Only available with `stack_solve_algo` option equal to `0` or `7`.

**steady_solve_algo = INTEGER**

See *solve_algo*. Used when computing the terminal steady state when option `endval_steady` has been specified to the *perfect_foresight_setup* command.

**steady_tolf = DOUBLE**

See *tolf*. Used when computing the terminal steady state when option `endval_steady` has been specified to the *perfect_foresight_setup* command.

**steady_tolx = DOUBLE**

See *tolx*. Used when computing the terminal steady state when option `endval_steady` has been specified to the *perfect_foresight_setup* command.

**steady_maxit = INTEGER**

See *maxit*. Used when computing the terminal steady state when option `endval_steady` has been specified to the *perfect_foresight_setup* command.

**steady_markowitz = DOUBLE**

See *markowitz*. Used when computing the terminal steady state when option `endval_steady` has been specified to the *perfect_foresight_setup* command.

*Output*

The simulated endogenous variables are available in global matrix *oo_.endo_simul*.

If any of the `first_simulation_period` or `last_simulation_period` option was passed to the preceding *perfect_foresight_setup* command, a time series object containing both endogenous and exogenous variables is stored in the workspace variable *Simulated_time_series*.

The variable *oo_.deterministic_simulation.status* indicates whether the simulation was successful or not.

**Command: simul ;**

**Command: simul**(OPTIONS...);

This command is deprecated. It is strictly equivalent to a call to *perfect_foresight_setup* followed by a call to *perfect_foresight_solver*.

*Options*

Accepts all the options of *perfect_foresight_setup* and *perfect_foresight_solver*.

**MATLAB/Octave variable: oo_.endo_simul**

This variable stores the result of a deterministic simulation (computed by *perfect_foresight_solver* or *perfect_foresight_with_expectation_errors_solver*) or of a stochastic simulation (computed by *stoch_simul* with the periods option or by *extended_path*). The variables are arranged row by row, in order of declaration (as in M_.endo_names). Note that this variable also contains initial and terminal conditions, so it has more columns than the value of the periods option: the first simulation period is in column 1+M_.maximum_lag, and the total number of columns is M_.maximum_lag+periods+M_.maximum_lead.

**MATLAB/Octave variable: oo_.exo_simul**

This variable stores the path of exogenous variables during a simulation (computed by *perfect_foresight_solver*, *perfect_foresight_with_expectation_errors_solver*, *stoch_simul* or *extended_path*). The variables are arranged in columns, in order of declaration (as in M_.exo_names). Periods are in rows. Note that this convention regarding columns and rows is the opposite of the convention for oo_.endo_simul! Also note that this variable also contains initial and terminal conditions, so it has more rows than the value of the periods option: the first simulation period is in row 1+M_.maximum_lag, and the total number of rows is M_.maximum_lag+periods+M_.maximum_lead.

**MATLAB/Octave variable: Simulated_time_series**

This variable stores, as a *dseries* object, the path of both endogenous and exogenous variables after a deterministic simulation (computed by *perfect_foresight_solver* or *perfect_foresight_with_expectation_errors_solver* using the first_simulation_period and/or last_simulation_period option).

**MATLAB/Octave variable: oo_.initial_steady_state**

If a permanent shock is simulated through the use of both initval and endval blocks, this variable contains the initial steady state, as determined by the initval block (when followed by a *steady* command). This variable has the same structure as *oo_.steady_state* (and this latter variable contains the terminal steady state, if the endval block is followed by a *steady* command).

**MATLAB/Octave variable: oo_.initial_exo_steady_state**

If a permanent shock is simulated through the use of both initval and endval blocks, this variable contains the initial steady state of the exogenous variables, as specified in the initval block. This variable has the same structure as *oo_.exo_steady_state* (and this latter variable contains the terminal steady state of the exogenous variables).

**MATLAB/Octave variable: M_.maximum_lag**

The maximum number of lags in the model. Note that this value is computed on the model *after* the transformations related to auxiliary variables, so in practice it is either 1 or 0 (the latter value corresponds to a purely forward or static model).

**MATLAB/Octave variable: M_.maximum_lead**

The maximum number of leads in the model. Note that this value is computed on the model *after* the transformations related to auxiliary variables, so in practice it is either 1 or 0 (the latter value corresponds to a purely backward or static model).

**MATLAB/Octave variable: oo_.deterministic_simulation.status**

Set to true by the *perfect_foresight_solver* command if the simulation succeeded, otherwise set to false.

**MATLAB/Octave variable: oo_.deterministic_simulation.homotopy_linearization**

Set to true by the *perfect_foresight_solver* command if linearization has been used to compute an approximate solution.

**MATLAB/Octave variable: oo_.deterministic_simulation.homotopy_marginal_linearization**

Set to true by the *perfect_foresight_solver* command if marginal linearization has been used to compute an approximate solution.

**MATLAB/Octave variable: oo_.deterministic_simulation.sim1**

Set by the *perfect_foresight_solver* command if either linearization or marginal linearization has

been used to compute an approximate solution. This structure contains the simulation corresponding to the greatest share of the shocks for which an exact solution could be computed. The subfield `homotopy_completion_share` contains that share. The subfields `endo_simul`, `exo_simul`, `steady_state` and `exo_steady_state` respectively contain the path of endogenous, the path of exogenous, the steady state of endogenous and the steady state of exogenous for that simulation (with the same conventions as the fields of of the same name in `oo_`).

**MATLAB/Octave variable: oo_.deterministic_simulation.sim2**

Set by the *perfect_foresight_solver* command if marginal linearization has been used to compute an approximate solution. This structure contains the simulation corresponding to a share marginally smaller that the one in *oo_.deterministic_simulation.sim1*. The subfields are the same as in *oo_.deterministic_simulation.sim1*.

### 4.12.2 Perfect foresight with expectation errors

The solution under perfect foresight that was presented in the previous section makes the assumption that agents learn the complete path of future shocks in period 1, without making any expectation errors.

One may however want to study a scenario where it turns out that agents make expectation errors, in the sense that the path they had anticipated in period 1 does not realize exactly. More precisely, in some simulation periods, they may receive new information that makes them revise their anticipation for the path of future shocks. Also, under this scenario, it is assumed that agents behave as under perfect foresight, *i.e.* they take their decisions as if there was no uncertainty and they knew exactly the path of future shocks; the new information that they may receive comes as a total surprise to them.

Such a scenario can be solved by Dynare using the *perfect_foresight_with_expectation_errors_setup* and *perfect_foresight_with_expectation_errors_solver* commands, alongside `shocks` and `endval` blocks which are given a special `learnt_in` option.

**Block: shocks**(learnt_in=INTEGER|DATE) ;

**Block: shocks**(learnt_in=INTEGER|DATE, overwrite) ;

The `shocks(learnt_in=INTEGER|DATE)` syntax can be used to specify temporary shocks that are learnt in a specific period. It should contain one or more occurrences of the following group of three lines, with the same semantics as a regular *shocks* block:

```
var VARIABLE_NAME;
periods INTEGER[:INTEGER] | DATE[:DATE] [[,] INTEGER[:INTEGER] | DATE[:DATE]]...;
values DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
```

If the period in which information is learnt is greater or equal than 2, then it is possible to specify the shock values in deviation with respect to the values that were expected from the perspective of the previous period. If the new information consists of an addition to the previously-anticipated value, the `values` keyword can be replaced by the `add` keyword; similarly, if the new information consists of a multiplication of the previously-anticipated value, the `values` keyword can be replaced by the `multiply` keyword.

The `overwrite` option says that this block cancels and replaces previous `shocks` and `mshocks` blocks that have the same `learnt_in` option. Note that a block with an integer-valued `learnt_in` option never overwrites a block with a date-valued `learnt_in` option, even if they correspond to the same period.

Also note that a `shocks(learnt_in=1)` block is equivalent to a regular *shocks* block.

*Example*

```
shocks(learnt_in=1);
  var x;
  periods 1:2 3:4 5;
  values 1 1.2 1.4;
end;

shocks(learnt_in=2);
  var x;
```

(continues on next page)

(continued from previous page)

```
  periods 3:4;
  add 0.1;
end;

shocks(learnt_in=4);
  var x;
  periods 5;
  multiply 2;
end;
```

This syntax means that:

- from the perspective of period 1, `x` is expected to be equal to 1 in periods 1 and 2, to 1.2 in periods 3 and 4, and to 1.4 in period 5;
- from the perspective of periods 2 (and 3), `x` is expected to be equal to 1 in period 2, to 1.3 in periods 3 and 4, and to 1.4 in period 5;
- from the perspective of periods 4 (and following), `x` is expected to be equal to 1.3 in period 4, and to 2.8 in period 5.

*Example* (with dates)

```
shocks(learnt_in=2023Q1);
  var x;
  periods 2023Q1:2023Q2 2023Q3:2023Q4 2024Q1;
  values 1 1.2 1.4;
end;
```

**Block:** **endval**(learnt_in=INTEGER|DATE) ;

The `endval(learnt_in=INTEGER|DATE)` can be used to specify terminal conditions that are learnt in a specific period.

Note that an `endval(learnt_in=1)` block is equivalent to a regular *endval* block.

Also note that, similarly to the regular *endval* block, any variable specified in this block will jump to its new value in the same period as the one in which the information is learnt; and, from the perspective of that period, the variable is expected by agents to remain to that value until the end of the simulation. In particular, this means that any temporary shock that may have been anticipated on that variable (as specified through a *shocks(learnt_in=INTEGER|DATE)* block for a previous informational period) will be overridden; if this is not the desired behaviour, then the temporary shock will have to be reinstated through another *shocks(learnt_in=INTEGER|DATE)* block.

It is possible to express the terminal condition by specifying the level of the exogenous variable (using an equal symbol, as in a regular *endval* blocks without the `learnt_in` option). But it is also possible to express the terminal condition as an addition to the value expected from the perspective of the previous period (using the += operator), or as a multiplicative factor over that previously expected value (using the *= operator).

*Example*

```
endval(learnt_in = 3);
  x = 1.1;
  y += 0.1;
  z *= 2;
end;
```

This syntax means that, in period 3, the agents learn that:

- the terminal condition for `x` will be 1.1;
- the terminal condition for `y` will be 0.1 above the terminal condition for `y` that was expected from the perspective of period 2;

- the terminal condition for `z` will be 2 times the terminal condition for `z` that was expected from the perspective of period 2.

Those values will be the realized ones, unless there is another `endval(learnt_in=p)` block with `p>3`.

The three variables will jump to their new value in period 3 and, from the perspective of period 3, they are expected by agents to remain there until the end of the simulation. In particular, any temporary shock on either `x`, `y` or `z` specified through a regular `shocks` block or through a `shocks(learnt_in=2)` block will be overridden. If this is not the desired behaviour, a `shocks(learnt_in=3)` block will have to be added to reinstate the temporary shock.

*Example* (with a date)

```
endval(learnt_in = 2024Q1);
  x = 1.1;
end;
```

**Block:** **mshocks**(learnt_in=INTEGER|DATE) ;

**Block:** **mshocks**(learnt_in=INTEGER|DATE,OPTIONS...) ;

The `mshocks(learnt_in=INTEGER|DATE)` syntax can be used to specify temporary shocks that are learnt in a specific period, specified in a multiplicative way. It should contain one or more occurrences of the following group of three lines, with the same semantics as a regular *mshocks* block:

```
var VARIABLE_NAME;
periods INTEGER[:INTEGER] | DATE[:DATE] [[,] INTEGER[:INTEGER] | DATE[:DATE]]...;
values DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
```

As in the regular *mshocks* block (without the `learnt_in` option), the values are interpreted as a multiplicative factor over the steady-state value of the exogenous variable (the latter being taken either from the `initval` or `endval`, see *mshocks* for the details).

If the terminal steady state as specified in the `endval` block is used as a basis for the multiplication, its value as anticipated from the period given in the `learnt_in` option will be used.

Note that a `mshocks(learnt_in=1)` block is equivalent to a regular *mshocks* block.

*Options*

**overwrite**

This block cancels and replaces previous `shocks` and `mshocks` blocks that have the same `learnt_in` option. Note that a block with an integer-valued `learnt_in` option never overwrites a block with a date-valued `learnt_in` option, even if they correspond to the same period.

**relative_to_initval**

Same meaning as in the regular *mshocks* block.

*Example*

```
mshocks(learnt_in=2);
  var x;
  periods 3:4;
  values 1.1;
end;
```

This syntax means that from the perspective of period 2, `x` in periods 3 and 4 is expected to be equal to 1.1 times its steady state. If there is no *endval* block, the initial steady state as given by *initval* is used; if there is an *endval* block, the terminal steady state as anticipated from the perspective of period 2 is used (as specified in the relevant *endval(learnt_in=INTEGER|DATE)* block).

*Example* (with dates)

```
mshocks(learnt_in=2024Q2);
  var x;
  periods 2024Q3:2024Q4;
  values 1.1;
end;
```

**Command:** **perfect_foresight_with_expectation_errors_setup ;**

**Command:** **perfect_foresight_with_expectation_errors_setup**(OPTIONS...);

Prepares a perfect foresight simulation with expectation errors, by extracting the contents of the *initval*, *endval*, *shocks*, *mshocks*, *perfect_foresight_controlled_paths* and *shock_paths* blocks (the latter five types of blocks typically used with the `learnt_in` option); alternatively, the information about future shocks can be given in a CSV file using the `datafile` option.

This command must always be called before running the simulation with *perfect_foresight_with_expectation_errors_solver*.

Note that this command makes the assumption that the terminal condition is always a steady state. Hence, it will recompute the terminal steady state as many times as the anticipation about the terminal condition changes. In particular, the information about endogenous variables that may be given in the `endval` block is ignored. Said otherwise, the equivalent of option `endval_steady` of the *perfect_foresight_setup* command is always implicitly enabled.

*Options*

**periods = INTEGER**

Number of periods of the simulation.

**first_simulation_period = DATE**

Same meaning as the eponymous option of *perfect_foresight_setup*.

**last_simulation_period = DATE**

Same meaning as the eponymous option of *perfect_foresight_setup*.

**datafile = FILENAME**

Used to specify the information about future shocks and their anticipation through a CSV file, as an alternative to `shocks` and `endval` blocks.

The file has the following format:

- the first column is ignored (can be used to add descriptive labels)
- the first line contains names of exogenous variables
- the second line contains, in columns, indices of periods *at which* expectations are formed; the information set used in a given period is described by all the columns for which that line is equal to the period index
- the subsequent lines correspond to the periods *for which* expectations are formed, one period per line; each line gives the values of present and future exogenous variables, as seen from the period given in the second line
- the last line corresponds to the terminal condition for exogenous variables, as anticipated in the various informational periods

If `p` is the value of the `periods` option and `k` is the number of exogenous variables, then the CSV file has `p+3` lines and `k×p+1` columns.

Concretely, the value of a given exogenous in period `t`, as anticipated from period `s`, is given in line `t+2`, and in the column which has the name of the variable on the first line and `s` on the second line. Of course, values in cells corresponding to `t<s` are ignored.

**endval_steady_nocheck**

Similar to the *nocheck* option applied to the terminal steady state computation.

*Output*

`oo_.exo_simul` and `oo_.endo_simul` are initialized before the simulation. Temporary shocks are stored in `oo_.pfwee.shocks_info`, terminal conditions for exogenous variables are stored in `oo_.pfwee.terminal_info`.

*Example*

Here is a CSV file example that could be given to the `datafile` option (adding some extra padding space for clarity):

```
Exogenous      ,   x,   x,   x,   x,   x,   x,   x
Period   (info),   1,   2,   3,   4,   5,   6,   7
Period 1 (real), 1.2,    ,    ,    ,    ,    ,
Period 2 (real),   1, 1.3,    ,    ,    ,    ,
Period 3 (real),   1,   1, 1.4,    ,    ,    ,
Period 4 (real),   1,   1,   1,   1,    ,    ,
Period 5 (real),   1,   1,   1,   1,   1,    ,
Period 6 (real),   1,   1,   1,   1,   1, 1.1,
Period 7 (real),   1,   1,   1,   1,   1, 1.1, 1.1
Terminal (real),   1, 1.1, 1.2, 1.2, 1.2, 1.1, 1.1
```

In this example, there is only one exogenous variable (`x`), and 7 simulation periods. In the first period, agents learn a contemporary shock (1.2), but anticipate no further shock. In period 2, they learn an unexpected contemporary shock (1.3), and also a change in the terminal condition (1.1). In period 3 again there is an unexpected contemporary shock and a change in the terminal condition. No new information comes in period 4 and 5. In period 6, an unexpected permanent shock is learnt. No new information comes in period 7.

Alternatively, instead of using a CSV file, the same sequence of information sets could be described using the following blocks:

```
initval;
  x = 1;
end;

steady;

shocks(learnt_in = 1);
  var x;
  periods 1;
  values 1.2;
end;

shocks(learnt_in = 2);
  var x;
  periods 2;
  values 1.3;
end;

endval(learnt_in = 2);
  x = 1.1;
end;

shocks(learnt_in = 3);
  var x;
  periods 3;
  values 1.4;
end;

endval(learnt_in = 3);
```

(continues on next page)

(continued from previous page)

```
  x = 1.2;
end;

shocks(learnt_in = 6);
  var x;
  periods 6:7;
  values 1.1;
end;

endval(learnt_in = 6);
  x = 1.1;
end;
```

**Command:** **perfect_foresight_with_expectation_errors_solver ;**

**Command:** **perfect_foresight_with_expectation_errors_solver**(OPTIONS...);

Computes the perfect foresight simulation with expectation errors of the model.

Note that *perfect_foresight_with_expectation_errors_setup* must be called before this command, in order to set up the environment for the simulation.

*Options*

This command accepts all the options of *perfect_foresight_solver*, with the same semantics, plus the following one:

**constant_simulation_length**

By default, every time the information set changes, the simulation with the new information set is shorter than the previous one (because the terminal date is getting closer). When this option is set, every new simulation has the same length (as specified by the `periods` option of *perfect_foresight_with_expectation_errors_setup*); as a consequence, the simulated paths as stored in `oo_.endo_simul` will be longer when this option is set (if *s* is the last period in which the information set is modified, then they will contain *s+periods-1* periods, excluding initial and terminal conditions). Note that this option is not available if `last_simulation_period` option has been passed to *perfect_foresight_with_expectation_errors_setup*.

*Output*

The simulated paths of endogenous variables are available in `oo_.endo_simul`. The terminal steady-state values corresponding to the last period of the information set are available in `oo_.steady_state` and `oo_.exo_steady_state`.

If any of the `first_simulation_period` or `last_simulation_period` option was passed to the preceding *perfect_foresight_with_expectation_errors_setup* command, a time series object containing both endogenous and exogenous variables is stored in the workspace variable *Simulated_time_series*.

**MATLAB/Octave variable:** **oo_.pfwee.shocks_info**

This variable stores the temporary shocks used during perfect foresight simulations with expectation errors, after *perfect_foresight_with_expectation_errors_setup* has been run. It is a three-dimensional matrix: first dimension correspond to exogenous variables (in declaration order); second dimension corresponds to real time; third dimension corresponds to informational time. In other words, the value of exogenous indexed `k` in period `t`, as anticipated from period `s`, is stored in `oo_.pfwee.shocks_info(k,t,s)`.

**MATLAB/Octave variable:** **oo_.pfwee.terminal_info**

This variable stores the terminal conditions for exogenous variables used during perfect foresight simulations with expectation errors, after *perfect_foresight_with_expectation_errors_setup* has been run. It is a matrix, whose lines correspond to exogenous variables (in declaration order), and whose columns correspond to informational time. In other words, the terminal condition for exogenous indexed `k`, as anticipated from period `s`, is stored in `oo_.pfwee.terminal_info(k,s)`.

### 4.12.3 Controlling the path of endogenous variables

In the usual perfect foresight problem, the user controls the path of exogenous variables for the simulation periods and the initial and terminal conditions for endogenous variables, while Dynare solves for the path of endogenous variables for the simulation periods.

However, Dynare offers the possibility of controlling the value of some endogenous variables for some simulation periods (in which case some exogenous variables must be left free and are thus solved for by Dynare, to avoid over-determination of the problem). This exercise is called "conditional forecasting" in some contexts (even though one may argue that this is not really forecasting, since perfect foresight by the agents is assumed; for the stochastic case, see the *conditional_forecast* command).

The description of controlled endogenous variables is done using the `perfect_foresight_controlled_paths` block. The information given therein is then processed by the *perfect_foresight_setup* (or *perfect_foresight_with_expectation_errors_setup*) command, so that the next *perfect_foresight_solver* (or *perfect_foresight_with_expectation_errors_solver*) command computes the simulation with controlled paths. In particular, *oo_.exo_simul* will contain the computed value of exogenous variables that have been left free.

**Block: perfect_foresight_controlled_paths ;**

**Block: perfect_foresight_controlled_paths**(OPTIONS...);

This block is used to tell the perfect foresight solver that the value of some endogenous variables will be controlled (in other words, they will be exogenized). It also gives the period(s) for which this control applies, the value(s) imposed to the endogenous variable(s), and the exogenous variable(s) that are left free at the same period(s) (in other words, those exogenous are endogenized).

The block should contain one or more occurrences of the following group of four lines:

```
exogenize ENDOGENOUS_NAME;
periods INTEGER[:INTEGER] | DATE[:DATE] [[,] INTEGER[:INTEGER] | DATE[:DATE]]...;
values DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
endogenize EXOGENOUS_NAME;
```

Note that it is possible to have both `perfect_foresight_controlled_paths` and regular *shocks* blocks in the same `.mod` file (assuming of course that taken together they do not impose inconsistent constraints).

The `perfect_foresight_controlled_paths` block requires that the *stack_solve_algo* option be equal to either `0`, `1`, `2`, `3`, `6` or `7`, and is incompatible with the *block* and *bytecode* options of the *model* block and *model_options* command.

*Options*

**overwrite**

This block cancels and replaces previous `perfect_foresight_controlled_paths` blocks that have the same `learnt_in` option. Note that a block with an integer-valued `learnt_in` option never overwrites a block with a date-valued `learnt_in` option, even if they correspond to the same period.

**learnt_in = INTEGER | DATE**

Used in conjunction with *perfect_foresight_with_expectation_errors_setup* and *perfect_foresight_with_expectation_errors_solver* commands, specifies the period or date at which this controlled paths block is learnt by agents.

*Example (perfect foresight)*

```
var c k;
varexo x z;

...

shocks;
  var x;
  periods 1;
```

(continues on next page)

(continued from previous page)

```
  values 1.2;
end;

perfect_foresight_controlled_paths;
  exogenize c;
  periods 2 4:5;
  values 1.6 1.7;
  endogenize x;

  exogenize k;
  periods 7:9;
  values 13;
  endogenize z;
end;

perfect_foresight_setup(periods = 100);
perfect_foresight_solver;
```

In this example, the exogenous variable `x` is equal to 1.2 in period 1, but in periods 2, 4 and 5 it will be endogenized so that endogenous variable $c$ is equal to 1.6 in period 2 and then 1.7 in periods 4 and 5. Similarly, the exogenous variable `z` will be endogenized in periods 7 to 9 so that the endogenous variable `k` is equal to 13 over the same periods.

*Example (perfect foresight with expectation errors)*

```
var c;
varexo x;

...

perfect_foresight_controlled_paths;
  exogenize c;
  periods 2002Y 2003Y:2005Y;
  values 1.6 1.7;
  endogenize x;
end;

perfect_foresight_controlled_paths(learnt_in=2004Y);
  exogenize c;
  periods 2004Y:2005Y;
  values 1.8;
  endogenize x;
end;

perfect_foresight_with_expectation_errors_setup(periods = 30,
    first_simulation_period = 2001Y);
perfect_foresight_with_expectation_errors_solver;
```

In this example, agents in year 2001 (at beginning of the simulation) compute their plan under the assumption that the endogenous variable `c` will be equal to 1.6 in year 2002 and 1.7 from years 2003 to 2005, and that exogenous variable `x` will behave so as to fulfill that constraint. Then, when 2004 arrives, they recompute their plan under the assumption that `c` will be equal to 1.8 in years 2004 and 2005 (and again that `x` will be endogenized accordingly).

### 4.12.4 Alternative syntax for specifying deterministic shocks

**Block:** **shock_paths** ;

**Block:** **shock_paths**(OPTIONS...);

This block provides an alternative way to describe shocks in a deterministic setup. It can be used to provide the same information as in the *shocks*, *mshocks*, *endval* and *perfect_foresight_controlled_paths* block (and cannot be used in conjunction with any of these, since it then supersedes them).

In addition to providing a single unified interface for describing temporary shocks, permanent shocks and controlled shocks, this block offers an extended syntax when describing the value of shocks: it is possible to refer to the value of variables from the *initval* block, from a previous simulation period, from external databases, or from a previous informational period (in the case of a simulation with expectation errors).

Two types of stanzas can appear within the block: shocks on exogenous variables (temporary and/or permanent), and controlled shocks as in *perfect_foresight_controlled_paths*.

Shocks on exogenous variables are described by stanzas of three lines of the following form:

```
var EXOGENOUS_NAME;
periods INTEGER[:INTEGER|end] | DATE[:DATE|end] | end [, INTEGER[:INTEGER|end] |␣
↪DATE[:DATE|end] | end]...;
values EXPRESSION [, EXPRESSION ]...;
```

Such a stanza describes a series of shocks on a given exogenous variable. There must be as many entries in the `periods` statement as there are in the `values` statement. Each entry gives the value of the exogenous variable over a single period or a range of periods (the latter if the syntax with a colon is used). Periods can be specified eiher as integers, as dates, or with the keyword `end`; the latter is used to impose a terminal condition on the variable, and thus describes a permanent shock.

As soon as some period is specified using the `end` keyword, the `shock_paths` block is considered as describing a permanent shock (possibly in combination with other types of shocks). This in turn implies that the *endval_steady* option of the *perfect_foresight_solver* command will be implicitly enabled, so that the terminal condition will be a steady state (also note that the `shock_paths` offers no way of specifying the terminal condition for endogenous variables, contrary to the *endval* block; and if used in combination with the *perfect_foresight_with_expectation_errors_solver* command, there is no change in behaviour, since the latter command always recomputes the terminal steady state).

Expressions in the `values` statement can be arbitrary algebraic expressions as in the *model* block. However, when referencing a variable, a "scope" must be given (as a prefix) to indicate where the value should be taken from (this does not apply to parameters which can be referenced without a scope):

- `initval.VARIABLE_NAME` indicates that the value of the variable should be taken from the *initval* block (for endogenous, it will be the steady state if a *steady* command follows the `initval` block). This can be abbreviated as `init.VARIABLE_NAME`.
- `self.VARIABLE_NAME(INTEGER)` refers to the value of an exogenous variable set in the `shock_paths` block. It can either refer to the variable currently being defined, in which case it must reference a previous period (using a lag notation similar to the one used in the *model* block); this can for example be useful to construct auto-regressive processes (see the example below). Or it can refer to a variable defined in a previous stanza (at the current period, or at a previous period using the lag notation).
- `DATABASE_NAME.VARIABLE_NAME` or `DATABASE_NAME.VARIABLE_NAME(INTEGER)` indicates that the value of the variable should be taken from a database (as declared with the *database* command). If the database has a time dimension, the value will be taken from corresponding period in the `periods` statement (possibly with a lead or lag if specified). If the database is flat and has no time dimension, then any period information is ignored.
- `prev.VARIABLE_NAME` or `prev.VARIABLE_NAME(INTEGER)` indicates that the value of the variable should be taken from the previous informational period, in the context of a simulation with expectation errors (when the `learnt_in` option is passed to the `shock_paths` block). In other words, it refers to the value of the variable as declared in the previous `shock_paths` block with the most recent value

for the `learnt_in` option. The value will be taken from the corresponding period in the `periods` statement (possibly with a lead or lag if specified).

- `learnt_in(INTEGER|DATE).VARIABLE_NAME` or `learnt_in(INTEGER|DATE).VARIABLE_NAME(INTEGER)` is similar to the `prev.VARIABLE_NAME` syntax, but offers the possibility to refer to an arbitrary previous `shock_paths` block with a different value for the `learnt_in` option. The informational period can be specified either as an integer or as a date.

Note that expressions do not need to be parenthesized as in *shocks* or *mshocks*, since separating commas are mandatory.

Controlled shocks are described by stanzas of four lines of the following form:

```
exogenize ENDOGENOUS_NAME;
periods INTEGER[:INTEGER] | DATE[:DATE] [, INTEGER[:INTEGER] | DATE[:DATE]]...;
values EXPRESSION [, EXPRESSION]...;
endogenize EXOGENOUS_NAME;
```

This tells the perfect foresight solver that the value of some endogenous variables will be controlled (in other words, they will be exogenized). It also gives the period(s) for which this control applies, the value(s) imposed to the endogenous variable(s), and the exogenous variable(s) that are left free at the same period(s) (in other words, those exogenous are endogenized).

Similarly to the case of exogenous shocks, variables referred in expressions must have a scope specified. Only the `initval` scope (possibly abbreviated as `init`) is currently supported.

The special `end` keyword for period is not supported (in other words, it is not possible to exogenize an endogenous variable for the terminal steady-state computation).

When a controlled shock appears in the block, the same restrictions as for *perfect_foresight_controlled_paths* apply to the *stack_solve_algo* option of the solver commands, and for the *block* and *bytecode* options of the *model* block and *model_options* commands.

In the various stanzas above, dates are accepted only if the *first_simulation_period* or *last_simulation_period* is passed to either *perfect_foresight_setup* or *perfect_foresight_with_expectation_errors_setup*.

*Options*

**learnt_in = INTEGER | DATE**

Used in conjunction with *perfect_foresight_with_expectation_errors_setup* and *perfect_foresight_with_expectation_errors_solver* commands, specifies the period or date at which this block is learnt by agents. Default: 1.

**overwrite**

This block cancels and replaces previous `shock_paths` blocks that have the same `learnt_in` option. Note that a block with an integer-valued `learnt_in` option never overwrites a block with a date-valued `learnt_in` option, even if they correspond to the same period.

*Example with integer periods*

```
db = table(transpose(linspace(0, 1, 101)), 'VariableNames', {'foo'});

database db;

shock_paths;
  // An autoregressive process for the first 5 periods, which
  // becomes a permanent shock thereafter.
  var x;
  periods 1, 2:5, 6:end;
  values initval.x*1.05, self.x(-1)*1.05, self.x(-1);

  // A temporary shock whose value is taken from a database
```

(continues on next page)

(continued from previous page)

```
  // Note that y will be 0 in period 1, 0.01 in period 2, 0.02 in
  // period 3
  var y;
  periods 1:3;
  values db.foo;

  // A controlled shock
  exogenize c;
  periods 2, 4:5;
  values 1.6, 1.7;
  endogenize z;
end;

perfect_foresight_setup(periods = 100);
perfect_foresight_solver;
```

*Example with dates*

It is numerically the same as the previous one.

```
db = dseries(transpose(linspace(0, 1, 101)), '2000Q1', {'foo'});

database db;

shock_paths;
  var x;
  periods 2000Q1, 2000Q2:2001Q1, 2001Q2:end;
  values initval.x*1.05, self.x(-1)*1.05, self.x(-1);

  var y;
  periods 2000Q1:2000Q3;
  values db.foo;

  // A controlled shock
  exogenize c;
  periods 2000Q2, 2000Q4:2001Q1;
  values 1.6, 1.7;
  endogenize z;
end;

perfect_foresight_setup(first_simulation_period = 2000Q1, periods = 100);
perfect_foresight_solver;
```

*Example with expectation errors*

```
shock_paths;
  var x;
  periods 1, 2:5, 6:end;
  values initval.x*1.05, self.x(-1)*1.05, self.x(-1);
end;

shock_paths(learnt_in = 6);
  // The permanent shock turns out to be 5% higher than initially␣
↪anticipated
  var x;
  periods 6:end;
  values prev.x * 1.05;
```

(continues on next page)

(continued from previous page)

```
end;

perfect_foresight_with_expectation_errors_setup(first_simulation_period␣
↪= 2000Q1,
                                                periods = 100);
perfect_foresight_with_expectation_errors_solver;
```

**Command:** **database** DATABASE_NAME [[,] DATABASE_NAME]... ;

Declares one or more databases to be referenced from expressions in the *shock_paths* block.

Each database should correspond to a variable of the same name in the MATLAB/Octave workspace. These variables can either contain:

- a MATLAB/Octave `table` object with a single row of data (no time dimension), and variable names in columns. The single row will be used for all simulation periods (including the terminal condition);
- a MATLAB/Octave `table` object with several rows (each corresponding to a time period), and variable names in columns. The first row will be used for the first simulation period, and so on up to the last simulation period. The row just after the one corresponding to the last simulation period will be used for the terminal condition;
- a *dseries* object. Such an object is accepted only if the *first_simulation_date* or *last_simulation_date* option is passed to either *perfect_foresight_setup* or *perfect_foresight_with_expectation_errors_setup*. Values for a given date in the timeseries object will be used for the corresponding simulation date. The values at the date just after the last simulation period will be used for the terminal condition.

## 4.13 Stochastic solution and simulation

In a stochastic context, Dynare computes one or several simulations corresponding to a random draw of the shocks.

The main algorithm for solving stochastic models relies on a Taylor approximation, up to third order, of the expectation functions (see Judd (1996), Collard and Juillard (2001), Collard and Juillard (2001), and Schmitt-Grohé and Uríbe (2004)). The details of the Dynare implementation of the first-order solution are given in Villemot (2011). Such a solution is computed using the *stoch_simul* command.

As an alternative, it is possible to compute a simulation to a stochastic model using the *extended path* method presented by Fair and Taylor (1983). This method is especially useful when there are strong nonlinearities or binding constraints. Such a solution is computed using the *extended_path* command.

### 4.13.1 Computing the stochastic solution

**Command:** **stoch_simul** [VARIABLE_NAME...];

**Command:** **stoch_simul**(OPTIONS...) [VARIABLE_NAME...];

Solves a stochastic (i.e. rational expectations) model, using perturbation techniques.

More precisely, `stoch_simul` computes a Taylor approximation of the model around the deterministic steady state and solves the decision and transition functions for the approximated model. Using this, it computes impulse response functions and various descriptive statistics (moments, variance decomposition, correlation and autocorrelation coefficients). For correlated shocks, the variance decomposition is computed as in the VAR literature through a Cholesky decomposition of the covariance matrix of the exogenous variables. When the shocks are correlated, the variance decomposition depends upon the order of the variables in the `varexo` command.

The Taylor approximation is computed around the steady state (see *Steady state*).

The IRFs are computed as the difference between the trajectory of a variable following a shock at the beginning of period 1 and its steady-state value. More details on the computation of IRFs can be found at https://git.dynare.org/Dynare/dynare/-/wikis/Irfs.

Variance decomposition, correlation, autocorrelation are only displayed for variables with strictly positive variance. Impulse response functions are only plotted for variables with response larger than $10^{-10}$.

Variance decomposition is computed relative to the sum of the contribution of each shock. Normally, this is of course equal to aggregate variance, but if a model generates very large variances, it may happen that, due to numerical error, the two differ by a significant amount. Dynare issues a warning if the maximum relative difference between the sum of the contribution of each shock and aggregate variance is larger than `0.01%`.

The covariance matrix of the shocks is specified with the `shocks` command (see *Shocks on exogenous variables*).

When a list of `VARIABLE_NAME` is specified, results are displayed only for these variables.

*Options*

**`ar = INTEGER`**

Order of autocorrelation coefficients to compute and to print. Default: `5`.

**`drop = INTEGER`**

Number of points (burn-in) dropped at the beginning of simulation before computing the summary statistics. Note that this option does not affect the simulated series stored in `oo_.endo_simul` and the workspace. Here, no periods are dropped. Default: `100`.

**`hp_filter = DOUBLE`**

Uses HP filter with $\lambda =$ `DOUBLE` before computing moments. If theoretical moments are requested, the spectrum of the model solution is filtered following the approach outlined in Uhlig (2001). Default: no filter.

**`one_sided_hp_filter = DOUBLE`**

Uses the one-sided HP filter with $\lambda =$ `DOUBLE` described in Stock and Watson (1999) before computing moments. This option is only available with simulated moments. Default: no filter.

**`bandpass_filter`**

Uses a bandpass filter with the default passband before computing moments. If theoretical moments are requested, the spectrum of the model solution is filtered using an ideal bandpass filter. If empirical moments are requested, the Baxter and King (1999) filter is used. Default: no filter.

**`bandpass_filter = [HIGHEST_PERIODICITY LOWEST_PERIODICITY]`**

Uses a bandpass filter before computing moments. The passband is set to a periodicity of to LOWEST_PERIODICITY, e.g. 6 to 32 quarters if the model frequency is quarterly. Default: `[6,32]`.

**`filtered_theoretical_moments_grid = INTEGER`**

When computing filtered theoretical moments (with either option `hp_filter` or option `bandpass_filter`), this option governs the number of points in the grid for the discrete Inverse Fast Fourier Transform. It may be necessary to increase it for highly autocorrelated processes. Default: `512`.

**`irf = INTEGER`**

Number of periods on which to compute the IRFs. Setting `irf=0` suppresses the plotting of IRFs. Default: `40`.

**`irf_shocks = ( VARIABLE_NAME [[,] VARIABLE_NAME ...] )`**

The exogenous variables for which to compute IRFs. Default: all.

**`relative_irf`**

Requests the computation of normalized IRFs. At first order, the normal shock vector of size one standard deviation is divided by the standard deviation of the current shock and multiplied by 100. The impulse responses are hence the responses to a unit shock of size 1 (as opposed to the regular shock size of one standard deviation), multiplied by 100. Thus, for a loglinearized model where the variables are measured in percent, the IRFs have the interpretation of the percent responses to a 100 percent shock. For example, a response of 400 of output to a TFP shock shows that output increases by 400 percent after a 100 percent TFP shock (you will see that TFP increases by 100 on impact). Given linearity at `order=1`, it is straightforward to rescale the IRFs stored in `oo_.irfs` to any desired size. At higher

order, the interpretation is different. The `relative_irf` option then triggers the generation of IRFs as the response to a 0.01 unit shock (corresponding to 1 percent for shocks measured in percent) and no multiplication with 100 is performed. That is, the normal shock vector of size one standard deviation is divided by the standard deviation of the current shock and divided by 100. For example, a response of 0.04 of log output (thus measured in percent of the steady-state output level) to a TFP shock also measured in percent then shows that output increases by 4 percent after a 1 percent TFP shock (you will see that TFP increases by 0.01 on impact).

**`irf_plot_threshold = DOUBLE`**

Threshold size for plotting IRFs. All IRFs for a particular variable with a maximum absolute deviation from the steady state smaller than this value are not displayed. Default: `1e-10`.

**`nocorr`**

Don't print the correlation matrix (printing them is the default).

**`nodecomposition`**

Don't compute (and don't print) unconditional variance decomposition.

**`nofunctions`**

Don't print the coefficients of the approximated solution (printing them is the default).

**`nomoments`**

Don't print moments of the endogenous variables (printing them is the default).

**`nomodelsummary`**

Don't print the model summary and the covariance of the exogenous shocks (printing them is the default).

**`nograph`**

Do not create graphs (which implies that they are not saved to the disk nor displayed). If this option is not used, graphs will be saved to disk (to the format specified by `graph_format` option, except if `graph_format=none`) and displayed to screen (unless `nodisplay` option is used).

**`graph`**

Re-enables the generation of graphs previously shut off with `nograph`.

**`nodisplay`**

Do not display the graphs, but still save them to disk (unless `nograph` is used).

**`graph_format = FORMAT`**

**`graph_format = ( FORMAT, FORMAT... )`**

Specify the file format(s) for graphs saved to disk. Possible values are `eps` (the default), `pdf`, `fig` and `none`. Under Octave, `fig` will use Octave's `ofig` format. If the file format is set equal to `none`, the graphs are displayed but not saved to the disk.

**`noprint`**

See *noprint*.

**`print`**

See *print*.

**`order = INTEGER`**

Order of Taylor approximation. Note that for third order and above, the `k_order_solver` option is implied, and only empirical moments are available (you must provide a value for `periods` option). Default: `2` (except after an `estimation` command, in which case the default is the value used for the estimation).

**`k_order_solver`**

Use a k-order solver (implemented in C++) instead of the default Dynare solver. This option is not yet compatible with the `bytecode` option (see *Model declaration*). Default: disabled for order 1 and 2, enabled for order 3 and above.

**periods = INTEGER**

If different from zero, empirical moments will be computed instead of theoretical moments. The value of the option specifies the number of periods to use in the simulations. Values of the initval block, possibly recomputed by *steady*, will be used as starting point for the simulation. The simulated endogenous variables are made available to the user in a vector for each variable and in the global matrix `oo_.endo_simul` (see *oo_.endo_simul*). The simulated exogenous variables are made available in `oo_.exo_simul` (see *oo_.exo_simul*). Default: `0`.

**qz_criterium = DOUBLE**

Value used to split stable from unstable eigenvalues in reordering the Generalized Schur decomposition used for solving first-order problems. Default: `1.000001` (except when estimating with `lik_init` option equal to `1`: the default is `0.999999` in that case; see *Estimation based on likelihood*).

**qz_zero_threshold = DOUBLE**

See *qz_zero_threshold*.

**replic = INTEGER**

Number of simulated series used to compute the IRFs. Default: `1` if `order=1`, and `50` otherwise.

**simul_replic = INTEGER**

Number of series to simulate when empirical moments are requested (i.e. `periods` $> 0$). Note that if this option is greater than 1, the additional series will not be used for computing the empirical moments but will simply be saved in binary form to the file `FILENAME_simul` in the `FILENAME/Output` folder. Default: `1`.

**solve_algo = INTEGER**

See *solve_algo*, for the possible values and their meaning.

**aim_solver**

Deprecated option equivalent to setting `dr=aim`.

**conditional_variance_decomposition = INTEGER**

**conditional_variance_decomposition = [INTEGER1:INTEGER2]**

**conditional_variance_decomposition = [INTEGER1 INTEGER2 ...]**

Computes a conditional variance decomposition for the specified period(s). The periods must be strictly positive. Conditional variances are given by $var(y_{t+k}|t)$. For period 1, the conditional variance decomposition provides the decomposition of the effects of shocks upon impact.

The results are stored in `oo_.conditional_variance_decomposition` (see *oo_.conditional_variance_decomposition*). In the presence of measurement error, the `oo_.conditional_variance_decomposition` field will contain the variance contribution after measurement error has been taken out, i.e. the decomposition will be conducted of the actual as opposed to the measured variables. The variance decomposition of the measured variables will be stored in `oo_.conditional_variance_decomposition_ME` (see *oo_.conditional_variance_decomposition_ME*). The variance decomposition is only conducted, if theoretical moments are requested, *i.e.* using the `periods=0` option. Only available at `order<3` and without `pruning`. In case of `order=2`, Dynare provides a second-order accurate approximation to the true second moments based on the linear terms of the second-order solution (see Kim *et al.*, 2008). Note that the unconditional variance decomposition *i.e.* at horizon infinity) is automatically conducted if theoretical moments are requested and if `nodecomposition` is not set (see *oo_.variance_decomposition*).

**pruning**

Discard higher-order terms when iteratively computing simulations of the solution. At second order, Dynare uses the algorithm of Kim *et al.* (2008), while at third order and higher its generalization by Andreasen *et al.* (2018) is used. When specified, theoretical moments are based on the pruned state space, i.e. the computation of second moments uses all terms as in Andreasen *et al.* (2018), page 10 as opposed to simply providing a second-order accurate result based on the linear solution as in Kim *et al.* (2008).

**partial_information**

Computes the solution of the model under partial information, along the lines of Pearlman *et al.* (1986). Agents are supposed to observe only some variables of the economy. The set of observed variables is declared using the `varobs` command. Note that if `varobs` is not present or contains all endogenous variables, then this is the full information case and this option has no effect. More references can be found at https://git.dynare.org/Dynare/dynare/-/wikis/PartialInformation.

**dr = OPTION**

Determines the method used to compute the decision rule. Possible values for OPTION are:

`default`

Uses the default method to compute the decision rule based on the generalized Schur decomposition (see Villemot (2011) for more information).

`cycle_reduction`

Uses the cycle reduction algorithm of Bini *et al.* (2002) to solve the polynomial equation for retrieving the coefficients associated to the endogenous variables in the decision rule. This method is faster than the default one for large scale models.

`logarithmic_reduction`

Uses the logarithmic reduction algorithm of Bini *et al.* (2002) to solve the polynomial equation for retrieving the coefficients associated to the endogenous variables in the decision rule. This method is in general slower than the `cycle_reduction`.

**aim**

Use the Anderson-Moore Algorithm (AIM) to compute the decision rules, instead of using Dynare's default method based on a generalized Schur decomposition. This option is only valid for first-order approximation. See AIM website for more details on the algorithm.

Default value is `default`.

**dr_cycle_reduction_tol = DOUBLE**

The convergence criterion used in the cycle reduction algorithm. Its default value is `1e-7`.

**dr_cycle_reduction_maxiter = INTEGER**

The maximum number of iterations used in the cycle reduction algorithm. Its default value is `100`.

**dr_logarithmic_reduction_tol = DOUBLE**

The convergence criterion used in the logarithmic reduction algorithm. Its default value is `1e-12`.

**dr_logarithmic_reduction_maxiter = INTEGER**

The maximum number of iterations used in the logarithmic reduction algorithm. Its default value is `100`.

**loglinear**

See *loglinear*. Note that ALL variables are log-transformed by using the Jacobian transformation, not only selected ones. Thus, you have to make sure that your variables have strictly positive steady states. *stoch_simul* will display the moments, decision rules, and impulse responses for the log-linearized variables. The decision rules saved in `oo_.dr` and the simulated variables will also be the ones for the log-linear variables.

**tex**

Requests the printing of results and graphs in TeX tables and graphics that can be later directly included in LaTeX files.

**dr_display_tol = DOUBLE**

Tolerance for the suppression of small terms in the display of decision rules. Rows where all terms are smaller than `dr_display_tol` are not displayed. Default value: `1e-6`.

**contemporaneous_correlation**

Saves the contemporaneous correlation between the endogenous variables in `oo_.contemporaneous_correlation`. Requires the `nocorr` option not to be set.

**spectral_density**

Triggers the computation and display of the theoretical spectral density of the (filtered) model variables. Results are stored in `oo_.SpectralDensity`, defined below. Default: do not request spectral density estimates.

*Output*

This command sets `oo_.dr`, `oo_.mean`, `oo_.var`, `oo_.var_list`, and `oo_.autocorr`, which are described below.

If the `periods` option is present, sets `oo_.skewness`, `oo_.kurtosis`, and `oo_.endo_simul` (see *oo_.endo_simul*).

If option `irf` is different from zero, sets `oo_.irfs` (see below).

If the option `contemporaneous_correlation` is different from `0`, sets `oo_.contemporaneous_correlation`, which is described below.

*Example*

```
shocks;
var e;
stderr 0.0348;
end;

stoch_simul;
```

Performs the simulation of the 2nd-order approximation of a model with a single stochastic shock `e`, with a standard error of `0.0348`.

*Example*

```
stoch_simul(irf=60) y k;
```

Performs the simulation of a model and displays impulse response functions on 60 periods for variables `y` and `k`.

**MATLAB/Octave variable: oo_.mean**

After a run of *stoch_simul*, contains the mean of the endogenous variables. Contains theoretical mean if the `periods` option is not present, and simulated mean otherwise. The variables are arranged in declaration order.

**MATLAB/Octave variable: oo_.var**

After a run of *stoch_simul*, contains the variance-covariance of the endogenous variables. Contains theoretical variance if the `periods` option is not present and simulated variance otherwise. Only available for `order<4`. At `order=2` it will be a second-order accurate approximation (i.e. ignoring terms of order 3 and 4 that would arise when using the full second-order policy function). At `order=3`, theoretical moments are only available with `pruning`. The variables are arranged in declaration order.

**MATLAB/Octave variable: oo_.var_list**

The list of variables for which results are displayed.

**MATLAB/Octave variable: oo_.skewness**

After a run of *stoch_simul* contains the skewness (standardized third moment) of the simulated variables if the `periods` option is present. The variables are arranged in declaration order.

**MATLAB/Octave variable: oo_.kurtosis**

After a run of *stoch_simul* contains the excess kurtosis (standardized fourth moment) of the simulated variables if the `periods` option is present. The variables are arranged in declaration order.

**MATLAB/Octave variable: oo_.autocorr**

After a run of *stoch_simul*, contains a cell array of the autocorrelation matrices of the endogenous variables. The element number of the matrix in the cell array corresponds to the order of autocorrelation. The option ar specifies the number of autocorrelation matrices available. Contains theoretical autocorrelations if the periods option is not present and simulated autocorrelations otherwise. Only available for order<4. At order=2 it will be a second-order accurate approximation. At order=3, theoretical moments are only available with pruning. The field is only created if stationary variables are present.

The element oo_.autocorr{i}(k,l) is equal to the correlation between $y_t^k$ and $y_{t-i}^l$, where $y^k$ (resp. $y^l$) is the $k$-th (resp. $l$-th) endogenous variable in the declaration order.

Note that if theoretical moments have been requested, oo_.autocorr{i} is the same as oo_.gamma_y{i+1}.

**MATLAB/Octave variable: oo_.gamma_y**

After a run of *stoch_simul*, if theoretical moments have been requested (i.e. if the periods option is not present), this variable contains a cell array with the following values (where ar is the value of the option of the same name):

oo_.gamma{1}

Variance/covariance matrix.

oo_.gamma{i+1} (for i=1:ar)

Autocorrelation function. See *oo_.autocorr* for more details. **Beware**, this is the autocorrelation function, not the autocovariance function.

oo_.gamma{ar+2}

Unconditional variance decomposition, see *oo_.variance_decomposition*.

oo_.gamma{ar+3}

If a second-order approximation has been requested, contains the vector of the mean correction terms.

Only available at order<4. In case order=2, the theoretical second moments are a second-order accurate approximation of the true second moments. See conditional_variance_decomposition. At order=3, theoretical moments are only available with pruning.

**MATLAB/Octave variable: oo_.variance_decomposition**

After a run of *stoch_simul* when requesting theoretical moments (periods=0), contains a matrix with the result of the unconditional variance decomposition (i.e. at horizon infinity). The first dimension corresponds to the endogenous variables (in the order of declaration after the command or in M_.endo_names) and the second dimension corresponds to exogenous variables (in the order of declaration). Numbers are in percent and sum up to 100 across columns. In the presence of measurement error, the field will contain the variance contribution after measurement error has been taken out, *i.e.* the decomposition will be conducted of the actual as opposed to the measured variables.

**MATLAB/Octave variable: oo_.variance_decomposition_ME**

Field set after a run of *stoch_simul* when requesting theoretical moments (periods=0) if measurement error is present. It is similar to *oo_.variance_decomposition*, but the decomposition will be conducted of the measured variables. The field contains a matrix with the result of the unconditional variance decomposition (*i.e.* at horizon infinity). The first dimension corresponds to the observed endoogenous variables (in the order of declaration after the command) and the second dimension corresponds to exogenous variables (in the order of declaration), with the last column corresponding to the contribution of measurement error. Numbers are in percent and sum up to 100 across columns.

**MATLAB/Octave variable: oo_.conditional_variance_decomposition**

After a run of *stoch_simul* with the conditional_variance_decomposition option, contains a three-dimensional array with the result of the decomposition. The first dimension corresponds to the endogenous variables (in the order of declaration after the command or in M_.endo_names if not specified), the second

dimension corresponds to the forecast horizons (as declared with the option), and the third dimension corresponds to the exogenous variables (in the order of declaration). In the presence of measurement error, the field will contain the variance contribution after measurement error has been taken out, *i.e.* the decomposition will be conducted of the actual as opposed to the measured variables.

**MATLAB/Octave variable: oo_.conditional_variance_decomposition_ME**

Field set after a run of *stoch_simul* with the `conditional_variance_decomposition` option if measurement error is present. It is similar to *oo_.conditional_variance_decomposition*, but the decomposition will be conducted of the measured variables. It contains a three-dimensional array with the result of the decomposition. The first dimension corresponds to the endogenous variables (in the order of declaration after the command or in `M_.endo_names` if not specified), the second dimension corresponds to the forecast horizons (as declared with the option), and the third dimension corresponds to the exogenous variables (in the order of declaration), with the last column corresponding to the contribution of the measurement error.

**MATLAB/Octave variable: oo_.contemporaneous_correlation**

After a run of *stoch_simul* with the `contemporaneous_correlation option`, contains theoretical contemporaneous correlations if the `periods` option is not present, and simulated contemporaneous correlations otherwise. Only available for `order<4`. At `order=2` it will be be a second-order accurate approximation. At `order=3`, theoretical moments are only available with `pruning`. The variables are arranged in declaration order.

**MATLAB/Octave variable: oo_.SpectralDensity**

After a run of *stoch_simul* with option `spectral_density`, contains the spectral density of the model variables. There will be a `nvars` by `nfrequencies` subfield `freqs` storing the respective frequency grid points ranging from 0 to $2\pi$ and a same sized subfield `density` storing the corresponding density.

**MATLAB/Octave variable: oo_.irfs**

After a run of *stoch_simul* with option `irf` different from zero, contains the impulse responses, with the following naming convention: *VARIABLE_NAME_SHOCK_NAME*.

For example, `oo_.irfs.gnp_ea` contains the effect on `gnp` of a one-standard deviation shock on `ea`.

**MATLAB/Octave command:**
**`IRF_MATRIX=get_irf('EXOGENOUS_NAME' [, 'ENDOGENOUS_NAME']... );`**

Given the name of an exogenous variable, returns the IRFs for the requested endogenous variable(s) (as they are stored in `oo_.irfs`) in the output `IRF_MATRIX`. The periods are stored along the first dimension, with the steady state in the first row. The variables are stored along the second dimension. If no endogenous variables were specified, the matrix contains all variables stored in `oo_.irfs`.

The approximated solution of a model takes the form of a set of decision rules or transition equations expressing the current value of the endogenous variables of the model as function of the previous state of the model and shocks observed at the beginning of the period. The decision rules are stored in the structure `oo_.dr` which is described below.

**MATLAB/Octave variable: oo_.dr**

Structure storing the decision rules. The subfields for different orders of approximation are explained below.

### 4.13.2 Typology and ordering of variables

Dynare distinguishes four types of endogenous variables:

*Purely backward (or purely predetermined) variables*

Those that appear only at current and past period in the model, but not at future period (i.e. at $t$ and $t-1$ but not $t+1$). The number of such variables is equal to `M_.npred`.

*Purely forward variables*

Those that appear only at current and future period in the model, but not at past period (i.e. at $t$ and $t+1$ but not $t-1$). The number of such variables is stored in `M_.nfwrd`.

*Mixed variables*

Those that appear at current, past and future period in the model (i.e. at $t$, $t+1$ and $t-1$). The number of such variables is stored in `M_.nboth`.

*Static variables*

Those that appear only at current, not past and future period in the model (i.e. only at $t$, not at $t+1$ or $t-1$). The number of such variables is stored in `M_.nstatic`.

Note that all endogenous variables fall into one of these four categories, since after the creation of auxiliary variables (see *Auxiliary variables*), all endogenous have at most one lead and one lag. We therefore have the following identity:

```
M_.npred + M_.both + M_.nfwrd + M_.nstatic = M_.endo_nbr
```

Internally, Dynare uses two orderings of the endogenous variables: the order of declaration (which is reflected in `M_.endo_names`), and an order based on the four types described above, which we will call the DR-order ("DR" stands for decision rules). Most of the time, the declaration order is used, but for elements of the decision rules, the DR-order is used.

The DR-order is the following: static variables appear first, then purely backward variables, then mixed variables, and finally purely forward variables. Inside each category, variables are arranged according to the declaration order.

**MATLAB/Octave variable: `oo_.dr.order_var`**

This variable maps DR-order to declaration order.

**MATLAB/Octave variable: `oo_.dr.inv_order_var`**

This variable contains the inverse map.

In other words, the k-th variable in the DR-order corresponds to the endogenous variable numbered `oo_.dr.order_var(k)` in declaration order. Conversely, k-th declared variable is numbered `oo_.dr.inv_order_var(k)` in DR-order.

Finally, the state variables of the model are the purely backward variables and the mixed variables. They are ordered in DR-order when they appear in decision rules elements. There are `M_.nspred = M_.npred + M_.nboth` such variables. Similarly, one has `M_.nsfwrd = M_.nfwrd + M_.nboth`, and `M_.ndynamic = M_.nfwrd + M_.nboth + M_.npred`.

**MATLAB/Octave variable: `M_.state_var`**

Structure identifying the state variables of the model (purely backward and mixed variables). It has the following fields:

**`M_.state_var.declaration_order`**

Vector of numerical indices of the state variables in declaration order. *M_.endo_names(M_.state_var.declaration_order)* gives the names of the state variables.

**`M_.state_var.dr_order`**

Vector of numerical indices of the state variables in DR-order.

### 4.13.3 First-order approximation

The approximation has the stylized form:

$$y_t = y^s + Ay_{t-1}^h + Bu_t$$

where $y^s$ is the steady-state value of $y$ and $y_t^h = y_t - y^s$.

**MATLAB/Octave variable: `oo.dr.state_var`**

Vector of numerical indices identifying the state variables in the vector of declared variables, *given the current parameter values* for which the decision rules have been computed. It is in decision-rule ordering.

The coefficients of the decision rules are stored as follows:

- $y^s$ is stored in `oo_.dr.ys`. The vector rows correspond to all endogenous in the declaration order.

- $A$ is stored in `oo_.dr.ghx`. The matrix rows correspond to all endogenous in DR-order. The matrix columns correspond to state variables in DR-order, as given by `oo_.dr.state_var`.
- $B$ is stored `oo_.dr.ghu`. The matrix rows correspond to all endogenous in DR-order. The matrix columns correspond to exogenous variables in declaration order.

Of course, the shown form of the approximation is only stylized, because it neglects the required different ordering in $y^s$ and $y_t^h$. The precise form of the approximation that shows the way Dynare deals with differences between declaration and DR-order, is

$$y_t(\text{oo_.dr.order_var}) = y^s(\text{oo_.dr.order_var}) + A \cdot y_{t-1}(\text{oo_.dr.order_var(k2)}) - y^s(\text{oo_.dr.order_var(k2)}) + B \cdot u_t$$

where k2 selects the state variables, $y_t$ and $y^s$ are in declaration order and the coefficient matrices are in DR-order. Effectively, all variables on the right-hand side are brought into DR order for computations and then assigned to $y_t$ in declaration order.

### 4.13.4 Second-order approximation

The approximation has the form:

$$y_t = y^s + 0.5\Delta^2 + Ay_{t-1}^h + Bu_t + 0.5C(y_{t-1}^h \otimes y_{t-1}^h) + 0.5D(u_t \otimes u_t) + E(y_{t-1}^h \otimes u_t)$$

where $y^s$ is the steady-state value of $y$, $y_t^h = y_t - y^s$, and $\Delta^2$ is the shift effect of the variance of future shocks. For the reordering required due to differences in declaration and DR order, see the first-order approximation.

The coefficients of the decision rules are stored in the variables described for first-order approximation, plus the following variables:

- $\Delta^2$ is stored in `oo_.dr.ghs2`. The vector rows correspond to all endogenous in DR-order.
- $C$ is stored in `oo_.dr.ghxx`. The matrix rows correspond to all endogenous in DR-order. The matrix columns correspond to the Kronecker product of the vector of state variables in DR-order.
- $D$ is stored in `oo_.dr.ghuu`. The matrix rows correspond to all endogenous in DR-order. The matrix columns correspond to the Kronecker product of exogenous variables in declaration order.
- $E$ is stored in `oo_.dr.ghxu`. The matrix rows correspond to all endogenous in DR-order. The matrix columns correspond to the Kronecker product of the vector of state variables (in DR-order) by the vector of exogenous variables (in declaration order).

### 4.13.5 Third-order approximation

The approximation has the form:

$$y_t = y^s + G_0 + G_1 z_t + G_2(z_t \otimes z_t) + G_3(z_t \otimes z_t \otimes z_t)$$

where $y^s$ is the steady-state value of $y$, and $z_t$ is a vector consisting of the deviation from the steady state of the state variables (in DR-order) at date $t-1$ followed by the exogenous variables at date $t$ (in declaration order). The vector $z_t$ is therefore of size $n_z$ = `M_.nspred` + `M_.exo_nbr`.

The coefficients of the decision rules are stored as follows:

- $y^s$ is stored in `oo_.dr.ys`. The vector rows correspond to all endogenous in the declaration order.
- $G_0$ is stored in `oo_.dr.g_0`. The vector rows correspond to all endogenous in DR-order.
- $G_1$ is stored in `oo_.dr.g_1`. The matrix rows correspond to all endogenous in DR-order. The matrix columns correspond to state variables in DR-order, followed by exogenous in declaration order.
- $G_2$ is stored in `oo_.dr.g_2`. The matrix rows correspond to all endogenous in DR-order. The matrix columns correspond to the Kronecker product of state variables (in DR-order), followed by exogenous (in declaration order). Note that the Kronecker product is stored in a folded way, i.e. symmetric elements are stored only once, which implies that the matrix has $n_z(n_z+1)/2$ columns. More precisely, each column of this matrix corresponds to a pair $(i_1, i_2)$ where each index represents an element of $z_t$ and is therefore between 1 and $n_z$. Only non-decreasing pairs are stored, i.e. those for which $i_1 \leq i_2$. The columns are

arranged in the lexicographical order of non-decreasing pairs. Also note that for those pairs where $i_1 \neq i_2$, since the element is stored only once but appears two times in the unfolded $G_2$ matrix, it must be multiplied by 2 when computing the decision rules.

- $G_3$ is stored in `oo_.dr.g_3`. The matrix rows correspond to all endogenous in DR-order. The matrix columns correspond to the third Kronecker power of state variables (in DR-order), followed by exogenous (in declaration order). Note that the third Kronecker power is stored in a folded way, i.e. symmetric elements are stored only once, which implies that the matrix has $n_z(n_z+1)(n_z+2)/6$ columns. More precisely, each column of this matrix corresponds to a tuple $(i_1, i_2, i_3)$ where each index represents an element of $z_t$ and is therefore between 1 and $n_z$. Only non-decreasing tuples are stored, i.e. those for which $i_1 \leq i_2 \leq i_3$. The columns are arranged in the lexicographical order of non-decreasing tuples. Also note that for tuples that have three distinct indices (i.e. $i_1 \neq i_2$ and $i_1 \neq i_3$ and $i_2 \neq i_3$), since these elements are stored only once but appears six times in the unfolded $G_3$ matrix, they must be multiplied by 6 when computing the decision rules. Similarly, for those tuples that have two equal indices (i.e. of the form $(a, a, b)$ or $(a, b, a)$ or $(b, a, a)$), since these elements are stored only once but appears three times in the unfolded $G_3$ matrix, they must be multiplied by 3 when computing the decision rules.

### 4.13.6 Higher-order approximation

Higher-order approximations are simply a generalization of what is done at order 3.

The steady state is stored in `oo_.dr.ys` and the constant correction is stored in `oo_.dr.g_0`. The coefficient for orders 1, 2, 3, 4… are respectively stored in `oo_.dr.g_0`, `oo_.dr.g_1`, `oo_.dr.g_2`, `oo_.dr.g_3`, `oo_.dr.g_4`… The columns of those matrices correspond to multidimensional indices of state variables, in such a way that symmetric elements are never repeated (for more details, see the description of `oo_.dr.g_3` in the third-order case).

## 4.14 Extended path

**Command:** **`extended_path ;`**
**Command:** **`extended_path`**`(OPTIONS...);`

Simulates a stochastic (i.e. rational expectations) model, using the extended path method presented by *Fair and Taylor (1983)*. Time series for the endogenous variables are generated by assuming that the agents believe that there will no more shocks in the following periods.

This function first computes a random path for the exogenous variables (stored in `oo_.exo_simul`, see *`oo_.exo_simul`*) and then computes the corresponding path for endogenous variables, taking the steady state as starting point. The result of the simulation is stored in `oo_.endo_simul` (see *`oo_.endo_simul`*). Note that this simulation approach does not solve for the policy and transition equations but for paths for the endogenous variables.

The variable *`oo_.extended_path.status`* indicates whether the simulation was successful or not.

*Options*

**`periods = INTEGER`**

The number of periods for which the simulation is to be computed. No default value, mandatory option.

**`solver_periods = INTEGER`**

The number of periods used to compute the solution of the perfect foresight at every iteration of the algorithm. Default: `200`.

**`order = INTEGER`**

If order is greater than `0`, Dynare uses numerical integration to take into account the effects of future uncertainty; this is called *stochastic* extended path, see Adjemian and Juillard (2025).[13] If `order` $= S$, then the time series for the endogenous variables are generated by assuming

[13] An ungated and up-to-date version of the document is available as Adjemian and Juillard (2025).

that the agents believe that there will no more shocks after period $t + S$. This feature can be quite slow. A non-zero value is not compatible with the `bytecode` option of the *model* block or *model_options* command. Default: `0`.

**hybrid [ = INTEGER ]**

By default, use the constant of the second-order perturbation reduced form to correct the paths generated by the (stochastic) extended path algorithm. The constant of a higher approximation order can be used, if a value is provided (it must be an even integer).

**lmmcp**

Solves the perfect foresight model with a Levenberg-Marquardt mixed complementarity problem (LMMCP) solver (Kanzow and Petra, 2004), which allows to consider inequality constraints on the endogenous variables (such as a ZLB on the nominal interest rate or a model with irreversible investment). For specifying the necessary complementarity conditions, see *lmmcp*.

**use_first_order_solution**

Utilize the model simulation based on a first-order local approximation as the initial guess for the nonlinear solver in each period. If this is not applied, solution in previous period is used.

**integration = OPTION**

Governs the quadrature method for numerical integration over future uncertainty when using stochastic extended path, i.e. `order` greater than `0`. Possible values for OPTION are:

`tensor_gaussian_quadrature`
: Gauss-Hermite quadrature is employed for numerical integration. For integrations involving more than one dimension, a product rule is applied. This method faces challenges related to the curse of dimensionality as the number of shocks increases. It is the default.

`stroud`
: It employs a Stroud-type monomial rule of degree 5 for numerical integration. While this method is less precise than Gauss-Hermite quadrature, it avoids the curse of dimensionality, as the number of nodes increases only polynomially with the dimension of the integration problem (i.e., the number of shocks). If the model includes only one shock, it defaults to Gaussian quadrature.

`unscented`
: Uses the Unscented Transform for the numerical integration.

**tree = OPTION**

Governs the treatment of the tree of forward paths in numerical integration. When `order` increases, the size of the auxiliary problem to be solved in each period increases. Possible values for OPTION are:

`perfect`
: Uses the full tree of forward paths and is therefore subject to the curse of dimensionality with respect to `order`. In practical applications involving this tree structure, we can only consider moderate orders of approximation. This is the default.

`sparse`
: Uses fishbone integration. By eliminating branches that do not directly emerge from the trunk, the size of the tree increases only linearly with `order`.

**number_of_quadrature_nodes = INTEGER**

Specifies the number of nodes per dimension to be used in numerical integration when using `integration=tensor_gaussian_quadrature` . The provided value must be an odd integer. Default: `5`.

*Output*

The simulation results are stored in the same format as standard Dynare simulations: simulated endogenous variables in *oo_.endo_simul*.

**MATLAB/Octave variable: oo_.extended_path.status**

Set to `true` by the *extended_path* command if the simulation succeeded, otherwise set to `false`.

**MATLAB/Octave variable: Simulated_time_series**

`dseries` object storing the simulated time series generated by the *extended_path* command.

**MATLAB/Octave variable: oo_.exo_simul**

Values of the exogenous variables simulated by the *extended_path* command, see *oo_.exo_simul*. Note that while the initial, i.e., period 0 shocks are stored, the ones of terminal period `T+1` are not. The field will generally be one period shorted than for *perfect_foresight_solver*.

**MATLAB/Octave variable: oo_.endo_simul**

Values of the endogenous variables simulated by the *extended_path* command, see *oo_.endo_simul*. Note that while the initial condition, i.e., the period 0 values are stored, the ones of terminal period `T+1` are not. The field will generally be one period shorted than for *perfect_foresight_solver*.

## 4.15 Occasionally binding constraints (OccBin)

Dynare allows simulating models with up to two occasionally-binding constraints by relying on a piecewise linear solution as in Guerrieri and Iacoviello (2015). It also allows estimating such models employing either the inversion filter of Cuba-Borda *et al.* (2019) or the piecewise Kalman filter of Giovannini *et al.* (2021). To trigger computations involving occasionally-binding constraints requires

1. defining and naming the occasionally-binding constraints using an `occbin_constraints` block
2. specifying the model equations for the respective regimes in the `model` block using appropriate equation tags.
3. potentially specifying a sequence of surprise shocks using a *shocks(surprise)* block
4. setting up OccBin simulations or estimation with *occbin_setup*
5. triggering a simulation with *occbin_solver* or running *estimation* or *calib_smoother*.

All of these elements are discussed in the following.

**Block: occbin_constraints ;**

The `occbin_constraints` block specifies the occasionally-binding constraints. It contains one or two of the following lines:

name 'STRING'; bind EXPRESSION; [relax EXPRESSION;] [error_bind EXPRESSION;] [error_relax EXPRESSION;]

`STRING` is the name of constraint that is used to reference the constraint in `relax` / `bind` equation tags to identify the respective regime (see below). The `bind` expression is mandatory and defines a logical condition that is evaluated in the baseline/steady-state regime to check whether the specified constraint becomes binding. In contrast, the `relax` expression is optional and specifies a logical condition that is evaluated in the binding regime to check whether the regime returns to the baseline/steady-state regime. If not specified, Dynare will simply check in the binding regime whether the `bind` expression evaluates to false. However, there are cases where the `bind` expression cannot be evaluated in the binding regime(s), because the variables involved are constant by definition so that e.g. the value of the Lagrange multiplier on the complementary slackness condition needs to be checked. In these cases, it is necessary to provide an explicit condition that can be evaluated in the binding regime that allows to check whether it should be left.

Note that the baseline regime denotes the steady state of the model where the economy will settle in the long-run without shocks. For that matter, it may be one where e.g. a borrowing constraint is binding. In that type of setup, the `bind` condition is used to specify the condition when this borrowing constraint becomes non-binding so that the alternative regime is entered.

Three things are important to keep in mind when specifying the expressions. First, feasible expressions may only contain contemporaneous endogenous variables. If you want to include leads/lags or exogenous

variables, you need to define an auxiliary variable. Second, Dynare will at the current stage not linearly approximate the entered expressions. Because OccBin will work with a linearized model, consistency will often require the user to enter a linearized constraint. Otherwise, the condition employed for checking constraint violations may differ from the one employed within model simulations based on the piecewise-linear model solution. Third, in contrast to the original OccBin replication codes, the variables used in expressions are not automatically demeaned, i.e. they refer to the levels, not deviations from the steady state. To access the steady-state level of a variable, the `STEADY_STATE()` operator can be used.

Finally, it's worth keeping in mind that for each simulation period, OccBin will check the respective conditions for whether the current regime should be left. Small numerical differences from the cutoff point for a regime can sometimes lead to oscillations between regimes and cause a spurious periodic solution. Such cases may be prevented by introducing a small buffer between the two regimes, e.g.

```
occbin_constraints;
name 'ELB'; bind inom <= iss-1e-8; relax inom > iss+1e-8;
end;
```

The `error_bind` and `error_relax` options are optional and allow specifying numerical criteria for the size of the respective constraint violations employed in numerical routines. By default, Dynare will simply use the absolute value of the `bind` and `relax` inequalities. But occasionally, user-specified expressions perform better.

*Example*

```
occbin_constraints;
    name 'IRR'; bind log_Invest-log(steady_state(Invest))<log(phi);␣
↪relax Lambda<0;
    name 'INEG'; bind log_Invest-log(steady_state(Invest))<0;
end;
```

IRR is a constraint for irreversible investment that becomes binding if investment drops below its steady state by more than 0.025 percent in the non-binding regime. The constraint will be relaxed whenever the associated Lagrange multiplier `Lambda` in the binding regime becomes negative. Note that the constraint here takes on a linear form to be consistent with a piecewise linear model solution

The specification of the model equations belonging to the respective regimes is done in the `model` block, with equation tags indicating to which regime a particular equation belongs. All equations that differ across regimes must have a `name` tag attached to them that allows uniquely identifying different versions of the same equation. The name of the constraints specified is then used in conjunction with a `bind` or `relax` tag to indicate to which regime a particular equation belongs. In case of more than one occasionally-binding constraint, if an equation belongs to several regimes (e.g. both constraints binding), the constraint name tags must be separated by a comma. If only one name tag is present, the respective equation is assumed to hold for both states of the other constraint.

*Example*

```
[name='investment',bind='IRR,INEG']
(log_Invest - log(phi*steady_state(Invest))) = 0;
[name='investment',relax='IRR']
Lambda=0;
[name='investment',bind='IRR',relax='INEG']
(log_Invest - log(phi*steady_state(Invest))) = 0;
```

The three entered equations for the investment condition define the model equation for all four possible combinations of the two constraints. The first equation defines the model equation in the regime where both the IRR and INEG constraint are binding. The second equation defines the model equation for the regimes where the IRR constraint is non-binding, regardless of whether the INEG constraint is binding or not. Finally, the last equation defines the model equation for the final regime where the IRR constraint is binding, but the INEG one is not.

**Block:** **shocks**(surprise) ;

**Block:** **shocks**(surprise,overwrite);

The shocks(surprise) block allows specifying a sequence of temporary changes in the value of exogenous variables that in each period come as a surprise to agents, i.e. are not anticipated. Note that to actually use the specified shocks in subsequent commands like *occbin_solver*, the block needs to be followed by a call to *occbin_setup*.

The block mirrors the perfect foresight syntax in that it should contain one or more occurrences of the following group of three lines:

```
var VARIABLE_NAME;
periods INTEGER[:INTEGER] [[,] INTEGER[:INTEGER]]...;
values DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
```

*Example* (with vector values and overwrite option)

```
shockssequence = randn(100,1)*0.02;

shocks(surprise,overwrite);
var epsilon;
periods 1:100;
values (shockssequence);
end;
```

**Command:** **occbin_setup** ;

**Command:** **occbin_setup**(OPTIONS...);

Prepares a simulation with occasionally binding constraints. This command will also translate the contents of a *shocks(surprise)* block for use in subsequent commands.

In order to conduct *estimation* with occasionally binding constraints, it needs to be prefaced by a call to *occbin_setup* to trigger the use of either the inversion filter or the piecewise Kalman filter (default). An issue that can arise in the context of estimation is a structural shock dropping out of the model in a particular regime. For example, at the zero lower bound on interest rates, the monetary policy shock in the Taylor rule will not appear anymore. This may create a problem if there are then more observables than shocks. The way to handle this issue depends on the type of filter used. The first step is to set the data points for the zero interest rate period to NaN. For the piecewise Kalman filter, the standard deviation of the associated shock needs to be set to 0 for the corresponding periods using the heteroskedastic_shocks block. This avoids stochastic singularity. However, this approach does not work for the inversion filter as the heteroskedastic_shocks block does not do anything here. For the inversion filter, as many shocks as observables are required at each point in time. Dynare assumes a one-to-one mapping between the declared shocks in varexo and declared observables in varobs. For example, if the second declared observable is NaN in a given period, Dynare will drop the second declared shock.

> **Warning**
>
> If there are missing values, it is imperative for the inversion filter that the declaration order of shocks and observables is conformable. Sticking with our example, if the nominal interest is the second varobs and is set to NaN, the inversion filter will drop the second declared shock. If that second declared shock is, e.g., a TFP shock, it will be dropped instead of the intended monetary policy shock.

Note that models with unit roots will require the user to specify the diffuse_filter option as otherwise Blanchard-Kahn errors will be triggered. For the piecewise Kalman filter, the initialization steps in the diffuse filter will always rely on the model solved for the baseline regime, without checking whether this is the actual regime in the first period(s).

*Example*

```
occbin_setup(likelihood_inversion_filter,smoother_inversion_
↪filter);
estimation(smoother,heteroskedastic_filter,...);
```

The above piece of code sets up an estimation employing the inversion filter for both the likelihood evaluation and the smoother, while also accounting for `heteroskedastic_shocks` using the `heteroskedastic_filter` option.

Be aware that OccBin has largely command-specific options, i.e. there are separate options to control the behavior of OccBin when called by the smoother or when computing the likelihood. These latter commands will not inherit the options potentially previously set for simulations.

*Options*

### Simulation options

**`simul_periods = INTEGER`**

Number of periods of the simulation. Default: 100.

**`simul_maxit = INTEGER`**

Maximum number of iterations when trying to find the regimes of the piecewise solution. Default: 30.

**`simul_check_ahead_periods = INTEGER`**

Number of periods for which to check ahead for return to the baseline regime. This number should be chosen large enough, because OccBin requires the simulation to return to the baseline regime at the end of time. Default: 200.

**`simul_reset_check_ahead_periods`**

Allows to reset *`simul_check_ahead_periods`* to its specified value at the beginning of each simulation period. Otherwise, the original value may permanently increase endogenously at some point due to regimes that last very long in expectations. This may considerably slow down convergence in subsequent periods. Default: not enabled.

**`simul_reset_regime_in_new_period`**

**`simul_reset_regime_in_new_period = BOOLEAN`**

Disable the default feature that the guess regime in period `t+1` equals the 1-step-ahead expected regime in period `t`. This can improve convergence when the expected regime is inconsistent with the regime triggered by new shocks in `t+1`. Default: not enabled.

**`simul_max_check_ahead_periods = INTEGER`**

If set to a finite number, it enforces the OccBin algorithm to check ahead only for the maximum number of periods (i.e. when we want agents to be myopic beyond some future period) instead of potentially endogenously increasing *`simul_check_ahead_periods`* ever further. Default: Inf.

**`simul_curb_retrench`**

Instead of basing the initial regime guess for the current iteration on the last iteration, update the guess only one period at a time. This will slow down the iterations, but may lead to more robust convergence behavior. Default: not enabled.

**`simul_periodic_solution`**

Accept a periodic solution where the solution alternates between two sets of results across iterations, i.e. is not found to be unique. This is sometimes caused by spurious numerical errors that lead to oscillations between regiems and may be prevented by allowing for a small buffer in regime transitions. Default: not enabled.

**simul_periodic_solution_threshold = INTEGER**

Maximum difference allowed between regimes in the periodic loop. A value of 1 means the duration of a constrained regime can change by at most 1 period; otherwise, the periodic solution is rejected. Default: 1.

**simul_periodic_solution_strict**

**simul_periodic_solution_strict = BOOLEAN**

Enforce strict acceptance of a periodic solution consistent with *simul_periodic_solution_threshold*. When disabled, a periodic solution is accepted if at least one forward path of regimes within the periodic loop does not violate constraints in expectation. Default: true.

**simul_debug**

Provide additional debugging information during solving. Default: not enabled.

### Smoother options

**smoother_periods = INTEGER**

Number of periods employed during the simulation when called by the smoother (equivalent of *simul_periods*). Default: 100.

**smoother_maxit = INTEGER**

Maximum number of iterations employed during the simulation when called by the smoother (equivalent of *simul_maxit*). Default: 30.

**smoother_check_ahead_periods = INTEGER**

Number of periods for which to check ahead for return to the baseline regime during the simulation when called by the smoother (equivalent of *simul_check_ahead_periods*). Default: 200.

**smoother_max_check_ahead_periods = INTEGER**

If set to a finite number, it enforces the OccBin algorithm to check ahead only for the maximum number of periods (i.e. when we want agents to be myopic beyond some future period) instead of potentially endogenously increasing *smoother_check_ahead_periods* ever further. Equivalent of *simul_max_check_ahead_periods*. Default: Inf.

**smoother_curb_retrench**

Have the smoother invoke the *simul_curb_retrench* option during simulations. Default: not enabled.

**smoother_periodic_solution**

Accept periodic solution where solution alternates between two sets of results (equivalent of *simul_periodic_solution*). Default: not enabled.

**smoother_max_check_ahead_periods = INTEGER**

If set to a finite number, it enforces the OccBin algorithm to check ahead only for the maximum number of periods (i.e. when we want agents to be myopic beyond some future period) instead of potentially endogenously increasing *smoother_check_ahead_periods* ever further. Equivalent of *simul_max_check_ahead_periods*. Default: Inf.

**smoother_inversion_filter**

Employ the inversion filter of Cuba-Borda *et al.* (2019) when running the smoother. The underlying assumption is that the system starts at the steady state. In this case, the inversion filter will provide the required smoother output. Default: not enabled.

**smoother_piecewise_kalman_filter**

Employ the piecewise Kalman filter of Giovannini *et al.* (2021) when running the smoother. Default: enabled.

**smoother_plot**

Plot linear vs piecewise linear smoothed shock estimates. Default: enabled.

**smoother_first_period_occbin_update = INTEGER**

First period in which PKF filter steps are used when running the smoother. If greater than 1, linear Kalman filter steps are used before this period. Default: 1.

**smoother_max_number_of_iterations = INTEGER**

Maximum number of smoother iterations that attempt to find a consistent sequence of shocks and regimes. Default: 10.

### Likelihood computation options

**likelihood_periods = INTEGER**

Number of periods employed during the simulation when computing the likelihood (equivalent of *simul_periods*). Default: 100.

**likelihood_maxit = INTEGER**

Maximum number of iterations employed during the simulation when computing the likelihood (equivalent of *simul_maxit*). Default: 30.

**likelihood_check_ahead_periods = INTEGER**

Number of periods for which to check ahead for return to the baseline regime during the simulation when computing the likelihood (equivalent of *simul_check_ahead_periods*). Default: 200.

**likelihood_max_check_ahead_periods = INTEGER**

If set to a finite number, it enforces the OccBin algorithm to check ahead only for the maximum number of periods (i.e. when we want agents to be myopic beyond some future period) instead of potentially endogenously increasing *likelihood_check_ahead_periods* ever further (equivalent of *simul_max_check_ahead_periods*). Default: Inf.

**likelihood_curb_retrench**

Have the likelihood computation invoke the *simul_curb_retrench* option during simulations. Default: not enabled.

**likelihood_periodic_solution**

Accept periodic solution where solution alternates between two sets of results (equivalent of *simul_periodic_solution*). Default: not enabled.

**likelihood_inversion_filter**

Employ the inversion filter of Cuba-Borda *et al.* (2019) when estimating the model. Default: not enabled.

**likelihood_piecewise_kalman_filter**

Employ the piecewise Kalman filter of Giovannini *et al.* (2021) when estimating the model. Note that this filter is incompatible with univariate Kalman filters, i.e. *kalman_algo* =2,4. Default: enabled.

**likelihood_max_kalman_iterations**

Maximum number of iterations of the outer loop for the piecewise Kalman filter. Default: 10.

### Piecewise Kalman filter and particle filter options

**filter_use_relaxation**

Triggers relaxation within the guess and verify algorithm used in the update step of the piecewise Kalman filter. When old and new guess regime differ to much, use a new guess closer to the previous guess. In case of multiple solutions, tends to provide an occasionally binding regime with a shorter duration (typically preferable). Specifying this option may slow down convergence. Default: not enabled.

**filter_particle_draw_states_from_empirical_density**

**filter_particle_draw_states_from_empirical_density = BOOLEAN**

When using the particle filter, draw state particles from the empirical density of past draws. Default: true.

**filter_particle_state_importance_sampling_pkf_init**

**filter_particle_state_importance_sampling_pkf_init = BOOLEAN**

Use the piecewise Kalman filter proposal density in the particle filter's state-importance-sampling step. Default: enabled.

**filter_particle_state_importance_sampling_logpost_crit_threshold = DOUBLE**

Log-posterior critical threshold between PKF and PPF data density estimates above which additional robustness checks for PPF filter step are triggered.

Default: 5.

**filter_particle_state_importance_sampling_slice_override_iteration = INTEGER**

Number of importance sampling iterations (sequential Monte Carlo iterations) before resorting to slice sampling for state update.

Default: 100.

**filter_particle_state_importance_sampling_slice_burnin = INTEGER**

Burn-in iterations for slice sampling in state importance sampling. Default: 10.

**filter_particle_initial_state_ergodic_simul**

**filter_particle_initial_state_ergodic_simul = BOOLEAN**

Draw initial states from an ergodic simulation when initializing the particle filter. Default: false.

**filter_particle_diagnostics**

**filter_particle_diagnostics = BOOLEAN**

Trigger diagnostic tests/plots for the particle filter state update. Default: false.

**filter_particle_diagnostics_graph_periods = INTEGER_VECTOR**

Select specific critical periods for which to run the diagnostic tests. Default: empty.

**filter_particle_diagnostics_nograph**

**filter_particle_diagnostics_nograph = BOOLEAN**

Disable diagnostic graphs. Default: true.

**filter_particle_number_of_particles = INTEGER**

Number of particles used by the particle filter. Default: 127.

**filter_particle_number_of_shocks_per_particle = INTEGER**

Number of shock draws per particle. Default: 1.

**filter_particle_state_draws = NUMERICAL_VECTOR**

Provide a fixed set of state draws for particle filtering. Default: empty (draw internally).

**particle_filtering**

**particle_filtering = BOOLEAN**

Activate the particle filter branch within OccBin filtering. Default: false.

**filter_particle_use_pkf_updated_state_threshold = INTEGER**

Threshold controlling when to re-use PKF-updated states in the particle filter. Default: 1.

**filter_init_periods_using_particles**

**filter_init_periods_using_particles = BOOLEAN**

This triggers particle filtering only for initial periods, in an otherwise standard PKF likelihood evaluation. Since state uncertainty is larger in first periods, this may be enough to robustify estimates when constraints are binding at the beginning of the sample. The algorithm automatically swithes to PKF (no particles) once all constraints become slack for the first time. Default: false.

### Posterior importance sampling

**posterior_importance_sampling**

**posterior_importance_sampling = BOOLEAN**

Enables posterior importance sampling of estimated parameters/shocks starting from a previous MCMC estimation. For example after MCMC estimation with linear model, re-sample posterior draws using the piecewise Kalman filter (*posterior_importance_sampling_filter* ='pkf') or the piecewise particle filter (*posterior_importance_sampling_filter* ='ppf'). Or, after estimating with piecewise Kalman filter, run this with piecewise particle filter. Given that the piecewise Kalman filter is more computationally intensive than the standard Kalman filter and that the piecewise particle filter more costly than the piecewise Kalman filter, this allows for faster robustness checks regarding the possible bias of the estimation when using the original (KF or PKF) filter. This requires mh results from previous estimation to be present in `metropolis` folder

Default: false.

**posterior_importance_sampling_filter = QUOTED_STRING**

Sets the new filter to be used for importance sampling. Values are:

`'pkf'`
: Uses the piecewise Kalman filter.

`'ppf'`
: Uses the piecewise particle filter.

Default: `'pkf'`.

**posterior_importance_sampling_orig_filter = QUOTED_STRING**

Indicates the filter used for original posterior draws. Values are:

`'linear'`

`'pkf'`

Default: `'linear'`.

**posterior_importance_sampling_sub_draws = INTEGER**

Sets the number of sub-draws to be used for evaluating posterior kernel with new filter and subsequent importance sampling. Default: the one specified with *sub_draws*.

**posterior_importance_sampling_orig_dname = FILENAME**

Sets the `results` folder if it is not the default one. Default: current *dirname*.

**posterior_importance_sampling_orig_fname = FILENAME**

Sets the name of the original model, in case it is different from current one. Default: current `FNAME`.

*Output*

The paths for the exogenous variables are stored into `options_.occbin.simul.SHOCKS`.

**Command: occbin_solver ;**

**Command: occbin_solver**(OPTIONS...);

Computes a simulation with occasionally-binding constraints based on a piecewise-linear solution.

Note that *occbin_setup* must be called before this command in order for the simulation to take into account previous *shocks(surprise)* blocks.

*Options*

**simul_periods = INTEGER**

See *simul_periods*.

**simul_maxit = INTEGER**

See *simul_maxit*.

**simul_check_ahead_periods = INTEGER**

See *simul_check_ahead_periods*.

**simul_reset_check_ahead_periods**

See *simul_reset_check_ahead_periods*.

**simul_max_check_ahead_periods**

See *simul_max_check_ahead_periods*.

**simul_curb_retrench**

See *simul_curb_retrench*.

**simul_debug**

See *simul_debug*.

*Output*

The command outputs various objects into `oo_.occbin`.

**MATLAB/Octave variable: oo_.occbin.simul.piecewise**

Matrix storing the simulations based on the piecewise-linear solution. The variables are arranged by column, in order of declaration (as in `M_.endo_names`), while the the rows correspond to the `simul_periods`.

**MATLAB/Octave variable: oo_.occbin.simul.linear**

Matrix storing the simulations based on the linear solution, i.e. ignoring the occasionally binding constraint(s). The variables are arranged column by column, in order of declaration (as in `M_.endo_names`), while the the rows correspond to the `simul_periods`.

**MATLAB/Octave variable: oo_.occbin.simul.shocks_sequence**

Matrix storing the shock sequence employed during the simulation. The shocks are arranged column by column, with their order in `M_.exo_names` stored in `oo_.occbin.exo_pos`. The the rows correspond to the number of shock periods specified in a *shocks(surprise)* block, which may be smaller than `simul_periods`.

**MATLAB/Octave variable: oo_.occbin.simul.regime_history**

Structure storing information on the regime history, conditional on the shock that happened in the respective period (stored along the rows). `type` is equal to either `smoother` or `simul`, depending on whether the output comes from a run of simulations or the smoother. The subfield `regime` contains a vector storing the regime state, while the the subfield `regimestart` indicates the expected start of the respective regime state. For example, if row 40 contains `[1,0]` for `regime2` and `[1,6]` for `regimestart2`, it indicates that - after the shock in period 40 has occurred - the second constraint became binding (1) and is expected to revert to non-binding (0) after six periods including the current one, i.e. period 45.

**MATLAB/Octave variable: oo_.occbin.simul.ys**

Vector of steady-state values

**Command: occbin_graph** `[VARIABLE_NAME...];`

**Command: occbin_graph**`(OPTIONS...) [VARIABLE_NAME...];`

Plots a graph comparing the simulation results of the piecewise-linear solution with the occasionally binding contraints to the linear solution ignoring the constraint.

*Options*

**noconstant**

Omit the steady state in the graphs.

**Command: occbin_write_regimes ;**

**Command:** **occbin_write_regimes**(OPTIONS...);

Write the information on the regime history stored in `oo_.occbin.simul.regime_history` or `oo_.occbin.smoother.regime_history` into an Excel file stored in the `FILENAME/Output` folder.

*Options*

**periods = INTEGER**

Number of periods for which to write the expected regime durations. Default: write all available periods.

**filename = FILENAME**

Name of the Excel file to write. Default: `FILENAME_occbin_regimes`.

**simul**

Selects the regime history from the last run of simulations. Default: enabled.

**smoother**

Selects the regime history from the last run of the smoother. Default: use `simul`.

## 4.16 Estimation based on likelihood

Provided that you have observations on some endogenous variables, it is possible to use Dynare to estimate some or all parameters. Both maximum likelihood (as in Ireland (2004)) and Bayesian techniques (as in Fernández-Villaverde and Rubio-Ramírez (2004), Rabanal and Rubio-Ramírez (2005), Schorfheide (2000) or Smets and Wouters (2003)) are available. Using Bayesian methods, it is possible to estimate DSGE models, VAR models, or a combination of the two techniques called DSGE-VAR.

Note that in order to avoid stochastic singularity, you must have at least as many shocks or measurement errors in your model as you have observed variables.

### 4.16.1 Specifying the observables, and trends

**Command:** **varobs** VARIABLE_NAME...;

This command lists the name of observed endogenous variables for the estimation procedure. These variables must be available in the data file (see *estimation*).

Alternatively, this command is also used in conjunction with the `partial_information` option of *stoch_simul*, for declaring the set of observed variables when solving the model under partial information.

Only one instance of `varobs` is allowed in a model file. If one needs to declare observed variables in a loop, the macro processor can be used as shown in the second example below.

*Example*

```
varobs C y rr;
```

Declares endogenous variables `C`, `y` and `rr` as observed variables.

*Example* (with a macro processor loop)

```
varobs
@#for co in countries
GDP_@{co}
@#endfor
;
```

**Block:** **observation_trends** ;

**Block:** **observation_trends**(observation_trend_base_period = INTEGER) ;

**Block:** **observation_trends**(observation_trend_base_period = DATE) ;

This block specifies linear trends for observed variables as functions of model parameters. In case the `loglinear` option is used, this corresponds to a linear trend in the logged observables, i.e. an exponential trend in the level of the observables.

Optional input argument `observation_trend_base_period` allows to keep control of the implied level of cointegrated observables that can be affected when the trend term is zero in first period (as opposed to e.g. the base period of the variables). Essentially the default trend [0:T] adds a constant = T/2 to all levels, which may require to systematically recalibrate all steady states while estimating the trend coefficient. This option allows to set the zero of the trend in any specified period, e.g. in the base period of the data.

Each line inside the block should be of the form:

```
VARIABLE_NAME(EXPRESSION);
```

In most cases, variables shouldn't be centered when `observation_trends` is used.

*Example*

```
observation_trends;
Y (eta);
P (mu/eta);
end;
```

*Example*

```
observation_trends(observation_trend_base_period=2010Q2);
Y (eta);
P (mu/eta);
end;
```

### 4.16.2 Specifying the estimated parameters and priors

**Block:** **estimated_params** ;

**Block:** **estimated_params**(overwrite) ;

This block lists all parameters to be estimated and specifies bounds and priors as necessary.

Each line corresponds to an estimated parameter.

In a maximum likelihood or a method of moments estimation, each line follows this syntax:

```
stderr VARIABLE_NAME | corr VARIABLE_NAME_1, VARIABLE_NAME_2 | skew VARIABLE_
↪NAME | PARAMETER_NAME
, INITIAL_VALUE [, LOWER_BOUND, UPPER_BOUND ];
```

In a Bayesian MCMC or a penalized method of moments estimation, each line follows this syntax:

```
stderr VARIABLE_NAME | corr VARIABLE_NAME_1, VARIABLE_NAME_2 | skew VARIABLE_
↪NAME | PARAMETER_NAME | DSGE_PRIOR_WEIGHT
[, INITIAL_VALUE [, LOWER_BOUND, UPPER_BOUND]], PRIOR_SHAPE,
PRIOR_MEAN, PRIOR_STANDARD_ERROR [, PRIOR_3RD_PARAMETER [,
PRIOR_4TH_PARAMETER [, SCALE_PARAMETER ] ] ];
```

The first part of the line consists of one of the five following alternatives:

- `stderr VARIABLE_NAME`

  Indicates that the standard error of the exogenous variable VARIABLE_NAME, or of the observation error/measurement errors associated with endogenous observed variable VARIABLE_NAME, is to be estimated.

- `corr VARIABLE_NAME1, VARIABLE_NAME2`

  Indicates that the correlation between the exogenous variables VARIABLE_NAME1 and VARIABLE_NAME2, or the correlation of the observation errors/measurement errors associated with endogenous observed variables VARIABLE_NAME1 and VARIABLE_NAME2, is to be estimated. Note that correlations set by previous `shocks` blocks or estimation commands are kept at their value set prior to estimation if they are not estimated again subsequently. Thus, the treatment is the same as in the case of deep parameters set during model calibration and not estimated.

- `skew VARIABLE_NAME`

  Indicates that the skewness coefficient of the exogenous variable VARIABLE_NAME is to be estimated. Dynare subsequently assumes that the shocks are skew normally distributed. Estimating coskewness is currently not supported.

- `PARAMETER_NAME`

  The name of a model parameter to be estimated

- `DSGE_PRIOR_WEIGHT`

  Special name for the weigh of the DSGE model in DSGE-VAR model.

The rest of the line consists of the following fields, some of them being optional:

**INITIAL_VALUE**

Specifies a starting value for the posterior mode optimizer or the maximum likelihood estimation. If unset, defaults to the prior mean.

**LOWER_BOUND**

Specifies a lower bound for the parameter value in maximum likelihood estimation. In a Bayesian estimation context, sets a lower bound only effective while maximizing the posterior kernel. This lower bound does not modify the shape of the prior density, and is only aimed at helping the optimizer in identifying the posterior mode (no consequences for the MCMC). For some prior densities (namely inverse gamma, gamma, uniform, beta or Weibull) it is possible to shift the support of the prior distributions to the left or the right using *prior_3rd_parameter*. In this case the prior density is effectively modified (note that the truncated Gaussian density is not implemented in Dynare). If unset, defaults to minus infinity (ML) or the natural lower bound of the prior (Bayesian estimation). Lower bounds for estimated `corr` parameters are automatically set to -1, while lower bounds for estimated `skew` parameters are automatically set to -0.995 (theoretical bound for skew normal distribution).

**UPPER_BOUND**

Same as `lower_bound`, but specifying an upper bound instead. Upper bounds for estimated `corr` parameters are automatically set to 1, while upper bounds for estimated `skew` parameters are automatically set to 0.995 (theoretical bound for skew normal distribution).

**PRIOR_SHAPE**

A keyword specifying the shape of the prior density. The possible values are: `beta_pdf`, `gamma_pdf`, `normal_pdf`, `uniform_pdf`, `inv_gamma_pdf`, `inv_gamma1_pdf`, `inv_gamma2_pdf` and `weibull_pdf`. Note that `inv_gamma_pdf` is equivalent to `inv_gamma1_pdf`.

**PRIOR_MEAN**

The mean of the prior distribution.

**PRIOR_STANDARD_ERROR**

The standard error of the prior distribution.

**PRIOR_3RD_PARAMETER**

A third parameter of the prior used for generalized beta distribution, generalized gamma, generalized Weibull, the truncated normal, and for the uniform distribution. Default: `-Inf` for normal distribution, `0` otherwise.

**PRIOR_4TH_PARAMETER**

A fourth parameter of the prior used for generalized beta distribution, the truncated normal, and for the uniform distribution. Default: `Inf` for normal distribution, `1` otherwise.

**SCALE_PARAMETER**

A parameter specific scale parameter for the jumping distribution's covariance matrix of the Metropolis-Hasting algorithm. If no `SCALE_PARAMETER` parameter is specified, *mh_jscale* is used for all parameters. If *mh_jscale* isn't set, the default value of `0.2` applies for all parameters. If `mode_compute=6` is used or the `posterior_sampler_option` called `scale_file` is specified, the values set in *estimated_params* will be overwritten.

Note that INITIAL_VALUE, LOWER_BOUND, UPPER_BOUND, PRIOR_MEAN, PRIOR_STANDARD_ERROR, PRIOR_3RD_PARAMETER, PRIOR_4TH_PARAMETER and SCALE_PARAMETER can be any valid EXPRESSION. Some of them can be empty, in which Dynare will select a default value depending on the context and the prior shape.

In case of the uniform distribution, it can be specified either by providing an upper and a lower bound using *PRIOR_3RD_PARAMETER* and *PRIOR_4TH_PARAMETER* or via mean and standard deviation using *PRIOR_MEAN*, *PRIOR_STANDARD_ERROR*. The other two will automatically be filled out. Note that providing both sets of hyperparameters will yield an error message.

As one uses options more towards the end of the list, all previous options must be filled: for example, if you want to specify SCALE_PARAMETER, you must specify `PRIOR_3RD_PARAMETER` and `PRIOR_4TH_PARAMETER`. Use empty values, if these parameters don't apply.

*Example*

```
corr eps_1, eps_2, 0.5,  ,  , beta_pdf, 0, 0.3, -1, 1;
```

Sets a generalized beta prior for the correlation between `eps_1` and `eps_2` with mean `0` and variance `0.3`. By setting `PRIOR_3RD_PARAMETER` to `-1` and `PRIOR_4TH_PARAMETER` to `1` the standard beta distribution with support `[0,1]` is changed to a generalized beta with support `[-1, 1]`. Note that LOWER_BOUND and UPPER_BOUND are left empty and thus default to `-1` and `1`, respectively. The initial value is set to `0.5`.

*Example*

```
corr eps_1, eps_2, 0.5,  -0.5,  1, beta_pdf, 0, 0.3, -1, 1;
```

Sets the same generalized beta distribution as before, but now truncates this distribution to `[-0.5,1]` through the use of LOWER_BOUND and UPPER_BOUND.

*Parameter transformation*

Sometimes, it is desirable to estimate a transformation of a parameter appearing in the model, rather than the parameter itself. It is of course possible to replace the original parameter by a function of the estimated parameter everywhere is the model, but it is often unpractical.

In such a case, it is possible to declare the parameter to be estimated in the parameters statement and to define the transformation, using a pound sign (#) expression (see *Model declaration*).

*Example*

```
parameters bet;

model;
# sig = 1/bet;
c = sig*c(+1)*mpk;
end;

estimated_params;
bet, normal_pdf, 1, 0.05;
end;
```

It is possible to have several *estimated_params* blocks. By default, subsequent blocks are concatenated with the previous ones; this can be useful when building models in a modular fashion (see also *estimated_params_remove* for that use case). However, if an *estimated_params* block has the `overwrite` option, its contents become the new list of estimated parameters, cancelling previous blocks; this can be useful when doing several estimations in a single `.mod` file.

**Block:** **estimated_params_init** ;

**Block:** **estimated_params_init**(OPTIONS...);

This block declares numerical initial values for the optimizer when these are different from the prior mean. It should be specified after the *estimated_params* block as otherwise the specified starting values are overwritten by the latter.

Each line has the following syntax:

```
stderr VARIABLE_NAME | corr VARIABLE_NAME_1, VARIABLE_NAME_2 | skew VARIABLE_
↪NAME | PARAMETER_NAME, INITIAL_VALUE;
```

*Options*

**use_calibration**

For not specifically initialized parameters, use the deep parameters and the elements of the covariance matrix specified in the `shocks` block from calibration as starting values for estimation. For components of the `shocks` block that were not explicitly specified during calibration or which violate the prior, the prior mean is used.

See *estimated_params*, for the meaning and syntax of the various components.

**Block:** **estimated_params_bounds** ;

This block declares lower and upper bounds for parameters in maximum likelihood estimation.

Each line has the following syntax:

```
stderr VARIABLE_NAME | corr VARIABLE_NAME_1, VARIABLE_NAME_2 | skew VARIABLE_
↪NAME | PARAMETER_NAME, LOWER_BOUND, UPPER_BOUND;
```

See *estimated_params*, for the meaning and syntax of the various components.

**Block:** **estimated_params_remove** ;

This block partially undoes the effect of a previous *estimated_params* block, by removing some parameters from the estimation.

Each line has the following syntax:

```
stderr VARIABLE_NAME | corr VARIABLE_NAME_1, VARIABLE_NAME_2 | skew␣
↪VARIABLE_NAME | PARAMETER_NAME;
```

*"Endogenous" prior restrictions*

It is also possible to impose implicit "endogenous" priors about IRFs and moments on the model during estimation. For example, one can specify that all valid parameter draws for the model must generate fiscal multipliers that are bigger than 1 by specifying how the IRF to a government spending shock must look like. The prior restrictions can be imposed via `irf_calibration` and `moment_calibration` blocks (see *IRF/Moment calibration*). The way it works internally is that any parameter draw that is inconsistent with the "calibration" provided in these blocks is discarded, i.e. assigned a prior density of 0. When specifying these blocks, it is important to keep in mind that one won't be able to easily do *model_comparison* in this case, because the prior density will not integrate to 1.

### 4.16.3 The estimation command

**Command:** **estimation** [VARIABLE_NAME...];

**Command:** **estimation**(OPTIONS...) [VARIABLE_NAME...];

This command runs Bayesian or maximum likelihood estimation.

The following information will be displayed by the command:

- Results from posterior optimization (also for maximum likelihood)
- Marginal log data density
- Posterior mean and highest posterior density interval (shortest credible set) from posterior simulation
- Convergence diagnostic table when only one MCM chain is used or Metropolis-Hastings convergence graphs documented in Pfeifer (2014) in case of multiple MCM chains
- Table with numerical inefficiency factors of the MCMC
- Graphs with prior, posterior, and mode
- Graphs of smoothed shocks, smoothed observation errors, smoothed and historical variables

Note that the posterior moments, smoothed variables, k-step ahead filtered variables and forecasts (when requested) will only be computed on the variables listed after the *estimation* command. Alternatively, one can choose to compute these quantities on all endogenous or on all observed variables (see `consider_all_endogenous`, `consider_all_endogenous_and_auxiliary`, and `consider_only_observed` options in *Variable selection for posterior objects*). If no variable is listed after the estimation command, then Dynare will interactively ask which variable set to use.

Also, during the MCMC (Bayesian estimation with *mh_replic* $> 0$) a (graphical or text) waiting bar is displayed showing the progress of the Monte Carlo and the current value of the acceptance ratio. Note that if the *load_mh_file* option is used (see below) the reported acceptance ratio does not take into account the draws from the previous MCMC. In the literature there is a general agreement for saying that the acceptance ratio should be close to one third or one quarter. If this not the case, you can stop the MCMC (`Ctrl-C`) and change the value of option *mh_jscale* (see below).

Note that by default Dynare generates random numbers using the algorithm `mt199937ar` (i.e. Mersenne Twister method) with a seed set equal to `0`. Consequently, the MCMCs in Dynare are deterministic: one will get exactly the same results across different Dynare runs (*ceteris paribus*). For instance, the posterior moments or posterior densities will be exactly the same. This behaviour allows to easily identify the consequences of a change on the model, the priors or the estimation options. But one may also want to check that across multiple runs, with different sequences of proposals, the returned results are almost identical. This should be true if the number of iterations (i.e. the value of `mh_replic`) is important enough to ensure the convergence of the MCMC to its ergodic distribution. In this case the default behaviour of the random number generators in not wanted, and the user should set the seed according to the system clock before the estimation command using the following command:

```
set_dynare_seed('clock');
```

so that the sequence of proposals will be different across different runs.

Finally, Dynare does not always properly distinguish between maximum likelihood and Bayesian estimation in its field names. While there is an important conceptual distinction between frequentist confidence intervals and Bayesian highest posterior density intervals (HPDI) as well as between posterior density and likelilhood, Dynare sometimes uses the Bayesian terms as a stand-in in its display of maximum likelihood results. An example is the storage of the output of the *forecast* option of *estimation* with ML, which will use `HPDinf/HPDsup` to denote the confidence interval.

*Algorithms*

The Monte Carlo Markov Chain (MCMC) diagnostics are generated by the estimation command if *mh_replic* is larger than 2000 and if option *nodiagnostic* is not used. By default, the convergence diagnostics (see *Convergence diagnostics*) of Geweke (1992), Geweke (1999) is computed for each chain. It uses a chi-square test to compare the means of the first and last draws specified by *geweke_interval* after

discarding the burn-in of *mh_drop*. The test is computed using variance estimates under the assumption of no serial correlation as well as using tapering windows specified in *taper_steps*. If *mh_nblocks* is larger than 1, the convergence diagnostics of Brooks and Gelman (1998) are also provided. As described in section 3 of Brooks and Gelman (1998) the univariate convergence diagnostics are based on comparing pooled and within MCMC moments (Dynare displays the second and third order moments, and the length of the Highest Probability Density interval covering 80% of the posterior distribution). Due to computational reasons, the multivariate convergence diagnostic does not follow Brooks and Gelman (1998) strictly, but rather applies their idea for univariate convergence diagnostics to the range of the posterior likelihood function instead of the individual parameters. The posterior kernel is used to aggregate the parameters into a scalar statistic whose convergence is then checked using the Brooks and Gelman (1998) univariate convergence diagnostic.

The inefficiency factors are computed as in Giordani *et al.* (2011) based on Parzen windows as in e.g. Andrews (1991).

#### 4.16.3.1 Data treatment options

One of the `datafile` and `dataseries` options has to be specified (unless a *data* command is present). Both cannot be used simultaneously.

**`datafile = FILENAME`**

The datafile: a `.m` file, a `.mat` file, a `.csv` file, or a `.xls`/`.xlsx` file (under Octave, the io package is required for the `.xlsx` format, and the `.xls` file extension is not supported). Note that the base name (i.e. without extension) of the datafile has to be different from the base name of the model file. If there are several files named `FILENAME`, but with different file endings, the file name must be included in quoted strings and provide the file ending like:

```
estimation(datafile='../fsdat_simul.mat',...);
```

The format of the file is described in the *dseries* constructor that takes a filename. In particular, when using the `.csv`, `.xls` or `.xlsx` format, it is possible to specify dates in the first column of the file.

**`dataseries = DSERIES_NAME`**

The name of a *dseries* object available in the workspace containing the data series to be used.

**`dirname = FILENAME`**

Directory in which to store *estimation* output. To pass a subdirectory of a directory, you must quote the argument. Default: `<mod_file>`.

**`xls_sheet = QUOTED_STRING`**

The name of the sheet with the data in an Excel file. Default: the first sheet.

**`xls_range = RANGE`**

The range with the data in an Excel file, given by its top-left and bottom-right cells. For example, `xls_range=B2:D200`. The first row of the range must contain the names of the variables, and its first column may contain the dates (see the *dseries* constructor that takes a filename). Default: the whole sheet.

**`nobs = INTEGER`**

The number of observations following *first_obs* to be used. Default: all observations in the file or dseries object after `first_obs`.

**`nobs = [INTEGER1:INTEGER2]`**

Runs a recursive estimation and forecast for samples of size ranging of `INTEGER1` to `INTEGER2`. Option `forecast` must also be specified. The forecasts are stored in the `RecursiveForecast` field of the results structure (see *RecursiveForecast*). The respective results structures `oo_` are saved in `oo_recursive_` (see *oo_recursive_*) and are indexed with the respective sample length.

**`first_obs = INTEGER | DATE`**

The number or the date of the first observation to be used. In case of estimating a DSGE-VAR, `first_obs` needs to be larger than the number of lags. Default: 1.

**first_obs = [INTEGER1:INTEGER2]**

Runs a rolling window estimation and forecast for samples of fixed size `nobs` starting with the first observation ranging from `INTEGER1` to `INTEGER2`. Option `forecast` must also be specified. This option is incompatible with requesting recursive forecasts using an expanding window (see *nobs*). The respective results structures `oo_` are saved in `oo_recursive_` (see *oo_recursive_*) and are indexed with the respective first observation of the rolling window.

**last_obs = INTEGER | DATE**

The number or the date of the last observation to be used. Default: `first_obs+nobs-1` if `nobs` is set, otherwise the last observation in the file or dseries object.

**prefilter = INTEGER**

A value of 1 means that the estimation procedure will demean each data series by its empirical mean. If the *loglinear* option without the *logdata* option is requested, the data will first be logged and then demeaned. Default: `0`, i.e. no prefiltering.

**presample = INTEGER**

The number of observations after *first_obs* to be skipped before evaluating the likelihood. These presample observations do not enter the likelihood, but are used as a training sample for starting the Kalman filter iterations. This option is incompatible with estimating a DSGE-VAR. Default: `0`.

**loglinear**

Computes a log-linear approximation of the model instead of a linear approximation. As always in the context of estimation, the data must correspond to the definition of the variables used in the model (see Pfeifer (2013) for more details on how to correctly specify observation equations linking model variables and the data). If you specify the `loglinear` option, Dynare will take the logarithm of both your model variables and of your data as it assumes the data to correspond to the original non-logged model variables. The displayed posterior results like impulse responses, smoothed variables, and moments will be for the logged variables, not the original un-logged ones. Default: computes a linear approximation.

**logdata**

Dynare applies the *log* transformation to the provided data if a log-linearization of the model is requested (*loglinear*) unless `logdata` option is used. This option is necessary if the user provides data already in logs, otherwise the *log* transformation will be applied twice (this may result in complex data).

#### 4.16.3.2 Graph output options

**plot_priors = INTEGER**

Control the plotting of priors.

`0`

No prior plot.

1

Prior density for each estimated parameter is plotted. It is important to check that the actual shape of prior densities matches what you have in mind. Ill-chosen values for the prior standard density can result in absurd prior densities.

Default value is 1.

**nograph**

See *nograph*.

**posterior_nograph**

Suppresses the generation of graphs associated with Bayesian IRFs (*bayesian_irf*), posterior smoothed objects (*smoother*), and posterior forecasts (*forecast*).

**posterior_graph**

Re-enables the generation of graphs previously shut off with *posterior_nograph*.

**nodisplay**

See *nodisplay*.

**graph_format = FORMAT**

**graph_format = ( FORMAT, FORMAT... )**

See *graph_format*.

**tex**

See *tex*.

#### 4.16.3.3 Mode computing options

**mode_compute = INTEGER | FUNCTION_NAME**

Specifies the optimizer for the mode computation:

0

The mode isn't computed. When the *mode_file = FILENAME* option is specified, the mode is simply read from that file.

When the *mode_file = FILENAME* option is not specified, Dynare reports the value of the log posterior (log likelihood) evaluated at the initial value of the parameters.

When *mode_file = FILENAME* is not specified and there is no *estimated_params* block, but the *smoother* option is used, it is a roundabout way to compute the smoothed value of the variables of a model with calibrated parameters.

1

Uses `fmincon` optimization routine (available under MATLAB if the Optimization Toolbox is installed; available under Octave if the optim package, version 1.6 or above, is installed).

2

Uses the continuous simulated annealing global optimization algorithm described in Corana *et al.* (1987) and Goffe *et al.* (1994).

3

Uses `fminunc` optimization routine (available under MATLAB if the Optimization Toolbox is installed; available under Octave if the optim package is installed).

4

Uses Chris Sims's `csminwel`.

5

Uses Marco Ratto's `newrat`. This value is not compatible with non-linear filters or DSGE-VAR models. This is a slice optimizer: most iterations are a sequence of univariate optimization step, one for each estimated parameter or shock. Uses `csminwel` for line search in each step.

6

Uses a Monte Carlo based optimisation routine (see https://git.dynare.org/Dynare/dynare/-/wikis/mode-compute-6 for more details).

7

Uses `fminsearch`, a simplex-based optimization routine (available under MATLAB if the Optimization Toolbox is installed; available under Octave if the optim package is installed).

`8`

Uses Dynare implementation of the Nelder-Mead simplex-based optimization routine (generally more efficient than the MATLAB or Octave implementation available with `mode_compute=7`).

`9`

Uses the CMA-ES (Covariance Matrix Adaptation Evolution Strategy) algorithm of Hansen and Kern (2004), an evolutionary algorithm for difficult non-linear non-convex optimization.

`10`

Uses the `simpsa` algorithm, based on the combination of the non-linear simplex and simulated annealing algorithms as proposed by Cardoso *et al.* (1996).

`11`

Currently not in use. The Liu and West (2001) filter that used to be available under this option value is now triggered with `posterior_sampling_method='online'`.

`12`

Uses the `particleswarm` optimization routine (available under MATLAB if the Global Optimization Toolbox is installed; not available under Octave).

`13`

Uses the `lsqnonlin` non-linear least squares optimization routine (available under MATLAB if the Optimization Toolbox is installed; available under Octave if the optim package is installed). Only supported for `method_of_moments`.

`101`

Uses the `SolveOpt` algorithm for local nonlinear optimization problems proposed by Kuntsevich and Kappel (1997).

`102`

Uses `simulannealbnd` optimization routine (available under MATLAB if the Global Optimization Toolbox is installed; not available under Octave)

`FUNCTION_NAME`

It is also possible to give a FUNCTION_NAME to this option, instead of an INTEGER. In that case, Dynare takes the return value of that function as the posterior mode.

Default value is `5`.

**`mode_file = FILENAME`**

Name of the file containing previous value for the mode. When computing the mode, Dynare stores the mode (`xparam1`) and the hessian (`hh`, only if `cova_compute=1`) in a file called `MODEL_FILENAME_mode.mat` in the `FILENAME/Output` folder. After a successful run of the estimation command, the `mode_file` will be disabled to prevent other function calls from implicitly using an updated mode file. Thus, if the `.mod` file contains subsequent *`estimation`* commands, the `mode_file` option, if desired, needs to be specified again.

**`additional_optimizer_steps = [INTEGER]`**

**`additional_optimizer_steps = [INTEGER1:INTEGER2]`**

**additional_optimizer_steps = [INTEGER1 INTEGER2 ...]**

Vector of additional minimization algorithms run after `mode_compute`. Default: no additional optimization iterations.

**optim = (NAME, VALUE, ...)**

A list of NAME and VALUE pairs. Can be used to set options for the optimization routines. The set of available options depends on the selected optimization routine (i.e. on the value of option *mode_compute*):

`1, 3, 7, 12, 13`

Available options are given in the documentation of the MATLAB Optimization Toolbox or in Octave's documentation.

`2`

Available options are:

`'initial_step_length'`

Initial step length. Default: `1`.

`'initial_temperature'`

Initial temperature. Default: `15`.

`'MaxIter'`

Maximum number of function evaluations. Default: `100000`.

`'neps'`

Number of final function values used to decide upon termination. Default: `10`.

`'ns'`

Number of cycles. Default: `10`.

`'nt'`

Number of iterations before temperature reduction. Default: `10`.

`'step_length_c'`

Step length adjustment. Default: `0.1`.

`'TolFun'`

Stopping criteria. Default: `1e-8`.

`'rt'`

Temperature reduction factor. Default: `0.1`.

`'verbosity'`

Controls verbosity of display during optimization, ranging from `0` (silent) to `3` (each function evaluation). Default: `1`

`4`

Available options are:

`'InitialInverseHessian'`

Initial approximation for the inverse of the Hessian matrix of the posterior kernel (or likelihood). Obviously this approximation has to be a square, positive definite and symmetric matrix. Default: `'1e-4*eye(nx)'`, where nx is the number of parameters to be estimated.

`'MaxIter'`

Maximum number of iterations. Default: `1000`.

`'NumgradAlgorithm'`

Possible values are `2`, `3` and `5`, respectively, corresponding to the two, three and five points formula used to compute the gradient of the objective function (see Abramowitz and Stegun, 1965). Values `13` and `15` are more experimental. If perturbations on the right and the left increase the value of the objective function (we minimize this function) then we force the corresponding element of the gradient to be zero. The idea is to temporarily reduce the size of the optimization problem. Default: `2`.

`'NumgradEpsilon'`

Size of the perturbation used to compute numerically the gradient of the objective function. Default: `1e-6`.

`'TolFun'`

Stopping criteria. Default: `1e-7`.

`'verbosity'`

Controls verbosity of display during optimization. Set to `0` to set to silent. Default: `1`.

`'SaveFiles'`

Controls saving of intermediate results during optimization. Set to `0` to shut off saving. Default: `1`.

`5`

Available options are:

`'Hessian'`

Triggers three types of Hessian computations. `0`: outer product gradient; `1`: default Dynare Hessian routine; `2`: 'mixed' outer product gradient, where diagonal elements are obtained using second-order derivation formula and outer product is used for correlation structure. Both {0} and {2} options require univariate filters, to ensure using maximum number of individual densities and a positive definite Hessian. Both {0} and {2} are quicker than default Dynare numeric Hessian, but provide decent starting values for Metropolis for large models (option {2} being more accurate than {0}). Default: `1`.

`'MaxIter'`

Maximum number of iterations. Default: `1000`.

`'TolFun'`

Stopping criteria. Default: `1e-5` for numerical derivatives, `1e-7` for analytic derivatives.

`'robust'`

Trigger more robust but computationally more expensive line search. Default: `false`.

`'TolGstep'`

Tolerance parameter used for tuning gradient step. Default: same value as `TolFun`.

`'TolGstepRel'`

Parameter used for tuning gradient step, governing the tolerance relative to the functions value. Default: not triggered.

`'verbosity'`

Controls verbosity of display during optimization. Set to `0` to set to silent. Default: `1`.

`'SaveFiles'`

Controls saving of intermediate results during optimization. Set to `0` to shut off saving. Default: `1`.

6

Available options are:

`'AcceptanceRateTarget'`

A real number between zero and one. The scale parameter of the jumping distribution is adjusted so that the effective acceptance rate matches the value of option `'AcceptanceRateTarget'`. Default: `1.0/3.0`.

`'InitialCovarianceMatrix'`

Initial covariance matrix of the jumping distribution. It is also used to initialize the covariance matrix during recursive updating. Default is `'previous'` if option *mode_file = FILENAME* is used, `'prior'` otherwise. The user can also specify `'identity'`, which will use an identity matrix with a diagonal of 0.1.

`'nclimb-mh'`

Number of iterations in the last MCMC (climbing mode). Default: `200000`.

`'ncov-mh'`

Number of iterations used for updating the covariance matrix of the jumping distribution. Default: `20000`.

`'nscale-mh'`

Maximum number of iterations used for adjusting the scale parameter of the jumping distribution. Default: `200000`.

`'NumberOfMh'`

Number of MCMC run sequentially. Default: `3`.

8

Available options are:

`'InitialSimplexSize'`

Initial size of the simplex, expressed as percentage deviation from the provided initial guess in each direction. Default: `.05`.

`'MaxIter'`

Maximum number of iterations. Default: `5000`.

`'MaxFunEvals'`

Maximum number of objective function evaluations. No default.

`'MaxFunEvalFactor'`

Set `MaxFunEvals` equal to `MaxFunEvalFactor` times the number of estimated parameters. Default: `500`.

'TolFun'

Tolerance parameter (w.r.t the objective function). Default: 1e-4.

'TolX'

Tolerance parameter (w.r.t the instruments). Default: 1e-4.

'verbosity'

Controls verbosity of display during optimization. Set to 0 to set to silent. Default: 1.

9

Available options are:

'CMAESResume'

Resume previous run. Requires the variablescmaes.mat from the last run. Set to 1 to enable. Default: 0.

'MaxIter'

Maximum number of iterations.

'MaxFunEvals'

Maximum number of objective function evaluations. Default: Inf.

'TolFun'

Tolerance parameter (w.r.t the objective function). Default: 1e-7.

'TolX'

Tolerance parameter (w.r.t the instruments). Default: 1e-7.

'verbosity'

Controls verbosity of display during optimization. Set to 0 to set to silent. Default: 1.

'SaveFiles'

Controls saving of intermediate results during optimization. Set to 0 to shut off saving. Default: 1.

10

Available options are:

'EndTemperature'

Terminal condition w.r.t the temperature. When the temperature reaches EndTemperature, the temperature is set to zero and the algorithm falls back into a standard simplex algorithm. Default: 0.1.

'MaxIter'

Maximum number of iterations. Default: 5000.

'MaxFunEvals'

Maximum number of objective function evaluations. No default.

'TolFun'

Tolerance parameter (w.r.t the objective function). Default: 1e-4.

'TolX'

Tolerance parameter (w.r.t the instruments). Default: 1e-4.

'verbosity'

Controls verbosity of display during optimization. Set to `0` to set to silent. Default: `1`.

`101`

Available options are:

`'LBGradientStep'`

Lower bound for the step size used for the difference approximation of gradients. Default: `1e-11`.

`'MaxIter'`

Maximum number of iterations. Default: `15000`

`'SpaceDilation'`

Coefficient of space dilation. Default: `2.5`.

`'TolFun'`

Tolerance parameter (w.r.t the objective function). Default: `1e-6`.

`'TolX'`

Tolerance parameter (w.r.t the instruments). Default: `1e-6`.

`'verbosity'`

Controls verbosity of display during optimization. Set to `0` to set to silent. Default: `1`.

`102`

Available options are given in the documentation of the MATLAB Global Optimization Toolbox.

*Example*

To change the defaults of `csminwel` (`mode_compute=4`):

```
estimation(..., mode_compute=4,optim=('NumgradAlgorithm',3,
↪'TolFun',1e-5),...);
```

**`silent_optimizer`**

Instructs Dynare to run mode computing/optimization silently without displaying results or saving files in between. Useful when running loops.

#### 4.16.3.4 Hessian/derivative options

**`cova_compute = INTEGER`**

When `0`, the covariance matrix of estimated parameters is not computed after the computation of posterior mode (or maximum likelihood). This increases speed of computation in large models during development, when this information is not always necessary. Of course, it will break all successive computations that would require this covariance matrix. Otherwise, if this option is equal to `1`, the covariance matrix is computed and stored in variable `hh` of `MODEL_FILENAME_mode.mat`. Default is `1`.

**`use_penalized_objective_for_hessian`**

Use the penalized objective instead of the objective function to compute numerically the hessian matrix at the mode. The penalties decrease the value of the posterior density (or likelihood) when, for some perturbations, Dynare is not able to solve the model (issues with steady-state existence, Blanchard and Kahn conditions, ...). In practice, the penalized and original objectives will only differ if the posterior mode is found to be near a region where the model is ill-behaved. By default, the original objective function is used.

**analytic_derivation**

Triggers estimation with analytic gradient at `order=1`. The final hessian at the mode is also computed analytically. Only works for stationary models without missing observations, i.e. for `kalman_algo<3`. Incompatible with *heteroskedastic_shocks*. Optimizers that rely on analytic gradients are `mode_compute=1,3,4,5,101`.

#### 4.16.3.5 Mode-check diagnostics

**mode_check**

Tells Dynare to plot the posterior density for values around the computed mode for each estimated parameter in turn. This is helpful to diagnose problems with the optimizer. Note that for `order>1` the likelihood function resulting from the particle filter is not differentiable anymore due to the resampling step. For this reason, the `mode_check` plot may look wiggly.

**mode_check_neighbourhood_size = DOUBLE**

Used in conjunction with option `mode_check`, gives the width of the window around the posterior mode to be displayed on the diagnostic plots. This width is expressed in percentage deviation. The `Inf` value is allowed, and will trigger a plot over the entire domain (see also `mode_check_symmetric_plots`). Default:`0.5`.

**mode_check_symmetric_plots = INTEGER**

Used in conjunction with option `mode_check`, if set to `1`, tells Dynare to ensure that the check plots are symmetric around the posterior mode. A value of `0` allows to have asymmetric plots, which can be useful if the posterior mode is close to a domain boundary, or in conjunction with `mode_check_neighbourhood_size = Inf` when the domain in not the entire real line. Default: `1`.

**mode_check_number_of_points = INTEGER**

Number of points around the posterior mode where the posterior kernel is evaluated (for each parameter). Default is `20`.

#### 4.16.3.6 Kalman filtering and likelihood options

**no_init_estimation_check_first_obs**

Do not check for stochastic singularity in first period. If used, *ESTIMATION CHECKS* does not return an error if the check fails only in first observation. This should only be used when observing stock variables (e.g. capital) in first period, on top of their associated flow (e.g. investment). Using this option may lead to a crash or provide undesired/wrong results for badly specified problems (e.g. the additional variable observed in first period is not predetermined).

For advanced use only.

**lik_init = INTEGER**

Type of initialization of Kalman filter:

`1`

For stationary models, the initial matrix of variance of the error of forecast is set equal to the unconditional variance of the state variables.

`2`

For nonstationary models: a wide prior is used with an initial matrix of variance of the error of forecast diagonal with 10 on the diagonal (follows the suggestion of Harvey and PHILLIPS (1979)).

`3`

For nonstationary models: use a diffuse filter (use rather the `diffuse_filter` option).

`4`

The filter is initialized with the fixed point of the Riccati equation.

5

Use i) option 2 for the non-stationary elements by setting their initial variance in the forecast error matrix to 10 on the diagonal and all covariances to 0 and ii) option 1 for the stationary elements.

Default value is 1. For advanced use only.

**conditional_likelihood**

Do not use the Kalman filter to evaluate the likelihood, but instead evaluate the conditional likelihood, based on the first-order reduced form of the model, by assuming that the initial state vector is at its steady state. This approach requires that:

1. The number of structural innovations is equal to the number of observed variables.
2. The absence of measurement errors (as introduced by the Dynare interface, see documentation about the *estimated_params* block).
3. The absence of missing observations.

The evaluation of the conditional likelihood is faster and more stable than the evaluation of the likelihood with the Kalman filter. Also, this approach does not require special treatment for models with unit roots. Note however that the conditional likelihood is sensitive to the choice for the initial condition, which can be an issue if the data are initially far from the steady state. This option is not compatible with `analytic_derivation`.

**kalman_algo = INTEGER**

0

Automatically use the Multivariate Kalman Filter for stationary models and the Multivariate Diffuse Kalman Filter for non-stationary models.

1

Use the Multivariate Kalman Filter.

2

Use the Univariate Kalman Filter.

3

Use the Multivariate Diffuse Kalman Filter.

4

Use the Univariate Diffuse Kalman Filter.

5

Use the Pruned Skewed Kalman Filter.

Default value is `0`. In case of missing observations of single or all series, Dynare treats those missing values as unobserved states and uses the Kalman filter to infer their value (see e.g. Durbin and Koopman (2012), Ch. 4.10) This procedure has the advantage of being capable of dealing with observations where the forecast error variance matrix becomes singular for some variable(s). If this happens, the respective observation enters with a weight of zero in the log-likelihood, i.e. this observation for the respective variable(s) is dropped from the likelihood computations (for details see Durbin and Koopman (2012), Ch. 6.4 and 7.2.5 and Koopman and Durbin (2000)). If the use of a multivariate Kalman filter is specified and a singularity is encountered, Dynare by default automatically switches to the univariate Kalman filter for this parameter draw. This behavior can be changed via the *use_univariate_filters_if_singularity_is_detected* option. In case of skew normally distributed shocks, the Pruned Skewed Kalman filter of Guljanov *et al.* (forthcoming) can be used by setting `kalman_algo=5`. This filter is currently not compatible with missing observations and does not switch to a univariate filter in case of singularity.

**fast_kalman_filter**

Select the fast Kalman filter using Chandrasekhar recursions as described by Herbst (2015). This setting is only used with `kalman_algo=1` or `kalman_algo=3`. In case of using the diffuse Kalman filter (`kalman_algo=3/lik_init=3`), the observables must be stationary. This option is neither compatible with *analytic_derivation* nor *heteroskedastic_shocks*.

**kalman_tol = DOUBLE**

Numerical tolerance for determining the singularity of the covariance matrix of the prediction errors during the Kalman filter (minimum allowed reciprocal of the matrix condition number). Default value is `1e-10`.

**diffuse_kalman_tol = DOUBLE**

Numerical tolerance for determining the singularity of the covariance matrix of the prediction errors ($F_\infty$) and the rank of the covariance matrix of the non-stationary state variables ($P_\infty$) during the Diffuse Kalman filter. Default value is `1e-6`.

**filter_covariance**

Saves the series of one step ahead error of forecast covariance matrices. With Metropolis, they are saved in *oo_.FilterCovariance*, otherwise in *oo_.Smoother.Variance*. Saves also k-step ahead error of forecast covariance matrices if `filter_step_ahead` is set. This option is not yet compatible with the pruned skewed Kalman filter (`kalman_algo=5`).

**filter_step_ahead = [INTEGER1:INTEGER2]**

**filter_step_ahead = [INTEGER1 INTEGER2 ...]**

Triggers the computation k-step ahead filtered values, i.e. $E_t y_{t+k}$. Stores results in `oo_.FilteredVariablesKStepAhead`. Also stores 1-step ahead values in `oo_.FilteredVariables`. `oo_.FilteredVariablesKStepAheadVariances` is stored if `filter_covariance`. This option is not yet compatible with the pruned skewed Kalman filter (`kalman_algo=5`).

**filter_decomposition**

Triggers the computation of the shock decomposition of the above k-step ahead filtered values. Stores results in `oo_.FilteredVariablesShockDecomposition`. This option is not yet compatible with the pruned skewed Kalman filter (`kalman_algo=5`).

**diffuse_filter**

Uses the diffuse Kalman filter (as described in Durbin and Koopman (2012) and Koopman and Durbin (2003) for the multivariate and Koopman and Durbin (2000) for the univariate filter) to estimate models with non-stationary observed variables. This option will also reset the `qz_criterium` to count unit root variables towards the stable variables. Trying to estimate a model with unit roots will otherwise result in a Blanchard-Kahn error.

When `diffuse_filter` is used the `lik_init` option of `estimation` has no effect.

When there are nonstationary exogenous variables in a model, there is no unique deterministic steady state. For instance, if productivity is a pure random walk:

$$a_t = a_{t-1} + e_t$$

any value of $\bar{a}$ of $a$ is a deterministic steady state for productivity. Consequently, the model admits an infinity of steady states. In this situation, the user must help Dynare in selecting one steady state, except if zero is a trivial model's steady state, which happens when the `linear` option is used in the model declaration. The user can either provide the steady state to Dynare using a *steady_state_model* block (or writing a steady-state file) if a closed form solution is available, see *steady_state_model*, or specify some constraints on the steady state, see *equation_tag_for_conditional_steady_state* so that Dynare computes the steady state conditionally on some predefined levels for the non-stationary variables. In both cases, the idea is to use dummy values for the steady-state level of the exogenous non-stationary variables.

Note that the nonstationary variables in the model must be integrated processes (their first difference or k-difference must be stationary).

**heteroskedastic_filter**

Runs filter, likelihood, and smoother using heteroskedastic definitions provided in a *heteroskedastic_shocks* block. This option is not yet compatible with the pruned skewed Kalman filter (`kalman_algo=5`).

**use_univariate_filters_if_singularity_is_detected = INTEGER**

Decide whether Dynare should automatically switch to univariate filter if a singularity is encountered in the likelihood computation (this is the behaviour if the option is equal to `1`). Alternatively, if the option is equal to `0`, Dynare will not automatically change the filter, but rather use a penalty value for the likelihood when such a singularity is encountered. Default: `1`.

**use_univariate_smoother_if_singularity_is_detected**

**use_univariate_smoother_if_singularity_is_detected = BOOLEAN**

Decide whether Dynare should automatically switch to univariate smoother if a singularity is encountered in the smoother (this is the behaviour if the option is `true`). Otherwise, an error will be thrown. Default: `false`.

**rescale_prediction_error_covariance**

Rescales the prediction error covariance in the Kalman filter to avoid badly scaled matrix and reduce the probability of a switch to univariate Kalman filters (which are slower). By default, no rescaling is done.

**lyapunov = OPTION**

Determines the algorithm used to solve the Lyapunov equation used to initialize the variance-covariance matrix of the Kalman filter. Possible values for OPTION are:

`default`

Uses the default solver for Lyapunov equations based on Bartels-Stewart algorithm.

`fixed_point`

Uses a fixed point algorithm to solve the Lyapunov equation. This method is faster than the `default` one for large scale models, but it could require a large amount of iterations.

`doubling`

Uses a doubling algorithm to solve the Lyapunov equation (`disclyap_fast`). This method is faster than the two previous one for large scale models.

`square_root_solver`

Uses a square-root solver for Lyapunov equations (`dlyapchol`). This method is fast for large scale models (available under MATLAB if the Control System Toolbox is installed; available under Octave if the control package is installed)

Default value is `default`.

**lyapunov_fixed_point_tol = DOUBLE**

This is the convergence criterion used in the fixed point Lyapunov solver. Its default value is `1e-10`.

**lyapunov_doubling_tol = DOUBLE**

This is the convergence criterion used in the doubling algorithm to solve the Lyapunov equation. Its default value is `1e-16`.

#### 4.16.3.7 Skewed Kalman filter options

**skewed_kalman_prune_tol = DOUBLE**

Tolerance for pruning redundant skewness dimensions in closed skew normal distribution during the skewed Kalman filter and smoother. Default is `0.01`. Only relevant for `kalman_algo=5`.

**skewed_kalman_rank_deficiency_transform**

In case of singular transition matrix, compute CSN parameters for joint distribution of states and shocks from state transition equation. Does either rank deficient or full rank linear transformation which might speed up or robustify the pruned skewed Kalman filter. Only relevant for `kalman_algo=5`.

**skewed_kalman_mvnlogcdf = QUOTED_STRING**

Name of function to compute log Gaussian cdf, possible values: `'gaussian_log_mvncdf_mendell_elston'` (default) or `'mvncdf'`. Only relevant for `kalman_algo=5`.

#### 4.16.3.8 Posterior objects

**moments_varendo**

Triggers the computation of the posterior distribution of the theoretical moments of the endogenous variables. Results are stored in `oo_.PosteriorTheoreticalMoments` (see *oo_.PosteriorTheoreticalMoments*). The number of lags in the autocorrelation function is controlled by the `ar` option. Not compatible with OccBin.

**ar = INTEGER**

See *ar*. Only useful in conjunction with option `moments_varendo`.

**contemporaneous_correlation**

See *contemporaneous_correlation*. Results are stored in `oo_.PosteriorTheoreticalMoments`. Note that the `nocorr` option has no effect.

**no_posterior_kernel_density**

Shuts off the computation of the kernel density estimator for the posterior objects (see *density* field).

**conditional_variance_decomposition = INTEGER**

**conditional_variance_decomposition = [INTEGER1:INTEGER2]**

**conditional_variance_decomposition = [INTEGER1 INTEGER2 ...]**

Computes the posterior distribution of the conditional variance decomposition for the specified period(s). The periods must be strictly positive. Conditional variances are given by $var(y_{t+k}|t)$. For period 1, the conditional variance decomposition provides the decomposition of the effects of shocks upon impact. The results are stored in `oo_.PosteriorTheoreticalMoments.dsge.ConditionalVarianceDecomposition`. Note that this option requires the option *moments_varendo* to be specified. In the presence of measurement error, the field will contain the variance contribution after measurement error has been taken out, *i.e.* the decomposition will be conducted of the actual as opposed to the measured variables. The variance decomposition of the measured variables will be stored in `oo_.PosteriorTheoreticalMoments.dsge.ConditionalVarianceDecompositionME`.

**filtered_vars**

Triggers the computation of the posterior distribution of filtered endogenous variables/one-step ahead forecasts, i.e. $E_t y_{t+1}$. Results are stored in `oo_.FilteredVariables` (see below for a description of this variable)

**frequentist_smoother = BOOLEAN**

If set to `true`, the frequentist smoother will be run in cases where the Bayesian smoother usually triggered with *smoother* is not run (e.g. ML estimation or `mh_replic=0`). Default: true.

**smoother**

Triggers the computation of the posterior distribution of smoothed endogenous variables and shocks, i.e. the expected value of variables and shocks given the information available in all observations up to the final date ($E_T y_t$). Results are stored in `oo_.SmoothedVariables`, `oo_.SmoothedShocks` and `oo_.SmoothedMeasurementErrors`. Also triggers the computation of `oo_.UpdatedVariables`, which contains the estimation of the expected value of variables given the information available at the current date ($E_t y_t$). See below for a description of all these variables.

**smoother_redux**

Triggers a faster computation of the smoothed endogenous variables and shocks for large models. It runs the smoother only for the state variables (i.e. with the same representation used for likelihood computations) and computes the remaining variables ex-post. Static unobserved objects (filtered, smoothed, updated, k-step ahead) are recovered, but there are exceptions to a full recovery, depending on how static unobserved variables depend on the restricted state space adopted. For example, lagged shocks which are ONLY used to recover NON-observed static variables will not be recovered). For such exceptions, only the following output is provided:

`FilteredVariablesKStepAhead`: will be fully recovered

**`SmoothedVariables`, `FilteredVariables`, `UpdatedVariables`: recovered for all periods beyond period d+1,**
where d denotes the number of diffuse filtering steps.

`FilteredVariablesKStepAheadVariances`, `Variance`, and `State_uncertainty` cannot be recovered, and ZERO is provided as output.

If you need variances for those variables, either do not set the option, or declare the variable as observed, using `NaN` as data points. Currently not compatible with the pruned skewed Kalman filter (`kalman_algo=5`).

**smoothed_state_uncertainty**

**smoothed_state_uncertainty = BOOLEAN**

Triggers the computation of the variance of smoothed estimates, i.e. $var_T(y_t)$. Stores results in `oo_.Smoother.State_uncertainty`. This option is not yet compatible with the pruned skewed Kalman filter (`kalman_algo=5`).

**smoothed_shock_variance**

**smoothed_shock_variance = BOOLEAN**

Triggers the computation of the variance of smoothed shocks, i.e. $Var(\eta_t|Y_{1:T})$. Without Metropolis, stores the full matrix in `oo_.Smoother.Shock_variance` and per-shock diagonal entries in `oo_.SmoothedShockVariances.EXONAME`. With Metropolis, stores posterior statistics of the full matrix in `oo_.SmoothedShockVariance`. Not compatible with the inversion filter.

**forecast = INTEGER**

Computes the posterior distribution of a forecast on INTEGER periods after the end of the sample used in estimation. If no Metropolis-Hastings is computed, the result is stored in variable `oo_.forecast` and corresponds to the forecast at the posterior mode. If a Metropolis-Hastings is computed, the distribution of forecasts is stored in variables `oo_.PointForecast` and `oo_.MeanForecast`. See *Forecasting*, for a description of these variables. Not compatible with OccBin, and currently not compatible with the pruned skewed Kalman filter (`kalman_algo=5`).

**conf_sig = DOUBLE**

Level of significance of the confidence interval used for classical forecasting after estimation. Default: 0.9.

**bayesian_irf**

Triggers the computation of the posterior distribution of IRFs. The length of the IRFs are controlled by the `irf` option. Results are stored in `oo_.PosteriorIRF.dsge` (see below for a description of this variable). Not compatible with OccBin.

**relative_irf**

See *relative_irf*.

**irf = INTEGER**

See *irf*. Only used if *bayesian_irf* is passed.

**irf_shocks = ( VARIABLE_NAME [[,] VARIABLE_NAME ...] )**

See *irf_shocks*. Only used if *bayesian_irf* is passed.

**irf_plot_threshold = DOUBLE**

See *irf_plot_threshold*. Only used if *bayesian_irf* is passed.

#### 4.16.3.9 Variable selection for posterior objects

**consider_all_endogenous**

Compute the posterior moments, smoothed variables, k-step ahead filtered variables and forecasts (when requested) on all the endogenous variables. This is equivalent to manually listing all the endogenous variables after the `estimation` command.

**consider_all_endogenous_and_auxiliary**

Compute the posterior moments, smoothed variables, k-step ahead filtered variables and forecasts (when requested) on all the endogenous variables and the auxiliary variables introduced by the preprocessor. This option is useful when e.g. running `smoother2histval` on the results of the Kalman smoother.

**consider_only_observed**

Compute the posterior moments, smoothed variables, k-step ahead filtered variables and forecasts (when requested) on all the observed variables. This is equivalent to manually listing all the observed variables after the `estimation` command.

**selected_variables_only**

Only run the classical smoother on the variables listed just after the `estimation` command. This option is incompatible with requesting classical frequentist forecasts and will be overridden in this case. When using Bayesian estimation, the smoother is by default only run on the declared endogenous variables. Default: run the smoother on all the declared endogenous variables.

#### 4.16.3.10 Model solution options

**solve_algo = INTEGER**

See *solve_algo*.

**order = INTEGER**

Order of approximation around the deterministic steady state. When greater than 1, the likelihood is evaluated with a particle or nonlinear filter (see Fernández-Villaverde and Rubio-Ramírez, 2005). Default is 1, i.e. the likelihood of the linearized model is evaluated using a standard Kalman filter.

**dr = OPTION**

See *dr*. Default: `default`, i.e. generalized Schur decomposition.

**dr_cycle_reduction_tol = DOUBLE**

See *dr_cycle_reduction_tol*. Default: `1e-7`.

**dr_cycle_reduction_maxiter = INTEGER**

See *dr_cycle_reduction_maxiter*. Default: `100`.

**dr_logarithmic_reduction_tol = DOUBLE**

See *dr_logarithmic_reduction_tol*. Default: `1e-12`.

**dr_logarithmic_reduction_maxiter = INTEGER**

See *dr_logarithmic_reduction_maxiter*. Default: `100`.

**qz_zero_threshold = DOUBLE**

See *qz_zero_threshold*.

#### 4.16.3.11 Posterior sampling options

**posterior_sampling_method = NAME**

Selects the sampler used to sample from the posterior distribution during Bayesian estimation. Default: `'random_walk_metropolis_hastings'`.

`'random_walk_metropolis_hastings'`

Instructs Dynare to use the Random-Walk Metropolis-Hastings. In this algorithm, the proposal density is recentered to the previous draw in every step.

`'dime_mcmc'`

Instructs Dynare to use the Differential-Independence Mixture Ensemble ("DIME") MCMC sampler of Boehl (2022) instead of the standard Random-Walk Metropolis-Hastings. DIME is robust for odd shaped, multimodal, black-box distributions and shown to require significantly less likelihood evaluations than alternative samplers. Many chains run simultaneously, thereby further increasing sampling speed. The algorithm is based on a gradient-free global multi-start optimizer and does not require any posterior mode density maximization prior to MCMC sampling. DIME proposals are generated from an endogenous and adaptive proposal distribution, thereby providing close-to-optimal proposal distributions for black box target distributions without manual fine-tuning. Does not yet support `moments_varendo`, `bayesian_irf`, and `smoother`.

Note that, since DIME is using parameter transformations, setting parameter bounds is often counterproductive. The `prior_trunc` option is disabled and set to zero.

`'tailored_random_block_metropolis_hastings'`

Instructs Dynare to use the Tailored randomized block (TaRB) Metropolis-Hastings algorithm proposed by Chib and Ramamurthy (2010) instead of the standard Random-Walk Metropolis-Hastings. In this algorithm, at each iteration the estimated parameters are randomly assigned to different blocks. For each of these blocks a mode-finding step is conducted. The inverse Hessian at this mode is then used as the covariance of the proposal density for a Random-Walk Metropolis-Hastings step. If the numerical Hessian is not positive definite, the generalized Cholesky decomposition of Schnabel and Eskow (1990) is used, but without pivoting. The TaRB-MH algorithm massively reduces the autocorrelation in the MH draws and thus reduces the number of draws required to representatively sample from the posterior. However, this comes at a computational cost as the algorithm takes more time to run.

`'independent_metropolis_hastings'`

Use the Independent Metropolis-Hastings algorithm where the proposal distribution - in contrast to the Random Walk Metropolis-Hastings algorithm - does not depend on the state of the chain.

`'slice'`

Instructs Dynare to use the Slice sampler of Planas *et al.* (2015). Note that `'slice'` is incompatible with `prior_trunc=0`.

Whereas one Metropolis-Hastings iteration requires one evaluation of the posterior, one slice iteration requires $neval$ evaluations, where, as a rule of thumb, $neval = 7 \times npar$ with $npar$ denoting the number of estimated parameters. Spending the same computational budget of $N$ posterior evaluations in the slice sampler then implies setting `mh_replic=N/neval`.

Note that the slice sampler will typically return less autocorrelated Monte Carlo Markov Chain draws than the MH-algorithm. Its relative (in)efficiency can be investigated via the reported inefficiency factors.

`'hssmc'`

Instructs Dynare to use the Herbst and Schorfheide (2014) version of the Sequential Monte Carlo sampler instead of the standard Random-Walk Metropolis-Hastings. Does not yet support `moments_varendo`, `bayesian_irf`, and `smoother`.

`'dsmh'`

Instructs Dynare to use the Dynamic Striated Metropolis-Hastings sampler proposed by Waggoner *et al.* (2016) instead of the standard Random-Walk Metropolis-Hastings.

`'online'`

Instructs Dynare to treat the (estimated) parameters as state variables and estimate them jointly with the original state variables of the model using a nonlinear filter. The algorithm implemented in Dynare is described in Liu and West (2001), and works with `k` order local approximations of the model.

**`posterior_sampler_options = (NAME, VALUE, ...)`**

A list of NAME and VALUE pairs. Can be used to set options for the posterior sampling methods. The set of available options depends on the selected posterior sampling routine (i.e. on the value of option *`posterior_sampling_method`*):

`'random_walk_metropolis_hastings'`

Available options are:

`'proposal_distribution'`

Specifies the statistical distribution used for the proposal density.

`'rand_multivariate_normal'`

Use a multivariate normal distribution. This is the default.

`'rand_multivariate_student'`

Use a multivariate student distribution.

`'student_degrees_of_freedom'`

Specifies the degrees of freedom to be used with the multivariate student distribution. Default: `3`.

`'use_mh_covariance_matrix'`

Indicates to use the covariance matrix of the draws from a previous MCMC run to define the covariance of the proposal distribution. Requires the *`load_mh_file`* option to be specified. Default: `0`.

`'scale_file'`

Provides the name of a `_mh_scale.mat` file storing the tuned scale factor from a previous run of `mode_compute=6`.

`'save_tmp_file'`

Save the MCMC draws into a `_mh_tmp_blck` file at the refresh rate of the status bar instead of just saving the draws when the current `_mh*_blck` file is full. Default: `0`

`'dime_mcmc'`

Available options are:

`'niter'`

Number of iterations. Default value is 1500.

`'nchain'`

Number of chains/particles. A range between $4ndim$ and $6ndim$ is recommended. For very difficult problems, double the number of chains. Default value is: $5ndim$.

`'tune'`

Number of ensemble iterations to keep after burn-in. Default value is chosen to obtain at least 50,000 samples.

`'aimh_prob'`

Probability to draw a global transition kernel. By default, this is set to $0.1$. It is usually not necessary to change this value.

`'df_proposal_dist'`

Degrees of freedom of the multivariate t distribution used for global kernel proposals. Defaults to $10$.

`'rho'`

Decay parameter for the mean and covariances of the global transition kernel. Defaults to $0.999$.

`'gamma'`

Mean stretch factor for the proposal vector. By default, it is $2.38/\sqrt{2\,\mathrm{ndim}}$ as recommended by Ter Braak (2006)

`'sigma'`

Standard deviation of the Gaussian used to stretch the proposal vector. This is normally negligible. Defaults to $1e-5$.

`'independent_metropolis_hastings'`

Takes the same options as in the case of `random_walk_metropolis_hastings`.

`'slice'`

Available options are:

`'rotated'`

Triggers rotated slice iterations using a covariance matrix from initial burn-in iterations. Requires either `use_mh_covariance_matrix` or `slice_initialize_with_mode`. Default: `0`.

`'mode_files'`

For multimodal posteriors, provide the name of a file containing a `nparam` by `nmodes` variable called `xparams` storing the different modes. This array must have one column vector per mode and the estimated parameters along the row dimension. With this info, the code will automatically trigger the `rotated` and `mode` options. Default: `[]`.

`'slice_initialize_with_mode'`

The default for slice is to set `mode_compute=0` and start the chain(s) from a random location in the prior space. This option first runs the mode-finder and then starts the chain from the mode. Together with `rotated`, it will use the inverse Hessian from the mode to perform rotated slice iterations. Default: `0`.

'initial_step_size'

Sets the initial size of the interval in the stepping-out procedure as fraction of the prior support, i.e. the size will be initial_step_size * (UB-LB). initial_step_size must be a real number in the interval [0,1]. Default: 0.8.

'use_mh_covariance_matrix'

See *use_mh_covariance_matrix*. Must be used with 'rotated'. Default: 0.

'save_iter_info_file'

Save iteration information to the metropolis subfolder. Default: 1.

'fast_likelihood_evaluation_for_rejection'

For OccBin, computes, for each t a 'predictive' upper bound of the likelihood density from t+1,...,T assuming perfect fit and rejects parameter draws worse ty more than 'fast_likelihood_evaluation_for_rejection_penalty' log points. This can save a lot of computation time by quickly rejecting poor parameter draws without evaluating the future regime sequence for additional t. Default: 0.

'use_prior_draws'

Use up to 10 Metropolis draws using the prior as proposal, within slice (Gibbs) sampling: in the first few slice iterations, prior sampling may already provide acceptable draws. Up to 10 new prior draws are tried for each slice iteration. If no prior draw is accepted after 10 attempts, prior sampling is stopped for all subsequent slice iterations. Particularly useful if the initial draw is infeasible. Default: 0.

'save_tmp_file'

See *save_tmp_file*. Default: 1.

'tailored_random_block_metropolis_hastings'

Available options are:

'proposal_distribution'

Specifies the statistical distribution used for the proposal density. See *proposal_distribution*.

new_block_probability = DOUBLE

Specifies the probability of the next parameter belonging to a new block when the random blocking in the TaRB Metropolis-Hastings algorithm is conducted. The higher this number, the smaller is the average block size and the more random blocks are formed during each parameter sweep. Default: 0.25.

mode_compute = INTEGER

Specifies the mode-finder run in every iteration for every block of the TaRB Metropolis-Hastings algorithm. See *mode_compute*. Default: 4.

optim = (NAME, VALUE,...)

Specifies the options for the mode-finder used in the TaRB Metropolis-Hastings algorithm. See *optim*.

'scale_file'

See *scale_file*..

`'save_tmp_file'`

See *save_tmp_file*. Default: 1.

`'hssmc'`

Available options are:

`'particles'`

Number of particles. Default value is: 20000.

`'steps'`

Number of weights $\phi_i \in [0,1]$ on the likelihood function used to define a sequence of tempered likelihoods. This parameter is denoted $N_\phi$ in Herbst and Schorfheide (2014), and we have $\phi_1 = 0$ and $\phi_{N_\phi} = 1$. Default value is: 25.

`'lambda'`

Positive parameter controlling the sequence of weights $\phi_i$, Default value is: 2. Weights are defined by:

$$\phi_i = \left(\frac{i-1}{N_\phi - 1}\right)^\lambda$$

for $i = 1, \ldots, N_\phi$. Usually we set $\lambda > 1$, so that $\Delta\phi_i = \phi_i - \phi_{i-1}$ is increasing with $i$.

`'target'`

Acceptance rate target. Default value is: .25.

`'scale'`

Scale parameter in the mutation step (on the proposal covariance matrix of the MH iteration). Default value is: .5.

`'dsmh'`

Available options are:

`'H'`

Number of tempering stages, i.e. the number of values in the sequence of tempering parameters $\lambda_i$ for $i = 1, \ldots, H$. The tempering schedule is defined by $\lambda_i = \exp\left(\frac{H-i}{H-1}\ln(\lambda_1)\right)$, so that $\lambda_1 = \frac{1}{10\,n\,T}$ (where $n$ is the number of estimated parameters and $T$ the number of observations) and $\lambda_H = 1$, following the recommendation of Waggoner *et al.* (2016). Default: 25.

`'N'`

Number of batches of draws collected in each mutation step. Together with `G` and `tau`, this controls the mutation step: `N*tau` Metropolis-Hastings iterations are run per group, and every `tau`-th draw is kept, yielding `N*G` draws (i.e. particles) per stage. Default: 20.

`'G'`

Number of groups (parallel chains) used in the tuning and mutation steps. Default: 10.

`'M'`

Number of striations used to build the dynamic grid of levels in the mutation step. Must divide `N*G` evenly. Default: 20.

`'K'`

Number of Metropolis-Hastings iterations per group member in the tuning step. It is used to adaptively calibrate the proposal scale parameter $c$. Default: 50.

`'alpha0'`

Lower bound of the target acceptance rate interval $[\alpha_0, \alpha_1]$ used when tuning the proposal scale parameter $c$. Default: 0.2.

`'alpha1'`

Upper bound of the target acceptance rate interval $[\alpha_0, \alpha_1]$ used when tuning the proposal scale parameter $c$. Default: 0.3.

`'tau'`

Thinning parameter in the mutation step. In each stage, `N*tau` Metropolis-Hastings iterations are run per group, and every `tau`-th iteration is retained to reduce autocorrelation in the draws. Default: 10.

**sub_draws = INTEGER**

Number of draws from the MCMC that are used to compute posterior distribution of various objects (smoothed variable, smoothed shocks, forecast, moments, IRF). The draws used to compute these posterior moments are sampled uniformly in the estimated empirical posterior distribution (i.e. draws of the MCMC). `sub_draws` should be smaller than the total number of MCMC draws available. Default: `min(posterior_max_subsample_draws, (Total number of draws)*(number of chains) )`.

**posterior_max_subsample_draws = INTEGER**

Maximum number of draws from the MCMC used to compute posterior distribution of various objects (smoothed variable, smoothed shocks, forecast, moments, IRF), if not overridden by option `sub_draws`. Default: `1200`.

**prior_trunc = DOUBLE**

Probability of extreme values of the prior density in each tail that is ignored when computing bounds for the parameters. Default: `1e-10` for `posterior_sampling_method=slice` and `0` otherwise.

**huge_number = DOUBLE**

Value for replacing infinite values in the definition of (prior) bounds when finite values are required for computational reasons. Default: `1e7`.

**endogenous_prior**

Use endogenous priors as in Christiano *et al.* (2011). The procedure is motivated by sequential Bayesian learning. Starting from independent initial priors on the parameters, specified in the `estimated_params` block, the standard deviations observed in a "pre-sample", taken to be the actual sample, are used to update the initial priors. Thus, the product of the initial priors and the pre-sample likelihood of the standard deviations of the observables is used as the new prior (for more information, see the technical appendix of Christiano *et al.* (2011)). This procedure helps in cases where the regular posterior estimates, which minimize in-sample forecast errors, result in a large overprediction of model variable variances (a statistic that is not explicitly targeted, but often of particular interest to researchers).

**estimate_initial_states_endogenous_prior**

**estimate_initial_states_endogenous_prior = BOOLEAN**

Jointly estimate the initial states along with the other parameters. This contrasts to the usual approach of marginalizing with respect to the states while estimating parameters, and then estimating the states offline using the smoother. This option uses the unconditional mean and variance of the states implied by the linear Kalman filter as an endogenous prior. This means that, for a linear Kalman filter, the joint posterior of parameters and states is exactly the same as what would be obtained from the usual marginalized likelihood approach combined with offline state estimation via the smoother. The initial-state values are treated as additional parameters to be estimated. The

option automatically sets `lik_init=2` and requires `posterior_sampling_method='slice'`. This option is useful for nonlinear models estimated e.g. via OccBin.

#### 4.16.3.12 MCMC options

**`mh_conf_sig = DOUBLE`**

Confidence/HPD interval used for the computation of prior and posterior statistics like: parameter distributions, prior/posterior moments, conditional variance decomposition, impulse response functions, Bayesian forecasting. Default: `0.9`.

**`mh_replic = INTEGER`**

Number of replications for each chain of the Metropolis-Hastings algorithm. The number of draws should be sufficient to achieve convergence of the MCMC and to meaningfully compute posterior objects. Default: `20000`.

**`mh_nblocks = INTEGER`**

Number of parallel chains for Metropolis-Hastings algorithm. Default: `2`.

**`mh_drop = DOUBLE`**

The fraction of initially generated parameter vectors to be dropped as a burn-in before using posterior simulations. Default: `0.5`.

**`mh_jscale = DOUBLE`**

The scale parameter of the jumping distribution's covariance matrix (Metropolis-Hastings or TaRB-algorithm). This option must be tuned to obtain, ideally, an acceptance ratio of 25%-33%. Basically, the idea is to increase the variance of the jumping distribution if the acceptance ratio is too high, and decrease the same variance if the acceptance ratio is too low. In some situations it may help to consider parameter-specific values for this scale parameter. This can be done in the *`estimated_params`* block.

Note that `mode_compute=6` will tune the scale parameter to achieve an acceptance rate of *AcceptanceRateTarget*. The resulting scale parameter will be saved into a file named `MODEL_FILENAME_mh_scale.mat` in the `FILENAME/Output` folder. This file can be loaded in subsequent runs via the `posterior_sampler_options` option *scale_file*. Both `mode_compute=6` and `scale_file` will overwrite any value specified in `estimated_params` with the tuned value. Default: `2.38/sqrt(n)`.

Note also that for the Random Walk Metropolis-Hastings algorithm, it is possible to use option *`mh_tune_jscale`*, to automatically tune the value of `mh_jscale`. In this case, the `mh_jscale` option must not be used.

**`mh_init_scale = DOUBLE (deprecated)`**

The scale to be used for drawing the initial value of the Metropolis-Hastings chain. Generally, the starting points should be overdispersed for the Brooks and Gelman (1998) convergence diagnostics to be meaningful. Default: `2*mh_jscale.`

It is important to keep in mind that `mh_init_scale` is set at the beginning of Dynare execution, i.e. the default will not take into account potential changes in `mh_jscale` introduced by either `mode_compute=6` or the `posterior_sampler_options` option *scale_file*. If `mh_init_scale` is too wide during initalization of the posterior sampler so that 100 tested draws are inadmissible (e.g. Blanchard-Kahn conditions are always violated), Dynare will request user input of a new `mh_init_scale` value with which the next 100 draws will be drawn and tested. If the *`nointeractive`* option has been invoked, the program will instead automatically decrease `mh_init_scale` by 10 percent after 100 futile draws and try another 100 draws. This iterative procedure will take place at most 10 times, at which point Dynare will abort with an error message.

**`mh_init_scale_factor = DOUBLE`**

The multiple of `mh_jscale` used for drawing the initial value of the Metropolis-Hastings chain. Generally, the starting points should be overdispersed for the Brooks and Gelman (1998) convergence diagnostics to be meaningful. Default: `2`

If `mh_init_scale_factor` is too wide during initialization of the posterior sampler so that 100 tested draws are inadmissible (e.g. Blanchard-Kahn conditions are always violated), Dynare will request user input of a new `mh_init_scale_factor` value with which the next 100 draws will be drawn and tested. If the *nointeractive* option has been invoked, the program will instead automatically decrease `mh_init_scale_factor` by 10 percent after 100 futile draws and try another 100 draws. This iterative procedure will take place at most 10 times, at which point Dynare will abort with an error message.

**mh_tune_jscale [= DOUBLE]**

Automatically tunes the scale parameter of the jumping distribution's covariance matrix (Metropolis-Hastings), so that the overall acceptance ratio is close to the desired level. Default value is `0.33`. It is not possible to match exactly the desired acceptance ratio because of the stochastic nature of the algorithm (the proposals and the initial conditions of the Markov chains if `mh_nblocks>1`). This option is only available for the Random Walk Metropolis-Hastings algorithm. Must not be used in conjunction with *mh_jscale = DOUBLE*.

**mh_tune_guess = DOUBLE**

Specifies the initial value for the *mh_tune_jscale* option. Default: `2.38/sqrt(n)`. Must not be set if *mh_tune_jscale* is not used.

**mh_recover**

Attempts to recover a Metropolis-Hastings simulation that crashed prematurely, starting with the last available saved `mh`-file. Shouldn't be used together with *load_mh_file* or a different `mh_replic` than in the crashed run. Since Dynare 4.5 the proposal density from the previous run will automatically be loaded. In older versions, to assure a neat continuation of the chain with the same proposal density, you should provide the *mode_file = FILENAME* used in the previous run or the same user-defined `mcmc_jumping_covariance` when using this option. Note that under Octave, a neat continuation of the crashed chain with the respective last random number generator state is currently not supported.

**mh_posterior_mode_estimation**

Skip optimizer-based mode-finding and instead compute the mode based on a run of a MCMC. The MCMC will start at the prior mode and use the prior variances to compute the inverse Hessian.

**load_mh_file**

Tells Dynare to add to previous Metropolis-Hastings simulations instead of starting from scratch. Since Dynare 4.5 the proposal density from the previous run will automatically be loaded. In older versions, to assure a neat continuation of the chain with the same proposal density, you should provide the *mode_file = FILENAME* used in the previous run or the same user-defined `mcmc_jumping_covariance` when using this option. Shouldn't be used together with *mh_recover*. Note that under Octave, a neat continuation of the chain with the last random number generator state of the already present draws is currently not supported.

**load_results_after_load_mh**

This option is available when loading a previous MCMC run without adding additional draws, i.e. when *load_mh_file* is specified with `mh_replic=0`. It tells Dynare to load the previously computed convergence diagnostics, marginal data density, and posterior statistics from an existing `_results` file instead of recomputing them.

**mh_initialize_from_previous_mcmc**

This option allows picking initial values for new MCMC from a previous one, where the model specification, the number of estimated parameters, (some) prior might have changed (so a situation where *load_mh_file* would not work). If an additional parameter is estimated, it is automatically initialized from prior_draw. Note that, if this option is used to skip the optimization step, you should use a sampling method which does not require a proposal density, like slice. Otherwise, optimization should always be done beforehand or a mode file with an appropriate posterior covariance matrix should be used.

**mh_initialize_from_previous_mcmc_directory = FILENAME**

If mh_initialize_from_previous_mcmc is set, users must provide here the path to the standard FNAME folder from where to load prior definitions and last MCMC values to be used to initialize the new MCMC.

Example: if previous project directory is /my_previous_dir and FNAME is mymodel, users should set the option as

```
mh_initialize_from_previous_mcmc_directory = '/my_previous_dir/mymodel'
```

Dynare will then look for the last record file into

/my_previous_dir/mymodel/metropolis/mymodel_mh_history_<LAST>.mat

and for the prior definition file into

/my_previous_dir/mymodel/prior/definition.mat

**mh_initialize_from_previous_mcmc_record = FILENAME**

If mh_initialize_from_previous_mcmc is set, and whenever the standard file or directory tree is not applicable to load initial values, users may directly provide here the path to the record file from which to load values to be used to initialize the new MCMC.

**mh_initialize_from_previous_mcmc_prior = FILENAME**

If mh_initialize_from_previous_mcmc is set, and whenever the standard file or directory tree is not applicable to load initial values, users may directly provide here the path to the prior definition file, to get info in the priors used in previous MCMC.

**mcmc_jumping_covariance = OPTION**

Tells Dynare which covariance to use for the proposal density of the MCMC sampler. OPTION can be one of the following:

hessian

Uses the Hessian matrix computed at the mode.

prior_variance

Uses the prior variances. No infinite prior variances are allowed in this case.

identity_matrix

Uses an identity matrix.

FILENAME

Loads an arbitrary user-specified covariance matrix from FILENAME.mat. The covariance matrix must be saved in a variable named jumping_covariance, must be square, positive definite, and have the same dimension as the number of estimated parameters.

Note that the covariance matrices are still scaled with *mh_jscale*. Default value is hessian.

#### 4.16.3.13 Convergence diagnostics

**nodiagnostic**

Does not compute the convergence diagnostics for Metropolis-Hastings. Default: diagnostics are computed and displayed.

**taper_steps = [INTEGER1 INTEGER2 ...]**

Percent tapering used for the spectral window in the Geweke (1992), Geweke (1999) convergence diagnostics (requires *mh_nblocks=1*). The tapering is used to take the serial correlation of the posterior draws into account. Default: [4 8 15].

**brooks_gelman_plotrows = INTEGER**

Number of parameters to depict along the rows of the figures depicting the Brooks and Gelman (1998) convergence diagnostics. Default: 3.

**geweke_interval = [DOUBLE DOUBLE]**

Percentage of MCMC draws at the beginning and end of the MCMC chain taken to compute the Geweke (1992), Geweke (1999) convergence diagnostics (requires *mh_nblocks=1*) after discarding the first *mh_drop* percent of draws as a burnin. Default: [0.2 0.5].

**raftery_lewis_diagnostics**

Triggers the computation of the Raftery and Lewis (1992) convergence diagnostics. The goal is delivering the number of draws required to estimate a particular quantile of the CDF `q` with precision `r` with a probability `s`. Typically, one wants to estimate the `q=0.025` percentile (corresponding to a 95 percent HPDI) with a precision of 0.5 percent (`r=0.005`) with 95 percent certainty (`s=0.95`). The defaults can be changed via *raftery_lewis_qrs*. Based on the theory of first-order Markov Chains, the diagnostics will provide a required burn-in (`M`), the number of draws after the burn-in (`N`) as well as a thinning factor that would deliver a first-order chain (`k`). The last line of the table will also deliver the maximum over all parameters for the respective values.

**raftery_lewis_qrs = [DOUBLE DOUBLE DOUBLE]**

Sets the quantile of the CDF `q` that is estimated with precision `r` with a probability `s` in the Raftery and Lewis (1992) convergence diagnostics. Default: `[0.025 0.005 0.95]`.

#### 4.16.3.14 DSGE-VAR

**dsge_var**

**dsge_var = DOUBLE**

Triggers the estimation of a DSGE-VAR model. If no `DOUBLE` is specified, the weight of the DSGE prior of the VAR model will be estimated (as in Adjemian *et al.* (2008)). In this case, the prior on the weight of the DSGE VAR prior, `dsge_prior_weight`, must be defined in the *estimated_params* block.

If a `DOUBLE` is passed, the weight of the DSGE prior of the VAR model is calibrated to the value passed (see Negro and Schorfheide (2004)). The value represents the ratio of dummy over actual observations. To assure that the prior is proper, the value must be bigger than $(k+n)/T$, where $k$ is the number of estimated parameters, $n$ is the number of observables, and $T$ is the number of observations.

**dsge_varlag = INTEGER**

The number of lags used to estimate a DSGE-VAR model. Default: `4`.

*Output*

**MATLAB/Octave variable: oo_.dsge_var.posterior_mode**

Stores the values at the mode of some of the objects arising during estimation.

#### 4.16.3.15 Non-linear filter options

**pruning**

See *pruning*. Governs the creation of posterior objects during estimation. When using nonlinear filters, it should be consistent with the setting of the `pruning` option of `particle_filter_options`. Default: not enabled.

**number_of_particles = INTEGER**

Number of particles used when evaluating the likelihood of a non-linear state-space model. Default: `5000`.

**filter_algorithm = OPTION**

Sets the particle filter algorithm. Possible values for OPTION are:

`sis`

Sequential importance sampling algorithm of the Gordon *et al.* (1993) type. This is the default value.

`apf`

Auxiliary particle filter of the Pitt and Shephard (1999) type.

`gf`

Gaussian filter of the Kotecha and Djuric (2003) type.

`gmf`

Gaussian mixture filter of the Kotecha and Djuric (2003) type.

`cpf`

Conditional particle filter of the Ionides (2003) type as used in, e.g., Amisano and Tristani (2010).

`nlkf`

Use a standard (linear) Kalman filter algorithm with the nonlinear measurement and state equations.

**`resampling = OPTION`**

Determines if resampling of the particles is done. Possible values for OPTION are:

`none`

No resampling.

`systematic`

Resampling at each iteration, this is the default value.

`generic`

Resampling if and only if the effective sample size is below a certain level defined by *`resampling_threshold`* * *`number_of_particles`*.

**`resampling_threshold = DOUBLE`**

A real number between zero and one. The resampling step is triggered as soon as the effective number of particles is less than this number times the total number of particles (as set by *`number_of_particles`*). This option is effective if and only if option *`resampling`* has value `generic`.

**`resampling_method = OPTION`**

Sets the resampling method. Possible values for OPTION are: `kitagawa`, `stratified` and `smooth`. The `smooth` option is incompatible with `filter_algorithm=apf`

**`proposal_approximation = OPTION`**

Sets the method for approximating the proposal distribution in case of the `filter_algorithm=cpf`, `gf`, `gpf` and `nlkf` filters. Possible values for OPTION are:

`unscented`

Unscented transform as in Julier and Uhlmann (1997) and Wan and van der Merwe (2001). This is the default.

`cubature`

Cubature method as in Arasaratnam and Haykin (2009).

`montecarlo`

Monte Carlo integration method. Not available with `filter_algorithm=gmf`.

**distribution_approximation = OPTION**

Sets the method for approximating the particle distribution for the `filter_algorithm=gf`, `gpf` and `nlkf` filters. Possible values for OPTION are: `cubature`, `montecarlo` and `unscented`. Default value is `unscented`. For details, see *proposal_approximation = OPTION*. Option value `montecarlo` is not supported for `filter_algorithm=gmf`.

**cpf_weights = OPTION**

Controls the method used to update the weights in conditional particle filter, possible values are

`amisanotristani`

Use the approach in Amisano and Tristani (2010). This is the default.

`murrayjonesparslow`

Use the approach in Murray *et al.* (2013).

**nonlinear_filter_initialization = INTEGER**

Sets the initial condition of the nonlinear filters. By default, the nonlinear filters are initialized with the unconditional covariance matrix of the state variables, computed with the reduced-form solution of the first-order approximation of the model. If `nonlinear_filter_initialization=2`, the nonlinear filter is instead initialized with a covariance matrix estimated with a stochastic simulation of the reduced-form solution of the second-order approximation of the model. Both these initializations assume that the model is stationary, and cannot be used if the model has unit roots (which can be seen with the *check* command prior to estimation). If the model has stochastic trends, user must use `nonlinear_filter_initialization=3`, the filters are then initialized with an identity matrix for the covariance matrix of the state variables. Default value is `nonlinear_filter_initialization=1` (initialization based on the first-order approximation of the model).

**particle_filter_options = (NAME, VALUE, ...)**

A list of NAME and VALUE pairs. Can be used to set some fine-grained options for the particle filter routines. The set of available options depends on the selected filter routine.

More information on particle filter options is available at https://git.dynare.org/Dynare/dynare/-/wikis/Particle-filters.

Available options are:

`'pruning'`

Enable pruning for particle filter-related simulations. It is currently only supported by the `sis`, `apf`. Default: `false`.

`'liu_west_delta'`

Set the value for delta for the Liu/West online filter (`posterior_sampling_method='online'`). Default: `0.99`.

`'unscented_alpha'`

Set the value for alpha for `unscented` option of `distribution_approximation` and `proposal_approximation`. The parameterization follows Wan and van der Merwe (2001). Value must be between 0 and 1. Default: 1.

`'unscented_beta'`

Set the value for beta for `unscented` option of `distribution_approximation` and `proposal_approximation`. Governs the covariance approximation. The parameterization follows Wan and van der Merwe (2001). Parameter needs to be weakly bigger than 0 and should be 2 for a Gaussian distribution. Default: `2`.

`'unscented_kappa'`

Set the value for kappa for `unscented` option of `distribution_approximation` and `proposal_approximation`. The parameterization follows Wan and van der Merwe (2001). Value must be weakly bigger than 1. Default: `1`.

`'initial_state_prior_std'`

Value of the diagonal elements for the initial covariance of the state variables when employing `nonlinear_filter_initialization=3`. Default: `1`.

`'mixture_state_variables'`

Number of mixture components in the Gaussian-mixture filter (`gmf`) for the state variables. Default: `5`.

`'mixture_structural_shocks'`

Number of mixture components in the Gaussian-mixture filter (`gmf`) for the structural shocks. Default: `1`.

`'mixture_measurement_shocks'`

Number of mixture components in the Gaussian-mixture filter (`gmf`) for the measurement errors. Default: `1`.

### 4.16.4 Estimation output variables

After running estimation, the following are updated: the structural parameters `M_.params`, the shock variance matrix `M_.Sigma_e`, the shock skewness coefficients `M_.Skew_e`, and the measurement error variance matrix `M_.H`. For maximum likelihood estimation or posterior mode computation without Metropolis iterations, these are set to the mode. After estimation with Metropolis iterations (option `mh_replic > 0` or option *load_mh_file*), they are set to the posterior mean.

Depending on the options, *estimation* stores results in various fields of the `oo_` structure, described below. In the following variables, we will adopt the following shortcuts for specific field names:

`MOMENT_NAME`

This field can take the following values:

`HPDinf`

Lower bound of a 90% HPD interval.[4]

`HPDsup`

Upper bound of a 90% HPD interval.

`HPDinf_ME`

Lower bound of a 90% HPD interval[5] for observables when taking measurement error into account (see e.g. Christoffel *et al.* (2011)), p.17).

`HPDsup_ME`

Upper bound of a 90% HPD interval for observables when taking measurement error into account.

`Mean`

Mean of the posterior distribution.

`Median`

Median of the posterior distribution.

`Std`

[4] See options *conf_sig* and *mh_conf_sig* to change the size of the HPD interval.
[5] See options *conf_sig* () and *mh_conf_sig* to change the size of the HPD interval.

Standard deviation of the posterior distribution.

`Variance`

Variance of the posterior distribution.

`deciles`

Deciles of the distribution.

`density`

Non parametric estimate of the posterior density following the approach outlined in Sköld and Roberts (2003). First and second columns are respectively abscissa and ordinate coordinates.

`ESTIMATED_OBJECT`

This field can take the following values:

`measurement_errors_corr`

Correlation between two measurement errors.

`measurement_errors_std`

Standard deviation of measurement errors.

`parameters`

Parameters.

`shocks_corr`

Correlation between two structural shocks.

`shocks_std`

Standard deviation of structural shocks.

`shocks_skew`

Skewness coefficient of structural shocks.

**MATLAB/Octave variable: `oo_.MarginalDensity.LaplaceApproximation`**

Variable set by the *estimation* command. Stores the marginal data density based on the Laplace Approximation.

**MATLAB/Octave variable: `oo_.MarginalDensity.ModifiedHarmonicMean`**

Variable set by the ::opt`estimation` command, if it is used with `mh_replic` > `0` or *load_mh_file* option. Stores the marginal data density based on Geweke (1999) Modified Harmonic Mean estimator.

**MATLAB/Octave variable: `oo_.posterior.optimization`**

Variable set by the *estimation* command if mode-finding is used. Stores the results at the mode. Fields are of the form:

```
oo_.posterior.optimization.OBJECT
```

where OBJECT is one of the following:

`mode`

Parameter vector at the mode.

`Variance`

Inverse Hessian matrix at the mode or MCMC jumping covariance matrix when used with the *MCMC_jumping_covariance* option.

`log_density`

Log likelihood (ML)/log posterior density (Bayesian) at the mode when used with `mode_compute>0`.

**MATLAB/Octave variable: oo_.posterior.metropolis**

Variable set by the *estimation* command if `mh_replic>0` is used. Fields are of the form:

```
oo_.posterior.metropolis.OBJECT
```

where OBJECT is one of the following:

`mean`

Mean parameter vector from the MCMC.

`Variance`

Covariance matrix of the parameter draws in the MCMC.

**MATLAB/Octave variable: oo_.FilteredVariables**

Variable set by the *estimation* command, if it is used with the *filtered_vars* option.

After an estimation without Metropolis, fields are of the form:

```
oo_.FilteredVariables.VARIABLE_NAME
```

After an estimation with Metropolis, fields are of the form:

```
oo_.FilteredVariables.MOMENT_NAME.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.FilteredVariablesKStepAhead**

Variable set by the *estimation* command, if it is used with the *filter_step_ahead* option. The k-steps are stored along the rows while the columns indicate the respective variables. The third dimension of the array provides the observation for which the forecast has been made. For example, if `filter_step_ahead=[1 2 4]` and `nobs=200`, the element (3,5,204) stores the four period ahead filtered value of variable 5 computed at time t=200 for time t=204. The periods at the beginning and end of the sample for which no forecasts can be made, e.g. entries (1,5,1) and (1,5,204) in the example, are set to zero. Note that in case of Bayesian estimation the variables will be ordered in the order of declaration after the estimation command (or in general declaration order if no variables are specified here). In case of running the classical smoother, the variables will always be ordered in general declaration order. If the *selected_variables_only* option is specified with the classical smoother, non-requested variables will be simply left out in this order.

**MATLAB/Octave variable: oo_.FilteredVariablesKStepAheadVariances**

Variable set by the *estimation* command, if it is used with the *filter_step_ahead* option. It is a 4 dimensional array where the k-steps are stored along the first dimension, while the fourth dimension of the array provides the observation for which the forecast has been made. The second and third dimension provide the respective variables. For example, if `filter_step_ahead=[1 2 4]` and `nobs=200`, the element (3,4,5,204) stores the four period ahead forecast error covariance between variable 4 and variable 5, computed at time t=200 for time t=204. Padding with zeros and variable ordering is analogous to `oo_.FilteredVariablesKStepAhead`.

**MATLAB/Octave variable: oo_.Filtered_Variables_X_step_ahead**

Variable set by the *estimation* command, if it is used with the *filter_step_ahead* option in the context of Bayesian estimation. Fields are of the form:

```
oo_.Filtered_Variables_X_step_ahead.VARIABLE_NAME
```

The n-th entry stores the k-step ahead filtered variable computed at time n for time n+k.

**MATLAB/Octave variable: oo_.FilteredVariablesShockDecomposition**

Variable set by the *estimation* command, if it is used with the *filter_step_ahead* option. The k-steps are stored along the rows while the columns indicate the respective variables. The third dimension corresponds to the shocks in declaration order. The fourth dimension of the array provides the observation for which the forecast has been made. For example, if `filter_step_ahead=[1 2 4]` and `nobs=200`, the element (3,5,2,204) stores the contribution of the second shock to the four period ahead filtered value of variable 5 (in deviations from the mean) computed at time t=200 for time t=204. The periods at the beginning and end of the sample for which no forecasts can be made, e.g. entries (1,5,1) and (1,5,204) in the example, are set to zero. Padding with zeros and variable ordering is analogous to `oo_.FilteredVariablesKStepAhead`.

**MATLAB/Octave variable: oo_.PosteriorIRF.dsge**

Variable set by the *estimation* command, if it is used with the *bayesian_irf* option. Fields are of the form:

```
oo_.PosteriorIRF.dsge.MOMENT_NAME.VARIABLE_NAME_SHOCK_NAME
```

**MATLAB/Octave variable: oo_.SmoothedMeasurementErrors**

Variable set by the *estimation* command, if it is used with the *smoother* option. Fields are of the form:

```
oo_.SmoothedMeasurementErrors.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.SmoothedShocks**

Variable set by the *estimation* command (if used with the *smoother* option), or by the *calib_smoother* command.

After an estimation without Metropolis, or if computed by *calib_smoother* , fields are of the form:

```
oo_.SmoothedShocks.VARIABLE_NAME
```

After an estimation with Metropolis, fields are of the form:

```
oo_.SmoothedShocks.MOMENT_NAME.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.SmoothedVariables**

Variable set by the *estimation* command (if used with the *smoother* option), or by the *calib_smoother* command.

After an estimation without Metropolis, or if computed by *calib_smoother* , fields are of the form:

```
oo_.SmoothedVariables.VARIABLE_NAME
```

After an estimation with Metropolis, fields are of the form:

```
oo_.SmoothedVariables.MOMENT_NAME.VARIABLE_NAME
```

**MATLAB/Octave command: get_smooth('VARIABLE_NAME' [, 'VARIABLE_NAME']...);**

Returns the smoothed values of the given endogenous or exogenous variable(s), as they are stored in the *oo_.SmoothedVariables* and *oo_.SmoothedShocks* variables.

**MATLAB/Octave command: get_smooth_dseries ;**

**MATLAB/Octave command: get_smooth_dseries('VARIABLE_NAME' [, 'VARIABLE_NAME']...);**

Similar to *get_smooth*, but returns the smoothed values as a *dseries* object, using the date information stored in *oo_.Smoother.initial_date*. If no variable name is given, returns a `dseries` containing all endogenous and exogenous variables.

**MATLAB/Octave variable: oo_.UpdatedVariables**

Variable set by the *estimation* command (if used with the *smoother* option), or by the *calib_smoother* command. Contains the estimation of the expected value of variables given the information available at the current date.

After an estimation without Metropolis, or if computed by *calib_smoother* , fields are of the form:

```
oo_.UpdatedVariables.VARIABLE_NAME
```

After an estimation with Metropolis, fields are of the form:

```
oo_.UpdatedVariables.MOMENT_NAME.VARIABLE_NAME
```

**MATLAB/Octave command: get_update('VARIABLE_NAME' [, 'VARIABLE_NAME']...);**

Returns the updated values of the given variable(s), as they are stored in the `oo_.UpdatedVariables` variable.

**MATLAB/Octave variable: oo_.FilterCovariance**

Three-dimensional array set by the *estimation* command if used with the *smoother* and Metropolis, if the *filter_covariance* option has been requested. Contains the series of one-step ahead forecast error covariance matrices from the Kalman smoother. The `M_.endo_nbr` times `M_.endo_nbr` times `T+1` array contains the variables in declaration order along the first two dimensions. The third dimension of the array provides the observation for which the forecast has been made. Fields are of the form:

```
oo_.FilterCovariance.MOMENT_NAME
```

Note that density estimation is not supported.

**MATLAB/Octave variable: oo_.Smoother.Variance**

Three-dimensional array set by the *estimation* command (if used with the *smoother* option) without Metropolis, or by the *calib_smoother* command, if the *filter_covariance* option has been requested. Contains the series of one-step ahead forecast error covariance matrices from the Kalman smoother. The `M_.endo_nbr` times `M_.endo_nbr` times `T+1` array contains the variables in declaration order along the first two dimensions. The third dimension of the array provides the observation for which the forecast has been made.

**MATLAB/Octave variable: oo_.Smoother.State_uncertainty**

Three-dimensional array set by the *estimation* command (if used with the *smoother* option) without Metropolis, or by the *calib_smoother* command, if the *smoothed_state_uncertainty* option has been requested. Contains the series of covariance matrices for the state estimate given the full data from the Kalman smoother. The `M_.endo_nbr` times `M_.endo_nbr` times `T` array contains the variables in declaration order along the first two dimensions. The third dimension of the array provides the observation for which the smoothed estimate has been made.

**MATLAB/Octave variable: oo_.Smoother.Shock_variance**

Three-dimensional array set by the *estimation* command (if used with the *smoother* option) without Metropolis, or by the *calib_smoother* command, if the *smoothed_shock_variance* option has been requested. Contains the variance of the smoothed structural shocks $Var(\eta_t|Y_{1:T})$. The `M_.exo_nbr` times `M_.exo_nbr` times `T` array contains the exogenous variables in declaration order along the first two dimensions.

**MATLAB/Octave variable: oo_.Smoother.initial_date**

A date object (see *Dates in a mod file*) corresponding to the date of the first observation on which the smoother was run (if no date information was given when passing the dataset, an arbitrary date is generated).

**MATLAB/Octave variable: oo_.SmoothedShockVariances**

Variable set by the *estimation* command (if used with the *smoother* option) without Metropolis, or by the *calib_smoother* command, if the *smoothed_shock_variance* option has been requested. Fields are of the form `oo_.SmoothedShockVariances.EXONAME` and contain the diagonal entries of the smoothed shock variance matrix for each exogenous variable (i.e., the posterior variance of each individual shock given all observations). Not set when Metropolis is used; see `oo_.SmoothedShockVariance` instead.

**MATLAB/Octave variable: oo_.SmoothedShockVariance**

Variable set by the *estimation* command if used with the *smoother* option and Metropolis, if the *smoothed_shock_variance* option has been requested. Contains posterior statistics (`Mean`, `Median`, `var`, `HPDinf`, `HPDsup`, `post_deciles`) of the full `M_.exo_nbr` times `M_.exo_nbr` times `T` smoothed shock variance matrix $Var(\eta_t|Y_{1:T})$.

**MATLAB/Octave variable: oo_.Smoother.SteadyState**

Variable set by the *estimation* command (if used with the *smoother* option) without Metropolis, or by the *calib_smoother* command. Contains the steady-state component of the endogenous variables used in the smoother in order of variable declaration.

**MATLAB/Octave variable: oo_.Smoother.TrendCoeffs**

Variable set by the *estimation* command (if used with the *smoother* option) without Metropolis, or by the *calib_smoother* command. Contains the trend coefficients of the observed variables used in the smoother in order of declaration of the observed variables.

**MATLAB/Octave variable: oo_.Smoother.Trend**

Variable set by the *estimation* command (if used with the *smoother* option), or by the *calib_smoother* command. Contains the trend component of the variables used in the smoother.

Fields are of the form:

```
oo_.Smoother.Trend.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.Smoother.Constant**

Variable set by the *estimation* command (if used with the *smoother* option), or by the *calib_smoother* command. Contains the constant part of the endogenous variables used in the smoother, accounting e.g. for the data mean when using the prefilter option.

Fields are of the form:

```
oo_.Smoother.Constant.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.Smoother.loglinear**

Indicator keeping track of whether the smoother was run with the *loglinear* option and thus whether stored smoothed objects are in logs.

**MATLAB/Octave variable: oo_.PosteriorTheoreticalMoments**

Variable set by the *estimation* command, if it is used with the *moments_varendo* option. Fields are of the form:

```
oo_.PosteriorTheoreticalMoments.dsge.THEORETICAL_MOMENT.ESTIMATED_
↪OBJECT.MOMENT_NAME.VARIABLE_NAME
```

where *THEORETICAL_MOMENT* is one of the following:

`covariance`

Variance-covariance of endogenous variables.

`contemporaneous_correlation`

Contemporaneous correlation of endogenous variables when the *contemporaneous_correlation* option is specified.

`correlation`

Auto- and cross-correlation of endogenous variables. Fields are vectors with correlations from 1 up to order `options_.ar`.

`VarianceDecomposition`

Decomposition of variance (unconditional variance, i.e. at horizon infinity).[6]

`VarianceDecompositionME`

Same as *VarianceDecomposition*, but contains the decomposition of the measured as opposed to the actual variable. The joint contribution of the measurement error will be saved in a field named `ME`.

`ConditionalVarianceDecomposition`

Only if the `conditional_variance_decomposition` option has been specified. In the presence of measurement error, the field will contain the variance contribution after measurement error has been taken out, i.e. the decomposition will be conducted of the actual as opposed to the measured variables.

`ConditionalVarianceDecompositionME`

Only if the `conditional_variance_decomposition` option has been specified. Same as *ConditionalVarianceDecomposition*, but contains the decomposition of the measured as opposed to the actual variable. The joint contribution of the measurement error will be saved in a field names `ME`.

**MATLAB/Octave variable: oo_.posterior_density**

Variable set by the *estimation* command, if it is used with *mh_replic* > 0 or *load_mh_file* option. Fields are of the form:

```
oo_.posterior_density.PARAMETER_NAME
```

**MATLAB/Octave variable: oo_.posterior_hpdinf**

Variable set by the *estimation* command, if it is used with *mh_replic* > 0 or *load_mh_file* option. Fields are of the form:

```
oo_.posterior_hpdinf.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.posterior_hpdsup**

Variable set by the *estimation* command, if it is used with *mh_replic* > 0 or *load_mh_file* option. Fields are of the form:

```
oo_.posterior_hpdsup.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.posterior_mean**

Variable set by the *estimation* command, if it is used with *mh_replic* > 0 or *load_mh_file* option. Fields are of the form:

```
oo_.posterior_mean.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.posterior_mode**

Variable set by the *estimation* command during mode-finding. If no mode-finding is conducted, the field will be set to the mode from the MCMC if *mh_replic* > 0. Fields are of the form:

```
oo_.posterior_mode.ESTIMATED_OBJECT.VARIABLE_NAME
```

[6] When the shocks are correlated, it is the decomposition of orthogonalized shocks via Cholesky decomposition according to the order of declaration of shocks (see *Variable declarations*)

**MATLAB/Octave variable: oo_.posterior_std_at_mode**

Variable set by the *estimation* command during mode-finding. It is based on the inverse Hessian at `oo_.posterior_mode`. Fields are of the form:

```
oo_.posterior_std_at_mode.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.posterior_std**

Variable set by the *estimation* command, if it is used with *mh_replic* > 0 or *load_mh_file* option. Fields are of the form:

```
oo_.posterior_std.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.posterior_var**

Variable set by the *estimation* command, if it is used with *mh_replic* > 0 or *load_mh_file* option. Fields are of the form:

```
oo_.posterior_var.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.posterior_median**

Variable set by the *estimation* command, if it is used with *mh_replic* > 0 or *load_mh_file* option. Fields are of the form:

```
oo_.posterior_median.ESTIMATED_OBJECT.VARIABLE_NAME
```

*Example*

Here are some examples of generated variables:

```
oo_.posterior_mode.parameters.alp
oo_.posterior_mean.shocks_std.ex
oo_.posterior_hpdsup.measurement_errors_corr.gdp_conso
```

**MATLAB/Octave variable: oo_.dsge_var.posterior_mode**

Structure set by the *dsge_var* option of the *estimation* command after *mode_compute*.

The following fields are saved:

`PHI_tilde`

Stacked posterior DSGE-BVAR autoregressive matrices at the mode (equation (28) of Negro and Schorfheide (2004)).

`SIGMA_u_tilde`

Posterior covariance matrix of the DSGE-BVAR at the mode (equation (29) of Negro and Schorfheide (2004)).

`iXX`

Posterior population moments in the DSGE-BVAR at the mode ( $inv(\lambda T\Gamma^*_{XX} + X'X)$).

`prior`

Structure storing the DSGE-BVAR prior.

`PHI_star`

Stacked prior DSGE-BVAR autoregressive matrices at the mode (equation (22) of Negro and Schorfheide (2004)).

`SIGMA_star`

Prior covariance matrix of the DSGE-BVAR at the mode (equation (23) of Negro and Schorfheide (2004)).

`ArtificialSampleSize`

Size of the artificial prior sample ( $inv(\lambda T)$).

`DF`

Prior degrees of freedom ( $inv(\lambda T - k - n)$).

`iGXX_star`

Inverse of the theoretical prior "covariance" between X and X ($\Gamma^*_{xx}$ in Negro and Schorfheide (2004)).

**`MATLAB/Octave variable: oo_.RecursiveForecast`**

Variable set by the *forecast* option of the *estimation* command when used with the *nobs = [INTEGER1:INTEGER2]* option.

Fields are of the form:

```
oo_.RecursiveForecast.FORECAST_OBJECT.VARIABLE_NAME
```

where `FORECAST_OBJECT` is one of the following[7] :

`Mean`

Mean of the posterior forecast distribution.

`HPDinf/HPDsup`

Upper/lower bound of the 90% HPD interval taking into account only parameter uncertainty (corresponding to *oo_.MeanForecast*).

`HPDTotalinf/HPDTotalsup`.

Upper/lower bound of the 90% HPD interval taking into account both parameter and future shock uncertainty (corresponding to *oo_.PointForecast*)

`VARIABLE_NAME` contains a matrix of the following size: number of time periods for which forecasts are requested using the *nobs = [INTEGER1:INTEGER2]* option times the number of forecast horizons requested by the forecast option. i.e., the row indicates the period at which the forecast is performed and the column the respective k-step ahead forecast. The starting periods are sorted in ascending order, not in declaration order.

**`MATLAB/Octave variable: oo_.convergence.geweke`**

Variable set by the convergence diagnostics of the *estimation* command. There is a subfield in the struct array for each MCMC chain.

Fields are of the form:

```
oo_.convergence.geweke.VARIABLE_NAME.DIAGNOSTIC_OBJECT
```

where *DIAGNOSTIC_OBJECT* is one of the following:

`posteriormean`

Mean of the posterior parameter distribution.

`posteriorstd`

Standard deviation of the posterior parameter distribution.

`nse_iid`

Numerical standard error (NSE) under the assumption of iid draws.

`rne_iid`

Relative numerical efficiency (RNE) under the assumption of iid draws.

[7] See *forecast* for more information.

`nse_taper_x`

Numerical standard error (NSE) when using an x% taper.

`rne_taper_x`

Relative numerical efficiency (RNE) when using an x% taper.

`pooled_mean`

Mean of the parameter when pooling the beginning and end parts of the chain specified in *geweke_interval* and weighting them with their relative precision. It is a vector containing the results under the iid assumption followed by the ones using the `taper_steps` option (see *taper_steps*).

`pooled_nse`

NSE of the parameter when pooling the beginning and end parts of the chain and weighting them with their relative precision. See `pooled_mean`.

`prob_chi2_test`

p-value of a chi-squared test for equality of means in the beginning and the end of the MCMC chain. See `pooled_mean`. A value above 0.05 indicates that the null hypothesis of equal means and thus convergence cannot be rejected at the 5 percent level. Differing values along the `taper_steps` signal the presence of significant autocorrelation in draws. In this case, the estimates using a higher tapering are usually more reliable.

**MATLAB/Octave variable: oo_.convergence.raftery_lewis**

Variable set by the convergence diagnostics of the *estimation* command when used with *raftery_lewis_diagnostics*. There is a subfield in the struct array for each MCMC chain. Contains the results of the test in individual fields.

### 4.16.5 Adding information about unobservable variables

The economist sometimes has some information or *a priori* knowledge about unobservable variables, that they want to incorporate in the estimation procedure. Such extra information is sometimes called "tunes". The `filter_tunes` block can be used for this purpose.

**Block: filter_tunes ;**

This block can be used to assign historical values to unobservable variables (therefore *de facto* making them observable at some periods). Optionally, a measurement error can be added.

Using this block is strictly equivalent to adding the corresponding variables to the set of observable variables with the *varobs* command, expanding the dataset for those variables (with missing observations for the periods where no tune is assigned), and optionally declaring measurement errors with the *shocks* block. Said otherwise, this block does not implement a specific algorithm, but simply provides a syntactic convenience for users who want to keep a strict formal separation between truly observable variables (and their accompanying datasets), and extra information for unobservable variables at some periods.

Tunes can be described by one or several stanzas of the following form:

```
var ENDOGENOUS_NAME;
periods INTEGER[:INTEGER] | DATE[:DATE] [, INTEGER[:INTEGER] | DATE[:DATE]]...;
values EXPRESSION [, EXPRESSION ]...;
[stderr EXPRESSION;]
```

Such a stanza describes a series of tunes on a given unobservable endogenous variable. There must be as many entries in the `periods` statement as there are in the `values` statement. Each entry gives the value of the endogenous variable over a single period or a range of periods (the latter if the syntax with a colon is used). Periods can be specified eiher as integers, or as dates. Optionally, a `stderr` statement can be used to specify a measurement error.

Note that it is assumed that tunes directly apply to the variables as defined in the model. This means that the *prefilter* and *logdata* options have no effect on tunes. As an exception, if the model is written in

nonstationary form and Dynare does the detrending (as documented in *var*), then the tunes apply to the stationarized variables.

*Example*

```
var c W gy_obs gp_obs;

...

varobs gp_obs gy_obs;

filter_tunes;
  var W;
  periods 1, 4:5;
  values 3.5, 3.75;

  var c;
  periods 2;
  values 0.5;
  stderr 0.01;
end;

calib_smoother(datafile=fsdat_simul);
```

In this example, tunes are set for the unobservable variables `W` and `c`, the latter with an additional measurement error. Note that the dataset in `fsdat_simul` contains only values for the observable variables `gp_obs` and `gy_obs`.

### 4.16.6 The data command

**Command:** **`data`**`(OPTIONS...);`

This command is just a convenience to provide the options for the dataset outside an *estimation* or *calib_smoother* command. It is functionally equivalent to specifying the corresponding options in those commands.

*Options*

**`file = FILENAME`**

Same as *datafile*. Mandatory if `series` option is not specified.

**`series = DSERIES_NAME`**

Same as *dataseries*. Mandatory if `file` option is not specified.

**`xls_sheet = QUOTED_STRING`**

See *xls_sheet = QUOTED_STRING*

**`xls_range = RANGE`**

See *xls_range = RANGE*

**`nobs = INTEGER`**

See *nobs*

**`first_obs = INTEGER | DATE`**

See *first_obs*

**`last_obs = INTEGER | DATE`**

See *last_obs*

### 4.16.7 Setting the initial state for the Kalman filter

**Block: filter_initial_state ;**

This block specifies the initial values of the endogenous states at the beginning of the Kalman filter recursions. That is, if the Kalman filter recursion starts with time t=1 being the first observation, this block provides the state estimate at time 0 given information at time 0, $E_0(x_0)$. If nothing is specified, the initial condition is assumed to be at the steady state (which is the unconditional mean for a stationary model).

This block is terminated by `end;`.

Each line inside of the block should be of the form:

```
VARIABLE_NAME(INTEGER)=EXPRESSION;
```

`EXPRESSION` is any valid expression returning a numerical value and can contain parameter values. This allows specifying relationships that will be honored during estimation. `INTEGER` refers to the lag with which a variable appears. By convention in Dynare, period 1 is the first period. Going backwards in time, the first period before the start of the simulation is period 0, then period -1, and so on. Note that the `filter_initial_state` block does not take non-state variables.

*Example*

```
filter_initial_state;
k(0)= ((1/bet-(1-del))/alp)^(1/(alp-1))*l_ss;
P(0)=2.5258;
m(0)= mst;
end;
```

### 4.16.8 Bayesian VAR estimation

Dynare also has the ability to estimate Bayesian VARs:

**Command: bvar_density ;**

Computes the marginal density of an estimated BVAR model, using Minnesota priors.

See `bvar-a-la-sims.pdf`, which comes with Dynare distribution, for more information on this command.

**Command: bvar_irf ;**

Computes the impulse responses of an estimated BVAR model, using Minnesota priors.

See `bvar-a-la-sims.pdf`, which comes with Dynare distribution, for more information on this command.

## 4.17 Estimation based on moments

Provided that you have observations on some endogenous variables or their dynamic behavior following structural shocks, Dynare provides a suite of tools for parameter estimation utilizing the method of moments approach. This includes the Simulated Method of Moments (SMM), the Generalized Method of Moments (GMM), and Impulse Response Function Matching (IRF matching). Each of these methods offers a distinct strategy for estimating some or all parameters by minimizing the distances between unconditional model objects (moments or impulse responses) and their empirical counterparts.

**GMM and SMM estimation**

For SMM Dynare computes model moments via stochastic simulations based on the perturbation approximation up to any order, whereas for GMM model moments are computed in closed-form based on the pruned state-space representation of the perturbation solution up to third order. The implementation of SMM is inspired by Born and Pfeifer (2014) and Ruge-Murcia (2012), whereas the one for GMM is adapted from Andreasen *et al.* (2018) and Mutschler (2018). Successful estimation heavily relies on the accuracy and efficiency of the perturbation approximation, so it is advised to tune this as much as possible (see *Computing the stochastic solution*). The method of moments estimator is consistent and asymptotically normally distributed given certain regularity conditions (see Duffie and Singleton (1993) for SMM and Hansen (1982) for GMM). For instance, it is required to have at least as many moment conditions as estimated parameters (over-identified or just identified). Moreover, the Jacobian of

the moments with respect to the estimated parameters needs to have full rank. *Performing identification analysis* helps to check this regularity condition.

In the over-identified case of declaring more moment conditions than estimated parameters, the choice of `weighting_matrix` matters for the efficiency of the estimation, because the estimated orthogonality conditions are random variables with unequal variances and usually non-zero cross-moment covariances. A weighting matrix allows to re-weight moments to put more emphasis on moment conditions that are more informative or better measured (in the sense of having a smaller variance). To achieve asymptotic efficiency, the weighting matrix needs to be chosen such that, after appropriate scaling, it has a probability limit proportional to the inverse of the covariance matrix of the limiting distribution of the vector of orthogonality conditions. Dynare uses a Newey-West-type estimator with a Bartlett kernel to compute an estimate of this so-called optimal weighting matrix. Note that in this over-identified case, it is advised to perform the estimation in at least two stages by setting e.g. `weighting_matrix=['DIAGONAL','DIAGONAL']` so that the computation of the optimal weighting matrix benefits from the consistent estimation of the previous stages. The optimal weighting matrix is used to compute standard errors and the J-test of overidentifying restrictions, which tests whether the model and selection of moment conditions fits the data sufficiently well. If the null hypothesis of a "valid" model is rejected, then something is (most likely) wrong with either your model or selection of orthogonality conditions.

In case the (presumed) global minimum of the moment distance function is located in a region of the parameter space that is typically considered unlikely (*dilemma of absurd parameters*), you may opt to choose the `penalized_estimator` option. Similar to adding priors to the likelihood, this option incorporates prior knowledge (i.e. the prior mean) as additional moment restrictions and weights them by their prior precision to guide the minimization algorithm to more plausible regions of the parameter space. Ideally, these regions are characterized by only slightly worse values of the objective function. Note that adding prior information comes at the cost of a loss in efficiency of the estimator.

**IRF matching**

Dynare employs a user-specified *simulation_method* to compute the impulse response function (IRF) for observable variables with respect to the structural shocks. Currently, only stochastic simulations based on the perturbation method are supported, and it is advised to fine-tune the perturbation approximation as much as possible for optimal results (see *Computing the stochastic solution* for guidance).

The core idea of IRF matching is then to treat empirical impulse responses (e.g. given from a SVAR or local projection estimation) as data and select model parameters that align the model's IRFs closely with their empirical counterparts. Dynare supports both Frequentist and Bayesian IRF matching approaches, using the same optimization and sampling techniques as those applied in likelihood-based estimation (sharing many options with the *estimation command*). The Frequentist approach to this is inspired by the work of Christiano *et al.* (2005), while the Bayesian method adapts from Christiano *et al.* (2010). A crucial element in IRF matching is the choice of the weighting matrix, which influences how the distances between model-generated and empirical IRFs are weighted in the estimation process. It is common practice to employ a diagonal weighting matrix, with the diagonal elements set to the inverse of the estimated variance of the respective empirical impulse response, thereby prioritizing more precisely estimated IRFs. While it's possible to also specify weights using covariances between different IRF components (possibly with shrinking), this is less common due to the complex interpretation involved (cross effects of different variables or different shocks or both).

Importantly, it is the user's responsibility to supply (1) the values of the empirical IRFs intended for matching and (2) their importance by choosing an appropriate weighting matrix. Dynare does not perform the SVAR or local projection estimation, it treats the empirical IRFs as given.

### 4.17.1 Method of moments specific blocks

**Command:** `varobs VARIABLE_NAME...;`

Required. All variables used in the `matched_moments`, `matched_irfs`, or `matched_irfs_weights` block need to be observable. See `varobs` for more details.

**Block:** `matched_moments ;`

This block specifies the product moments which are used in estimation. Currently, only linear product moments (e.g. $E[y_t], E[y_t^2], E[x_t y_t], E[y_t y_{t-1}], E[y_t^3 x_{t-4}^2]$) are supported. For other functions like $E[\log(y_t)e^{x_t}]$ you need to declare auxiliary endogenous variables.

Each line inside of the block should be of the form:

```
VARIABLE_NAME(LEAD/LAG)^POWER*VARIABLE_NAME(LEAD/LAG)^POWER*...*VARIABLE_
↪NAME(LEAD/LAG)^POWER;
```

where *VARIABLE_NAME* is the name of a declared observable variable, *LEAD/LAG* is either a negative integer for lags or a positive one for leads, and *POWER* is a positive integer indicating the exponent on the variable. You can omit *LEAD/LAG* equal to *0* or *POWER* equal to *1*.

*Example*

For $E[c_t], E[y_t], E[c_t^2], E[c_t y_t], E[y_t^2], E[c_t c_{t+3}], E[y_{t+1}^2 c_{t-4}^3], E[c_{t-5}^3 y_t^2]$ use the following block:

```
matched_moments;
c;
y;
c*c;
c*y;
y^2;
c*c(3);
y(1)^2*c(-4)^3;
c(-5)^3*y(0)^2;
end;
```

*Limitations*

1. For GMM, Dynare can only compute the theoretical mean, covariance, and autocovariances (i.e. first and second moments). Higher-order moments are only supported for SMM.

2. By default, the product moments are not demeaned, unless the *prefilter* option is set to 1. That is, by default, $c*c$ corresponds to $E[c_t^2]$ and not to $Var[c_t] = E[c_t^2] - E[c_t]^2$.

*Output*

Dynare translates the *matched_moments* block into a cell array `M_.matched_moments` where:

- the first column contains a vector of indices for the chosen variables in declaration order
- the second column contains the corresponding vector of leads and lags
- the third column contains the corresponding vector of powers

During the estimation phase, Dynare will eliminate all redundant or duplicate orthogonality conditions in `M_.matched_moments` and display which conditions were removed. In the example above, this would be the case for the last row, which is the same as the second-to-last one. The original block is saved in `M_.matched_moments_orig`.

**Block:** `matched_irfs ;`

**Block:** `matched_irfs(overwrite);`

This block specifies the values and diagonal weights of the empirical IRFs that are matched in estimation. The `overwrite` option replaces the current `matched_irfs` block with the new one.

Each line inside of the block should be of the form:

```
var ENDOGENOUS_NAME;
varexo EXOGENOUS_NAME;
periods INTEGER[:INTEGER] [[,] INTEGER[:INTEGER]]...;
values DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
weights DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
```

*ENDOGENOUS_NAME* is the name of a declared observable variable, whereas *EXOGENOUS_NAME* is the name of an exogenous variable. It is possible to specify individual horizons or a range of specified periods as lists with the `periods` keyword. Note that for each entry a corresponding entry in `values` needs to be provided; that is `values` is a list of the same length as `periods`. If only one value is specified, it is used at all corresponding `periods` in the list. `weights` are optional and specify the diagonal element of

the corresponding entry in the weighting matrix. Typically, these are set to the inverse of the variance of the empirical IRF. If only one weight is specified, it is used at all corresponding `periods` in the list. If not specified, the weight defaults to 1. For `values` and `weights` you can use expressions (e.g. variables or anonymous functions in the workspace) by by putting paranthesis around them. A new statement is started with either the `var` or `varexo` keyword.

*Example*

You can either enter the values directly or load them from variables in the workspace.

```
% MATLAB expressions that can be used
xx = [23,24,25];
ww = [51,52];
irfs_eR  = @(j) IRFFF(2:15,j); % gdp is the 3th column of IRFFF
weights_eR  = @(j) 1./(IRFFFSE(2:15,j).^2);
R_eR = IRFFF(1:15,3);
weight_R_eR = 1./(IRFFFSE(1:15,3).^2);

matched_irfs;
var gdp;  varexo eR;  periods 2:15; values (irfs_eR(3));  weights␣
↪(weights_eR(3));
var R;    varexo eR;  periods 1:15; values (R_eR);        weights␣
↪(weight_R_eR);
var y;    varexo eD;  periods 5;    values 7;             weights 25;
var r;    varexo eD;  periods 1,2;  values 17,18;         weights 37,38;
var c;    varexo eA;  periods 3:5;  values (xx);
var y;    varexo eA;  periods 1:2;  values 30;            weights (ww);

varexo eR;
var w;
periods 1, 13:15, 2:12;
values 2, (xx), 15;
weights 3, (xx), 4;
end;
```

*Output*

Dynare translates the *matched_irfs* block into a cell array where the rows correspond to the statements in the block `M_.matched_irfs` where:

- the first column contains the names of the endogenous variables
- the second column contains the names of the exogenous variables
- the third column contains a nested cell array that contains the list of horizons, values and weights.

**Block:** **matched_irfs_weights** ;

**Block:** **matched_irfs_weights**(overwrite) ;

This optional block specifies elements of the weighting matrix used for IRF matching. The `overwrite` option replaces the current `matched_irfs_weights` block with the new one.

The weighting matrix is initialized as a diagonal matrix with ones on the diagonal. Each line inside of the block should be of the form:

```
ENDOGENOUS_NAME_1(HORIZON_1), EXOGENOUS_NAME_1, ENDOGENOUS_NAME_2(HORIZON_2),␣
↪EXOGENOUS_NAME_2, WEIGHT;
```

where *ENDOGENOUS_NAME_1* and *ENDOGENOUS_NAME_2* are the names of declared observable variables, *EXOGENOUS_NAME_1* and *EXOGENOUS_NAME_2* are the names of exogenous variables, *HORIZON_1* and *HORIZON_2* are integers indicating the horizon of the IRFs and *WEIGHT* is a double value of the weight one wants to assign to the covariance between the two specified IRFs.

*Example*

You can either enter the values directly or load them from variables in the workspace.

```
matched_irfs_weights;
c(1), e_A, c(1), e_A, 20;
y(3), e_R, y(2), e_R, (empIRFsCovInv_yR3_yR2);
end;
```

*Output*

Dynare translates the *matched_irfs_weights* block into a cell array `M_.matched_irfs_weights` where:

- the first column contains the names of the first endogenous variables
- the second column contains the names of the first exogenous variables
- the third column contains the horizons of the IRFs for the first endogneous variable
- the fourth column contains the names of the second endogenous variables
- the fifth column contains the names of the second exogenous variables
- the sixth column contains the horizons of IRFs for the second endogenous variable
- the seventh column contains the vector of weights

All values that are not specified will be either one (if they are on the diagonal) or zero (if they are not on the diagonal). Symmetry is respected, so one does not need to specify both `c(1), e_A, y(3), e_R, WEIGHT` and `y(3), e_R, c(1), e_A, WEIGHT`. Default: empty cell.

**Block: estimated_params ;**

Required. This block lists all parameters to be estimated and specifies bounds and priors as necessary. See *estimated_params* for details and syntax.

**Block: estimated_params_init ;**

Optional. This block declares numerical initial values for the optimizer when these ones are different from the prior mean. See *estimated_params_init* for details and syntax.

**Block: estimated_params_bounds ;**

Optional. This block declares lower and upper bounds for parameters in maximum likelihood estimation. See *estimated_params_bounds* for details and syntax.

## 4.17.2 method_of_moments command

**Command: method_of_moments**(OPTIONS...);

This command runs the method of moments estimation. The following information will be displayed in the command window:

- Overview of options chosen by the user
- Estimation results for each stage and iteration
- Value of minimized moment distance objective function
- Result of the J-test (for SMM/GMM)
- Comparison plot of model IRFs and empirical IRFs (for IRF matching)
- Table of data moments/IRFs and estimated model moments/IRFs

### 4.17.2.1 Necessary Options

**mom_method = SMM|GMM|IRF_MATCHING**

"Simulated Method of Moments" is triggered by *SMM*, "Generalized Method of Moments" by *GMM* and "Impulse Response Function Matching" by *IRF_MATCHING*.

**datafile = FILENAME**

The name of the file containing the data (for GMM and SMM only). See *datafile* for the meaning and syntax. For IRF matching, the data is specified in the *matched_irfs* block.

#### 4.17.2.2 Common Options

**order = INTEGER**

Order of perturbation approximation. For GMM only orders 1|2|3 are supported. For SMM and IRF matching, you can choose an arbitrary order. Note that the order set in other functions will not overwrite the default. Default: 1.

**pruning**

Discard higher-order terms when iteratively computing simulations of the solution. See *pruning* for more details. Default: not set for SMM and IRF matching, always set for GMM.

**verbose**

Display and store intermediate estimation results in `oo_.mom`. Default: not set.

*Common options for SMM and GMM*

**penalized_estimator**

This option includes deviations of the estimated parameters from the prior mean as additional moment restrictions and weights them by their prior precision. Default: not set.

**weighting_matrix = ['WM1','WM2',...,'WMn']**

Determines the weighting matrix used at each estimation stage. The number of elements will define the number of stages, i.e. `weighting_matrix = ['DIAGONAL','DIAGONAL', 'OPTIMAL']` performs a three-stage estimation. Possible values for `WM` are:

`IDENTITY_MATRIX`

Sets the weighting matrix equal to the identity matrix.

`OPTIMAL`

Uses the optimal weighting matrix computed by a Newey-West-type estimate with a Bartlett kernel. At the first stage, the data-moments are used as initial estimate of the model moments, whereas at subsequent stages the previous estimate of model moments will be used when computing the optimal weighting matrix.

`DIAGONAL`

Uses the diagonal of the `OPTIMAL` weighting matrix. This choice puts weights on the specified moments instead of on their linear combinations.

`FILENAME`

The name of the MAT-file (extension `.mat`) containing a user-specified weighting matrix. The file must include a positive definite square matrix called `weighting_matrix` with both dimensions equal to the number of orthogonality conditions.

Default value is `['DIAGONAL','OPTIMAL']`.

**weighting_matrix_scaling_factor = DOUBLE**

Scaling of weighting matrix in objective function. This value should be chosen to obtain values of the objective function in a reasonable numerical range to prevent over- and underflows. Default: 1.

**bartlett_kernel_lag = INTEGER**

Bandwidth of kernel for computing the optimal weighting matrix. Default: `20`.

**se_tolx = DOUBLE**

Step size for numerical differentiation when computing standard errors with a two-sided finite difference method. Default: `1e-5`.

#### 4.17.2.3 SMM specific options

**burnin = INTEGER**

Number of periods dropped at the beginning of simulation. Default: `500`.

**bounded_shock_support**

Trim shocks in simulations to $\pm 2$ standard deviations. Default: not set.

**seed = INTEGER**

Common seed used in simulations. Default: `24051986`.

**simulation_multiple = INTEGER**

Multiple of data length used for simulation. Default: `7`.

**hp_filter = DOUBLE**

See *hp_filter = DOUBLE*

**one_sided_hp_filter = DOUBLE**

See *one_sided_hp_filter = DOUBLE*

**bandpass_filter**

See *bandpass_filter*

**bandpass_filter = [HIGHEST_PERIODICITY LOWEST_PERIODICITY]**

See *bandpass_filter = [HIGHEST_PERIODICITY LOWEST_PERIODICITY]*

#### 4.17.2.4 GMM specific options

**analytic_standard_errors**

Compute standard errors using analytical derivatives of moments with respect to estimated parameters. Default: not set, i.e. standard errors are computed using a two-sided finite difference method, see *se_tolx*.

#### 4.17.2.5 IRF matching specific options

**simulation_method = METHOD**

Method to compute IRFs. Possible values for `METHOD` are:

`STOCH_SIMUL`

Simulate the model with stochastic simulations and compute IRFs as the difference between the simulated and steady-state values. See *stoch_simul* for more details.

**irf_matching_file = FILENAME**

A MATLAB file containing additional transformations on the model IRFs. This enables more flexibility in matching the model IRFs to the empirical IRFs, e.g. by adding constants to model IRFs, multiplying them with factors, taking the cumulative sum, creating ratios etc. See `NK_irf_matching_file.m` in the examples directory for an example. Default: empty, i.e. model IRFs exactly match empirical IRFs.

**add_tiny_number_to_cholesky = DOUBLE**

In case of a non-positive definite covariance matrix, a tiny number is added to the Cholesky factor to avoid numerical problems when computing IRFs. Default: *1e-14*.

**drop = INTEGER**

Truncation when computing IRFs with perturbation at orders greater than 1. Default: `100`.

**relative_irf**

Requests the computation of normalized IRFs. See *relative_irf* for more details. Default: false.

**replic = INTEGER**

Number of simulated series used to compute the IRFs. Default: `1` if `order=1`, and `50` otherwise.

#### 4.17.2.6 General options

**dirname = FILENAME**

Directory in which to store *estimation* output. See *dirname* for more details. Default: `<mod_file>`.

**graph_format = FORMAT**

Specify the file format(s) for graphs saved to disk. See *graph_format* for more details. Default: `eps`.

**nodisplay**

See *nodisplay*. Default: not set.

**nograph**

See *nograph*. Default: not set.

**noprint**

See *noprint*. Default: not set.

**plot_priors = INTEGER**

Control the plotting of priors. See *plot_priors* for more details. Default: `1`, i.e. plot priors.

**prior_trunc = DOUBLE**

See *prior_trunc* for more details. Default: `1e-10`.

**tex**

See *tex*. Default: not set.

#### 4.17.2.7 Data options

**prefilter = INTEGER**

A value of 1 means that the estimation procedure will demean each data series by its empirical mean and each model moment by its theoretical mean. See *prefilter* for more details. Default: `0`, i.e. no prefiltering.

**first_obs = INTEGER**

See *first_obs*. Default: 1.

**nobs = INTEGER**

See *nobs*. Default: all observations are considered.

**logdata**

See *logdata*. Default: not set.

**xls_sheet = QUOTED_STRING**

See *xls_sheet*. Default: 1.

**xls_range = RANGE**

See *xls_range*. Default: empty.

#### 4.17.2.8 Optimization options

**mode_file = FILENAME**

Name of the file containing previous value for the mode. See *mode_file*. Default: empty.

**mode_compute = INTEGER | FUNCTION_NAME**

See *mode_compute*. Default: 13 for GMM and SMM and 5 for IRF matching.

**additional_optimizer_steps = [INTEGER]**

**additional_optimizer_steps = [INTEGER1:INTEGER2]**

**additional_optimizer_steps = [INTEGER1 INTEGER2]**

Vector of additional minimization algorithms run after mode_compute. If *verbose* option is set, then the additional estimation results are saved into the oo_.mom structure prefixed with *verbose_*. Default: empty, i.e. no additional optimization iterations.

**optim = (NAME, VALUE, ...)**

See *optim*. Default: empty.

**analytic_jacobian**

Use analytic Jacobian in optimization, only available for GMM and gradient-based optimizers. Default: not set.

**huge_number = DOUBLE**

See *huge_number*. Default: 1e7.

**silent_optimizer**

**silent_optimizer = BOOLEAN**

See *silent_optimizer*. Default: not set.

**use_penalized_objective_for_hessian**

See *use_penalized_objective_for_hessian*. Default: not set.

#### 4.17.2.9 Bayesian estimation options

**General options**

**posterior_sampling_method = NAME**

See *posterior_sampling_method*. Default: random_walk_metropolis_hastings.

**posterior_sampler_options = (NAME, VALUE, ...)**

See *posterior_sampler_options*. Default: not set.

**mh_posterior_mode_estimation**

See *mh_posterior_mode_estimation*. Default: not set.

**cova_compute = INTEGER**

See *cova_compute*. Default: 1.

**mcmc_jumping_covariance = OPTION**

See *mcmc_jumping_covariance*. Default: hessian.

**mh_replic = INTEGER**

See *mh_replic*. Default: 0.

**mh_nblocks = INTEGER**

See *mh_nblocks*. Default: 2.

**mh_jscale = DOUBLE**

See *mh_jscale*. Default: 2.38 divided by the square root of the number of estimated parameters.

**mh_tune_jscale [= DOUBLE]**

See *mh_tune_jscale*. Default: `0.33`.

**mh_tune_guess = DOUBLE**

See *mh_tune_guess*. Default: `2.38` divided by the square root of the number of estimated parameters.

**mh_conf_sig = DOUBLE**

See *mh_conf_sig*. Default: `0.9`.

**mh_drop = DOUBLE**

See *mh_drop*. Default: `0.5`.

**mh_init_scale_factor = DOUBLE**

See *mh_init_scale_factor*. Default: `2`.

**no_posterior_kernel_density**

See *no_posterior_kernel_density*. Default: not set.

**posterior_max_subsample_draws = INTEGER**

See *posterior_max_subsample_draws*. Default: `1200`.

**sub_draws = INTEGER**

See *sub_draws*. Default: `min(posterior_max_subsample_draws, (Total number of draws)*(number of chains) )`.

**use_pct**

**use_pct = BOOLEAN**

**use_pct = [QUOTED_STRING ...]**

Controls whether to use MATLAB's Parallel Computing Toolbox (PCT). Requires MATLAB R2024b or later, the PCT to be installed, and a valid license. The pool is automatically managed: if no pool is open when entering the sampler, one is created and closed afterwards; if a pool is already open, it is reused. This option has no effect under Octave.

When used without a value or set to `true`, all PCT-compatible sub-tasks of the estimation are run in parallel. When set to `false`, all sub-tasks run serially. When a list of quoted strings is provided, only the listed sub-tasks are parallelized with PCT while the others run serially.

Currently, the recognized sub-tasks are:

`'sampler'`

Parallelize the MCMC posterior samplers. Each chain (see *mh_nblocks*) is dispatched to a separate worker using `parfeval`. Per chain random number generator seeds are pre-computed so that results are identical whether running in serial or parallel.

`'smc_initialization'`

Parallelize the initialization of SMC samplers (HSSMC, DSMH, DIME). The initial particle draws from the prior distribution are distributed across workers using `parfor`. Per-particle random number generator seeds are pre-computed so that results are identical whether running in serial or parallel.

`'dime'`

Parallelize the likelihood evaluations within the DIME MCMC sampler iterations. Each candidate proposal's log-posterior is evaluated in parallel using `parfor`.

This option provides a simpler alternative to the cluster-based parallelization configured via *Parallel Execution and Configuration File* and the `parallel` command-line option. Unlike the cluster-based approach, it does not require a configuration file, SSH setup, or PsTools. It leverages MATLAB's built-in parallel pool and works out of the box on a single multi-core machine.

*Examples*

```
estimation(use_pct);
estimation(use_pct=true);
estimation(use_pct=false);
estimation(use_pct=['sampler']);
estimation(use_pct=['smc_initialization']);
estimation(use_pct=['smc_initialization','dime']);
```

Default: `true` (all sub-tasks parallelized when PCT is available and the license is valid).

**MCMC initialization and recovery**

**load_mh_file**

See *load_mh_file*. Default: not set.

**load_results_after_load_mh**

See *load_results_after_load_mh*. Default: not set.

**mh_initialize_from_previous_mcmc**

See *mh_initialize_from_previous_mcmc*. Default: not set.

**mh_initialize_from_previous_mcmc_directory = FILENAME**

See *mh_initialize_from_previous_mcmc_directory*. Default: empty.

**mh_initialize_from_previous_mcmc_prior = FILENAME**

See *mh_initialize_from_previous_mcmc_prior*. Default: empty.

**mh_initialize_from_previous_mcmc_record = FILENAME**

See *mh_initialize_from_previous_mcmc_record*. Default: empty.

**mh_recover**

See *mh_recover*. Default: not set.

**Convergence diagnostics**

**nodiagnostic**

See *nodiagnostic*. Default: not set.

**brooks_gelman_plotrows = INTEGER**

See *brooks_gelman_plotrows*. Default: `3`.

**geweke_interval = [DOUBLE DOUBLE]**

See *geweke_interval*. Default: `[0.2 0.5]`.

**taper_steps = [INTEGER1 INTEGER2 ...]**

See *taper_steps*. Default: `[4 8 15]`.

**raftery_lewis_diagnostics**

See *raftery_lewis_diagnostics*. Default: not set.

**raftery_lewis_qrs = [DOUBLE DOUBLE DOUBLE]**

See *raftery_lewis_qrs*. Default: `[0.025 0.005 0.95]`.

#### 4.17.2.10 Numerical algorithms options

**k_order_solver**

See *k_order_solver*. Default: disabled for order 1 and 2, enabled for order 3 and above.

**dr = OPTION**

See *dr*. Default: `default`, i.e. generalized Schur decomposition.

**dr_cycle_reduction_tol = DOUBLE**

See *dr_cycle_reduction_tol*. Default: `1e-7`.

**dr_cycle_reduction_maxiter = INTEGER**

See *dr_cycle_reduction_maxiter*. Default: `100`.

**dr_logarithmic_reduction_tol = DOUBLE**

See *dr_logarithmic_reduction_tol*. Default: `1e-12`.

**dr_logarithmic_reduction_maxiter = INTEGER**

See *dr_logarithmic_reduction_maxiter*. Default: `100`.

**lyapunov = OPTION**

See *lyapunov*. Default: `default`, i.e. based on Bartlets-Stewart algorithm.

**lyapunov_complex_threshold = DOUBLE**

See *lyapunov_complex_threshold*. Default: `1e-15`.

**lyapunov_fixed_point_tol = DOUBLE**

See *lyapunov_fixed_point_tol*. Default: `1e-10`.

**lyapunov_doubling_tol = DOUBLE**

See *lyapunov_doubling_tol*. Default: `1e-16`.

**qz_criterium = DOUBLE**

See *qz_criterium*. For unit roots (only possible at order=1) set e.g. to 1.000001. Default: `0.999999` as it is assumed that the observables are weakly stationary.

**qz_zero_threshold = DOUBLE**

See *qz_zero_threshold*. Default: `1e-6`.

**schur_vec_tol = DOUBLE**

Tolerance level used to find nonstationary variables in Schur decomposition of the transition matrix. Default: `1e-11`.

**mode_check**

Plots univariate slices through the moments distance objective function around the computed minimum for each estimated parameter. This is helpful to diagnose problems with the optimizer. Default: not set.

**mode_check_neighbourhood_size = DOUBLE**

See *mode_check_neighbourhood_size*. Default: `0.5`.

**mode_check_symmetric_plots = INTEGER**

See *mode_check_symmetric_plots*. Default: `1`.

**mode_check_number_of_points = INTEGER**

See *mode_check_number_of_points*. Default: `20`.

### 4.17.3 Method of moments specific outputs

`method_of_moments` stores user options in a structure called *options_mom_* in the global workspace. After running the estimation, the parameters `M_.params` and the covariance matrices of the shocks `M_.Sigma_e` and of the measurement errors `M_.H` are set to the parameters that either minimize the quadratic moments distance objective function or at the posterior mean in case of Bayesian MCMC estimation. The estimation results are stored in a subfolder of *dirname* called *method_of_moments*. Moreover, output is stored in the `oo_.mom` structure with the following fields:

**Common outputs**

**MATLAB/Octave variable: oo_.mom.data_moments**

Variable set by the `method_of_moments` command. Stores the mean of the selected empirical moments/IRFs of data. NaN values due to leads/lags or missing data are omitted when computing the mean for moments. Vector of dimension equal to the number of orthogonality conditions or IRFs.

**MATLAB/Octave variable: oo_.mom.model_moments**

Variable set by the `method_of_moments` command. Stores the implied selected model moments or IRFs given the current parameter guess. Model moments are computed in closed-form from the pruned state-space system for GMM, whereas for SMM these are based on averages of simulated data. Model IRFs are computed from the specified *simulation_method*. Vector of dimension equal to the number of orthogonality conditions.

**MATLAB/Octave variable: oo_.mom.model_moments_params_derivs**

Variable set by the `method_of_moments` command. Stores the analytically computed Jacobian matrix of the derivatives of the model moments with respect to the estimated parameters. Only for GMM with *analytic_standard_errors*. Matrix with dimension equal to the number of orthogonality conditions times number of estimated parameters.

**MATLAB/Octave variable: oo_.mom.weighting_info**

Variable set by the `method_of_moments` command. Stores the currently used weighting matrix (*W*), its Cholesky factor (*Sw*), and an indicator whether the weighting matrix is the optimal one (*Woptflag*). The inverse (*Winv*) and its log determinant (*Winv_logdet*) are also stored.

**MATLAB/Octave variable: oo_.mom.Q**

Variable set by the `method_of_moments` command. Stores the scalar value of the quadratic moment's distance objective function.

**MATLAB/Octave variable: oo_.mom.verbose**

Structure that contains intermediate estimation results if `verbose` is used.

**SMM and GMM specific outputs**

**MATLAB/Octave variable: oo_.mom.m_data**

Variable set by the `method_of_moments` command. Stores the selected empirical moments at each point in time. NaN values due to leads/lags or missing data are replaced by the corresponding mean of the moment. Matrix of dimension time periods times number of orthogonality conditions.

**MATLAB/Octave variable: oo_.mom.gmm_mode**

**MATLAB/Octave variable: oo_.mom.smm_mode**

Variables set by the `method_of_moments` command when estimating with GMM or SMM. Stores the estimated values of the final stage. The structures contain the following fields:

- `measurement_errors_corr`: estimated correlation between two measurement errors
- `measurement_errors_std`: estimated standard deviation of measurement errors
- `parameters`: estimated model parameters
- `shocks_corr`: estimated correlation between two structural shocks.
- `shocks_std`: estimated standard deviation of structural shocks.

**MATLAB/Octave variable: oo_.mom.gmm_std_at_mode**

**MATLAB/Octave variable: oo_.mom.smm_std_at_mode**

Variables set by the `method_of_moments` command when estimating with GMM or SMM. Stores the estimated standard errors of the final stage. The structures contain the following fields:

- `measurement_errors_corr`: standard error of estimated correlation between two measurement errors

- `measurement_errors_std`: standard error of estimated standard deviation of measurement errors
- `parameters`: standard error of estimated model parameters
- `shocks_corr`: standard error of estimated correlation between two structural shocks.
- `shocks_std`: standard error of estimated standard deviation of structural shocks.

**MATLAB/Octave variable: oo_.mom.J_test**

Variable set by the `method_of_moments` command. Structure where the value of the test statistic is saved into a field called `j_stat`, the degress of freedom into a field called `degrees_freedom` and the p-value of the test statistic into a field called `p_val`.

**IRF matching specific outputs**

**MATLAB/Octave variable: oo_.mom.irf_model_varobs**

Variable set by the `method_of_moments` command. Stores all the implied model impulse response functions (not only the matched ones) and is used for the comparison plot. Array of dimension equal to number of observables by number of shocks by maximum horizon.

**Bayesian specific outputs**

**MATLAB/Octave variable: oo_.mom.prior**

Variable set by the `method_of_moments` command if Bayesian estimation is used. Stores information of the joint prior. Fields are of the form:

```
oo_.mom.prior.OBJECT
```

where OBJECT is one of the following:

`mean`

Prior mean parameter vector.

`mode`

Prior mode parameter vector.

`variance`

Covariance matrix of joint prior.

`hyperparameters`

Vectors of hyperparameters of the prior distributions stored in fields `first` and `second`.

**MATLAB/Octave variable: oo_.mom.posterior.optimization**

Variable set by the `method_of_moments` command if mode-finding is used. Stores the results at the mode. Fields are of the form:

```
oo_.mom.posterior.optimization.OBJECT
```

where OBJECT is one of the following:

`mode`

Parameter vector at the mode.

`Variance`

Inverse Hessian matrix at the mode or MCMC jumping covariance matrix when used with the *`MCMC_jumping_covariance`* option.

`log_density`

Log likelihood (ML)/log posterior density (Bayesian) at the mode when used with `mode_compute>0`.

**MATLAB/Octave variable: oo_.mom.posterior.metropolis**

Variable set by the method_of_moments command if mh_replic>0 is used. Fields are of the form:

```
oo_.mom.posterior.metropolis.OBJECT
```

where OBJECT is one of the following:

mean

Mean parameter vector from the MCMC.

Variance

Covariance matrix of the parameter draws in the MCMC.

**MATLAB/Octave variable: oo_.mom.prior_density**

Variable set by the method_of_moments command, if it is used with mh_replic > 0 or load_mh_file option. Fields are of the form:

```
oo_.mom.prior_density.PARAMETER_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_density**

Variable set by the method_of_moments command, if it is used with mh_replic > 0 or load_mh_file option. Fields are of the form:

```
oo_.mom.posterior_density.PARAMETER_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_hpdinf**

Variable set by the method_of_moments command, if it is used with mh_replic > 0 or load_mh_file option. Fields are of the form:

```
oo_.mom.posterior_hpdinf.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_hpdsup**

Variable set by the method_of_moments command, if it is used with mh_replic > 0 or load_mh_file option. Fields are of the form:

```
oo_.mom.posterior_hpdsup.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_mean**

Variable set by the method_of_moments command, if it is used with mh_replic > 0 or load_mh_file option. Fields are of the form:

```
oo_.posterior_mean.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_mode**

Variable set by the method_of_moments command during mode-finding. Fields are of the form:

```
oo_.mom.posterior_mode.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_std_at_mode**

Variable set by the method_of_moments command during mode-finding. It is based on the inverse Hessian at oo_.mom.posterior_mode. Fields are of the form:

```
oo_.mom.posterior_std_at_mode.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_std**

Variable set by the method_of_moments command, if it is used with mh_replic > 0 or load_mh_file option. Fields are of the form:

```
oo_.mom.posterior_std.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_variance**

Variable set by the `method_of_moments` command, if it is used with `mh_replic` > `0` or `load_mh_file` option. Fields are of the form:

```
oo_.mom.posterior_variance.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_median**

Variable set by the `method_of_moments` command, if it is used with `mh_replic` > `0` or `load_mh_file` option. Fields are of the form:

```
oo_.mom.posterior_median.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.mom.posterior_deciles**

Variable set by the `method_of_moments` command, if it is used with `mh_replic` > `0` or `load_mh_file` option. Fields are of the form:

```
oo_.mom.posterior_deciles.ESTIMATED_OBJECT.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.mom.MarginalDensity.LaplaceApproximation**

Variable set by the `method_of_moments` command. Stores the marginal data density based on the Laplace Approximation.

**MATLAB/Octave variable: oo_.mom.MarginalDensity.ModifiedHarmonicMean**

Variable set by the `method_of_moments` command, if it is used with `mh_replic` > `0` or `load_mh_file` option. Stores the marginal data density based on Geweke (1999) Modified Harmonic Mean estimator.

## 4.18 Model Comparison

**Command: model_comparison** `FILENAME[(DOUBLE)]...;`

**Command: model_comparison**`(marginal_density = ESTIMATOR) FILENAME[(DOUBLE)]...;`

This command computes odds ratios and estimate a posterior density over a collection of models (see e.g. Koop (2003), Ch. 1). The priors over models can be specified as the *DOUBLE* values, otherwise a uniform prior over all models is assumed. In contrast to frequentist econometrics, the models to be compared do not need to be nested. However, as the computation of posterior odds ratios is a Bayesian technique, the comparison of models estimated with maximum likelihood is not supported.

It is important to keep in mind that model comparison of this type is only valid with proper priors. If the prior does not integrate to one for all compared models, the comparison is not valid. This may be the case if part of the prior mass is implicitly truncated because Blanchard and Kahn conditions (instability or indeterminacy of the model) are not fulfilled, or because for some regions of the parameters space the deterministic steady state is undefined (or Dynare is unable to find it). The compared marginal densities should be renormalized by the effective prior mass, but this not done by Dynare: it is the user's responsibility to make sure that model comparison is based on proper priors. Note that, for obvious reasons, this is not an issue if the compared marginal densities are based on Laplace approximations.

*Options*

**marginal_density = ESTIMATOR**

Specifies the estimator for computing the marginal data density. *ESTIMATOR* can take one of the following two values: `laplace` for the Laplace estimator or `modifiedharmonicmean` for the Geweke (1999) Modified Harmonic Mean estimator. Default value: `laplace`

*Output*

The results are stored in `oo_.Model_Comparison`, which is described below.

*Example*

```
model_comparison my_model(0.7) alt_model(0.3);
```

This example attributes a 70% prior over `my_model` and 30% prior over `alt_model`.

**MATLAB/Octave variable: oo_.Model_Comparison**

Variable set by the `model_comparison` command. Fields are of the form:

```
oo_.Model_Comparison.FILENAME.VARIABLE_NAME
```

where FILENAME is the file name of the model and VARIABLE_NAME is one of the following:

`Prior`

(Normalized) prior density over the model.

`Log_Marginal_Density`

Logarithm of the marginal data density.

`Bayes_Ratio`

Ratio of the marginal data density of the model relative to the one of the first declared model

`Posterior_Model_Probability`

Posterior probability of the respective model.

## 4.19 Shock Decomposition

**Command: shock_decomposition** [VARIABLE_NAME]...;

**Command: shock_decomposition**(OPTIONS...) [VARIABLE_NAME]...;

This command computes the historical shock decomposition for a given sample based on the Kalman smoother, i.e. it decomposes the historical deviations of the endogenous variables from their respective steady-state values into the contribution coming from the various shocks. The `variable_names` provided govern for which variables the decomposition is plotted.

Note that, in case of an estimated model, this command must come after *estimation*.

Forecasts can also be shock decomposed. That applies to both unconditional forecast generated by the *forecast* command, the *estimation* with the *forecast* option and conditional forecasts generated by the *conditional_forecast* command. Note that these forecasts need to have been computed before calling `shock_decomposition`.

*Options*

**parameter_set = OPTION**

Specify the parameter set to use for running the smoother. Possible values for OPTION are:

- `calibration`
- `prior_mode`
- `prior_mean`
- `posterior_mode`
- `posterior_mean`
- `posterior_median`
- `mle_mode`

Note that the parameter set used in subsequent commands like *stoch_simul* will be set to the specified `parameter_set`. Default value: `posterior_mean` if Metropolis has been run, `mle_mode` if MLE has been run.

**datafile = FILENAME**

See *datafile*. Useful when computing the shock decomposition on a calibrated model.

**first_obs = INTEGER**

See *first_obs*.

**nobs = INTEGER**

See *nobs*.

**prefilter = INTEGER**

See *prefilter*.

**loglinear**

See *loglinear*.

**diffuse_kalman_tol = DOUBLE**

See *diffuse_kalman_tol*.

**diffuse_filter**

See *diffuse_filter*.

**kalman_algo = INTEGER**

See *kalman_algo = INTEGER*.

**kalman_tol = DOUBLE**

See *kalman_tol = DOUBLE*.

**use_univariate_filters_if_singularity_is_detected**

**use_univariate_filters_if_singularity_is_detected = BOOLEAN**

See *use_univariate_filters_if_singularity_is_detected*.

**xls_sheet = QUOTED_STRING**

See *xls_sheet*.

**xls_range = RANGE**

See *xls_range*.

**use_shock_groups [= NAME]**

Uses shock grouping defined by the string instead of individual shocks in the decomposition. The groups of shocks are defined in the *shock_groups* block. If no group name is given, `default` is assumed.

**colormap = VARIABLE_NAME**

Controls the `colormap` used for the shocks decomposition graphs. VARIABLE_NAME must be the name of a MATLAB/Octave variable that has been declared beforehand and whose value will be passed to the MATLAB/Octave `colormap` function (see the MATLAB/Octave manual for the list of acceptable values).

**nograph**

See *nograph*. Suppresses the display and creation only within the `shock_decomposition` command, but does not affect other commands. See *plot_shock_decomposition* for plotting graphs.

**init_state = BOOLEAN**

If equal to 0, the shock decomposition is computed conditional on the smoothed state variables in period `0`, i.e. the smoothed shocks starting in period 1 are used. If equal to `1`, the shock decomposition is computed conditional on the smoothed state variables in period 1. Default: `0`.

**with_epilogue**

If set, then also compute the decomposition for variables declared in the `epilogue` block (see *Epilogue Variables*).

**forecast_type = unconditional | conditional**

Controls the type of forecast to be decomposed. For both types, the historical shock decomposition for the given sample based on the Kalman smoother. In case of unconditional forecasts, the ones generated by the *forecast* command will be decomposed if `parameter_set=calibration`, and mean forecasts generated by the *estimation* command otherwise. If this option is specified, the shock decomposition plot displays the historical and the forecast periods, with a vertical bar separating the two periods. Default: not set, i.e. only historical shock decomposition is computed.

*Output*

**MATLAB/Octave variable: oo_.shock_decomposition**

The results are stored in the field `oo_.shock_decomposition`, which is a three dimensional array. The first dimension contains the `M_.endo_nbr` endogenous variables. The second dimension stores in the first `M_.exo_nbr` columns the contribution of the respective shocks. Column `M_.exo_nbr+1` stores the contribution of the initial conditions, while column `M_.exo_nbr+2` stores the smoothed value of the respective endogenous variable in deviations from their steady state, i.e. the mean and trends are subtracted. The third dimension stores the time periods. Both the variables and shocks are stored in the order of declaration, i.e. `M_.endo_names` and `M_.exo_names`, respectively.

**MATLAB/Octave variable: oo_.forecast_shock_decomposition**

The results from the forecast shock decomposition are stored in the field `oo_.forecast_shock_decomposition`. It is a structure that will have a field `unconditional` for unconditional forecasts and a field `conditional` for conditional ones. Each field is a three dimensional array, where the third dimension stores the time periods, including the forecast horizon.

**Block: shock_groups ;**

**Block: shock_groups**(OPTIONS...);

Shocks can be regrouped for the purpose of shock decomposition. The composition of the shock groups is written in a block delimited by `shock_groups` and `end`.

Each line defines a group of shocks as a list of exogenous variables:

```
SHOCK_GROUP_NAME   = VARIABLE_1 [[,] VARIABLE_2 [,]...];
'SHOCK GROUP NAME' = VARIABLE_1 [[,] VARIABLE_2 [,]...];
```

*Options*

**name = NAME**

Specifies a name for the following definition of shock groups. It is possible to use several `shock_groups` blocks in a model file, each grouping being identified by a different name. This name must in turn be used in the `shock_decomposition` command. If no name is given, `default` is used.

*Example*

```
varexo e_a, e_b, e_c, e_d;
...

shock_groups(name=group1);
supply = e_a, e_b;
'aggregate demand' = e_c, e_d;
end;

shock_decomposition(use_shock_groups=group1);
```

This example defines a shock grouping with the name `group1`, containing a set of supply and demand shocks and conducts the shock decomposition for these two groups.

**Command: realtime_shock_decomposition** [VARIABLE_NAME]...;

**Command: realtime_shock_decomposition**(OPTIONS...) [VARIABLE_NAME]...;

This command computes the realtime historical shock decomposition for a given sample based on the Kalman smoother. For each period $T = [\texttt{presample}, \ldots, \texttt{nobs}]$, it recursively computes three objects:

- Real-time historical shock decomposition $Y(t|T)$ for $t = [1, \ldots, T]$, i.e. without observing data in $[T+1, \ldots,$ `nobs`$]$. This results in a standard shock decomposition being computed for each additional datapoint becoming available after `presample`.
- Forecast shock decomposition $Y(T+k|T)$ for $k = [1, \ldots, forecast]$, i.e. the $k$-step ahead forecast made for every $T$ is decomposed in its shock contributions.
- Real-time conditional shock decomposition of the difference between the real-time historical shock decomposition and the forecast shock decomposition. If *vintage* is equal to `0`, it computes the effect of shocks realizing in period $T$, i.e. decomposes $Y(T|T) - Y(T|T-1)$. Put differently, it conducts a 1-period ahead shock decomposition from $T-1$ to $T$, by decomposing the update step of the Kalman filter. If `vintage>0` and smaller than `nobs`, the decomposition is conducted of the forecast revision $Y(T+k|T+k) - Y(T+k|T)$.

Like *shock_decomposition* it decomposes the historical deviations of the endogenous variables from their respective steady-state values into the contribution coming from the various shocks. The `variable_names` provided govern for which variables the decomposition is plotted.

Note that this command must come after either *estimation* (in case of an estimated model) or *stoch_simul* (in case of a calibrated model).

*Options*

**parameter_set = OPTION**

See *parameter_set* for possible values.

**datafile = FILENAME**

See *datafile*.

**first_obs = INTEGER**

See *first_obs*.

**nobs = INTEGER**

See *nobs*.

**use_shock_groups [= NAME]**

See *use_shock_groups*.

**colormap = VARIABLE_NAME**

See *colormap*.

**nograph**

See *nograph*. Only shock decompositions are computed and stored in `oo_.realtime_shock_decomposition`, `oo_.conditional_shock_decomposition` and `oo_.realtime_forecast_shock_decomposition` but no plot is made (See *plot_shock_decomposition*).

**presample = INTEGER**

Data point above which recursive realtime shock decompositions are computed, *i.e.* for $T = [$`presample+1`$\ldots$`nobs`$]$.

**forecast = INTEGER**

Compute shock decompositions up to $T + k$ periods, i.e. get shock contributions to k-step ahead forecasts.

**save_realtime = INTEGER_VECTOR**

Choose for which vintages to save the full realtime shock decomposition. Default: `0`.

**fast_realtime = INTEGER**

**fast_realtime = [INTEGER1:INTEGER2]**

**fast_realtime = [INTEGER1 INTEGER2 ...]**

Runs the smoother only for the data vintages provided by the specified integer (vector).

**with_epilogue**

See *with_epilogue*.

**kalman_algo = INTEGER**

See *kalman_algo = INTEGER*.

**kalman_tol = DOUBLE**

See *kalman_tol = DOUBLE*.

*Output*

**MATLAB/Octave variable: oo_.realtime_shock_decomposition**

Structure storing the results of realtime historical decompositions. Fields are three-dimensional arrays with the first two dimension equal to the ones of *oo_.shock_decomposition*. The third dimension stores the time periods and is therefore of size `T+forecast`. Fields are of the form:

```
oo_.realtime_shock_decomposition.OBJECT
```

where OBJECT is one of the following:

`pool`

Stores the pooled decomposition, i.e. for every real-time shock decomposition terminal period $T = [\texttt{presample}, \ldots, \texttt{nobs}]$ it collects the last period's decomposition $Y(T|T)$ (see also *plot_shock_decomposition*). The third dimension of the array will have size `nobs+forecast`.

`time_*`

Stores the vintages of realtime historical shock decompositions if `save_realtime` is used. For example, if `save_realtime=[5]` and `forecast=8`, the third dimension will be of size 13.

**MATLAB/Octave variable: oo_.realtime_conditional_shock_decomposition**

Structure storing the results of real-time conditional decompositions. Fields are of the form:

```
oo_.realtime_conditional_shock_decomposition.OBJECT
```

where OBJECT is one of the following:

`pool`

Stores the pooled real-time conditional shock decomposition, i.e. collects the decompositions of $Y(T|T) - Y(T|T-1)$ for the terminal periods $T = [\texttt{presample}, \ldots, \texttt{nobs}]$. The third dimension is of size `nobs`.

`time_*`

Store the vintages of $k$-step conditional forecast shock decompositions $Y(t|T+k)$, for $t = [T \ldots T+k]$. See *vintage*. The third dimension is of size `1+forecast`.

**MATLAB/Octave variable: oo_.realtime_forecast_shock_decomposition**

Structure storing the results of realtime forecast decompositions. Fields are of the form:

```
oo_.realtime_forecast_shock_decomposition.OBJECT
```

where `OBJECT` is one of the following:

`pool`

Stores the pooled real-time forecast decomposition of the 1-step ahead effect of shocks on the 1-step ahead prediction, i.e. $Y(T|T-1)$.

`time_*`

Stores the vintages of $k$-step out-of-sample forecast shock decompositions, i.e. $Y(t|T)$, for $t = [T \dots T+k]$. See *vintage*.

**Command:** **plot_shock_decomposition** [VARIABLE_NAME]...;

**Command:** **plot_shock_decomposition**(OPTIONS...) [VARIABLE_NAME]...;

This command plots the historical shock decomposition already computed by `shock_decomposition` or `realtime_shock_decomposition`. For that reason, it must come after one of these commands. The `variable_names` provided govern which variables the decomposition is plotted for.

Further note that, unlike the majority of Dynare commands, the options specified below are overwritten with their defaults before every call to `plot_shock_decomposition`. Hence, if you want to reuse an option in a subsequent call to `plot_shock_decomposition`, you must pass it to the command again.

*Options*

**use_shock_groups [= NAME]**

See *use_shock_groups*.

**colormap = VARIABLE_NAME**

See *colormap*.

**nodisplay**

See *nodisplay*.

**nograph**

See *nograph*.

**graph_format = FORMAT**

**graph_format = ( FORMAT, FORMAT... )**

See *graph_format*.

**detail_plot**

Plots shock contributions using subplots, one per shock (or group of shocks). Default: not activated

**interactive**

Under MATLAB, add uimenus for detailed group plots. Default: not activated

**screen_shocks**

For large models (i.e. for models with more than 16 shocks), plots only the shocks that have the largest historical contribution for chosen selected `variable_names`. Historical contribution is ranked by the mean absolute value of all historical contributions.

**steadystate**

If passed, the the $y$-axis value of the zero line in the shock decomposition plot is translated to the steady-state level. Default: not activated

**type = qoq | yoy | aoa**

For quarterly data, valid arguments are: `qoq` for quarter-on-quarter plots, `yoy` for year-on-year plots of growth rates, `aoa` for annualized variables, i.e. the value in the last quarter for each year is plotted. Default value: empty, i.e. standard period-on-period plots (`qoq` for quarterly data).

**fig_name = STRING**

Specifies a user-defined keyword to be appended to the default figure name set by `plot_shock_decomposition`. This can avoid to overwrite plots in case of sequential calls to `plot_shock_decomposition`.

**write_xls**

Saves shock decompositions to Excel file in the main directory, named `FILENAME_shock_decomposition_TYPE_FIG_NAME.xls`. This option requires your system to be configured to be able to write Excel files.[8]

[8] In case of Excel not being installed, https://mathworks.com/matlabcentral/fileexchange/38591-xlwrite–generate-xls-x–files-without-excel-on-mac-linux-win may be helpful.

**realtime = INTEGER**

Which kind of shock decomposition to plot. INTEGER can take the following values:

- 0: standard historical shock decomposition. See *shock_decomposition*.
- 1: realtime historical shock decomposition. See *realtime_shock_decomposition*.
- 2: conditional realtime shock decomposition. See *realtime_shock_decomposition*.
- 3: realtime forecast shock decomposition. See *realtime_shock_decomposition*.

If no vintage is requested, i.e. vintage=0 then the pooled objects from *realtime_shock_decomposition* will be plotted and the respective vintage otherwise. Default: 0.

**vintage = INTEGER**

Selects a particular data vintage in $[presample, \ldots, nobs]$ for which to plot the results from *realtime_shock_decomposition* selected via the *realtime* option. If the standard historical shock decomposition is selected (realtime=0), vintage will have no effect. If vintage=0 the pooled objects from *realtime_shock_decomposition* will be plotted. If vintage>0, it plots the shock decompositions for vintage $T =$ vintage under the following scenarios:

- realtime=1: the full vintage shock decomposition $Y(t|T)$ for $t = [1, \ldots, T]$
- realtime=2: the conditional forecast shock decomposition from $T$, i.e. plots $Y(T+j|T+j)$ and the shock contributions needed to get to the data $Y(T+j)$ conditional on $T =$ vintage, with $j = [0, \ldots,$ forecast$]$.
- realtime=3: plots unconditional forecast shock decomposition from $T$, i.e. $Y(T+j|T)$, where $T =$ vintage and $j = [0, \ldots,$ forecast$]$.

Default: 0.

**plot_init_date = DATE**

If passed, plots decomposition using plot_init_date as initial period. Default: first observation in estimation

**plot_end_date = DATE**

If passed, plots decomposition using plot_end_date as last period. Default: last observation in estimation

**diff**

If passed, plot the decomposition of the first difference of the list of variables. If used in combination with *flip*, the diff operator is first applied. Default: not activated

**flip**

If passed, plot the decomposition of the opposite of the list of variables. If used in combination with *diff*, the diff operator is first applied. Default: not activated

**max_nrows**

Maximum number of rows in the subplot layout of detailed shock decomposition graphs. Note that columns are always 3. Default: 6

**with_epilogue**

See *with_epilogue*.

**init2shocks**

**init2shocks = NAME**

Use the information contained in an *init2shocks* block, in order to attribute initial conditions to shocks. The name of the block can be explicitly given, otherwise it defaults to the default block.

**Block; init2shocks ;**

**Block:** **init2shocks**(OPTIONS...);

This blocks gives the possibility of attributing the initial condition of endogenous variables to the contribution of exogenous variables in the shock decomposition.

For example, in an AR(1) process, the contribution of the initial condition on the process variable can naturally be assigned to the innovation of the process.

Each line of the block should have the syntax:

```
VARIABLE_1 [,] VARIABLE_2;
```

Where VARIABLE_1 is an endogenous variable whose initial condition will be attributed to the exogenous VARIABLE_2.

The information contained in this block is used by the *plot_shock_decomposition* command when given the `init2shocks` option.

*Options*

**name = NAME**

Specifies a name for the block, that can be referenced from `plot_shock_decomposition`, so that several such blocks can coexist in a single model file. If the name is unspecified, it defaults to `default`.

*Example*

```
var y y_s R pie dq pie_s de A y_obs pie_obs R_obs;
varexo e_R e_q e_ys e_pies e_A;
...

model;
  dq = rho_q*dq(-1)+e_q;
  A = rho_A*A(-1)+e_A;
  ...
end;

...

init2shocks;
  dq e_q;
  A e_A;
end;

shock_decomposition(nograph);

plot_shock_decomposition(init2shocks) y_obs R_obs pie_obs dq de;
```

In this example, the initial conditions of `dq` and `A` will be respectively attributed to `e_q` and `e_A`.

**Command:** **initial_condition_decomposition** [VARIABLE_NAME]...;

**Command:** **initial_condition_decomposition**(OPTIONS...) [VARIABLE_NAME]...;

This command computes and plots the decomposition of the effect of smoothed initial conditions of state variables. The `variable_names` provided govern which variables the decomposition is plotted for.

Further note that, unlike the majority of Dynare commands, the options specified below are overwritten with their defaults before every call to `initial_condition_decomposition`. Hence, if you want to reuse an option in a subsequent call to `initial_condition_decomposition`, you must pass it to the command again.

*Options*

**colormap = VARIABLE_NAME**

See *colormap*.

**nodisplay**

See *nodisplay*.

**graph_format = FORMAT**

**graph_format = ( FORMAT, FORMAT... )**

See *graph_format*.

**detail_plot**

Plots shock contributions using subplots, one per shock (or group of shocks). Default: not activated

**steadystate**

If passed, the the $y$-axis value of the zero line in the shock decomposition plot is translated to the steady-state level. Default: not activated

**type = qoq | yoy | aoa**

For quarterly data, valid arguments are: `qoq` for quarter-on-quarter plots, `yoy` for year-on-year plots of growth rates, `aoa` for annualized variables, i.e. the value in the last quarter for each year is plotted. Default value: empty, i.e. standard period-on-period plots (`qoq` for quarterly data).

**fig_name = STRING**

Specifies a user-defined keyword to be appended to the default figure name set by `plot_shock_decomposition`. This can avoid to overwrite plots in case of sequential calls to `plot_shock_decomposition`.

**write_xls**

Saves shock decompositions to Excel file in the main directory, named `FILENAME_shock_decomposition_TYPE_FIG_NAME_initval.xls`. This option requires your system to be configured to be able to write Excel files.[Page 173, 8]

**plot_init_date = DATE**

If passed, plots decomposition using `plot_init_date` as initial period. Default: first observation in estimation

**plot_end_date = DATE**

If passed, plots decomposition using `plot_end_date` as last period. Default: last observation in estimation

**diff**

If passed, plot the decomposition of the first difference of the list of variables. If used in combination with *flip*, the `diff` operator is first applied. Default: not activated

**flip**

If passed, plot the decomposition of the opposite of the list of variables. If used in combination with *diff*, the `diff` operator is first applied. Default: not activated

**Command: squeeze_shock_decomposition** [VARIABLE_NAME]...;

For large models, the size of the information stored by shock decompositions (especially various settings of realtime decompositions) may become huge. This command allows to squeeze this information in two possible ways:

- Automatic (default): only the variables for which plotting has been explicitly required with `plot_shock_decomposition` will have their decomposition left in `oo_` after this command is run;
- If a list of variables is passed to the command, then only those variables will have their decomposition left in `oo_` after this command is run.

## 4.20 Calibrated Smoother

Dynare can also run the smoother on a calibrated model:

**Command: calib_smoother** [VARIABLE_NAME]...;

**Command:** **calib_smoother**(OPTIONS...) [VARIABLE_NAME]...;

This command computes the smoothed variables (and possible the filtered variables) on a calibrated model.

A datafile must be provided, and the observable variables declared with `varobs`. The smoother is based on a first-order approximation of the model.

By default, the command computes the smoothed variables and shocks and stores the results in `oo_.SmoothedVariables` and `oo_.SmoothedShocks`. It also fills `oo_.UpdatedVariables`.

The initial condition of the Kalman filter can be set using the *filter_initial_state* block.

*Options*

**datafile = FILENAME**

See *datafile*.

**dataseries = DSERIES_NAME**

See *dataseries*.

**filtered_vars**

**filtered_vars = BOOLEAN**

Triggers the computation of filtered variables. See *filtered_vars*, for more details.

**filter_step_ahead = [INTEGER1:INTEGER2]**

See *filter_step_ahead*.

**prefilter = INTEGER**

See *prefilter*.

**parameter_set = OPTION**

See *parameter_set* for possible values. Default: `calibration`.

**loglinear**

See *loglinear*.

**first_obs = INTEGER | DATE**

See *first_obs*.

**last_obs = INTEGER | DATE**

See *last_obs*.

**filter_decomposition**

**filter_decomposition = BOOLEAN**

See *filter_decomposition*.

**filter_covariance**

**filter_covariance = BOOLEAN**

See *filter_covariance*.

**smoother_redux**

**smoother_redux = BOOLEAN**

See *smoother_redux*.

**kalman_algo = INTEGER**

See *kalman_algo*.

**diffuse_filter = INTEGER**

See *diffuse_filter*.

**diffuse_kalman_tol = DOUBLE**

See *diffuse_kalman_tol*.

**use_univariate_smoother_if_singularity_is_detected**

**use_univariate_smoother_if_singularity_is_detected = BOOLEAN**

See *use_univariate_smoother_if_singularity_is_detected*.

**xls_sheet = QUOTED_STRING**

See *xls_sheet*.

**xls_range = RANGE**

See *xls_range*.

**heteroskedastic_filter**

**heteroskedastic_filter = BOOLEAN**

See *heteroskedastic_filter*.

**nobs = INTEGER**

**nobs = [INTEGER1:INTEGER2]**

See *nobs*.

## 4.21 Forecasting

On a calibrated model, forecasting is done using the *forecast* command. On an estimated model, use the *forecast* option of the *estimation* command.

It is also possible to compute forecasts on a calibrated or estimated model for a given constrained path of the future endogenous variables. This is done, from the reduced-form representation of the DSGE model, by finding the structural shocks that are needed to match the restricted paths. Use *conditional_forecast*, *conditional_forecast_paths* and *plot_conditional_forecast* for that purpose.

Finally, it is possible to do forecasting with a Bayesian VAR using the *bvar_forecast* command.

**Command:** **forecast** [VARIABLE_NAME...];

**Command:** **forecast**(OPTIONS...) [VARIABLE_NAME...];

This command computes a simulation of a stochastic model from an arbitrary initial point.

When the model also contains deterministic exogenous shocks, the simulation is computed conditionally to the agents knowing the future values of the deterministic exogenous variables.

`forecast` must be called after *stoch_simul*.

`forecast` plots the trajectory of endogenous variables. When a list of variable names follows the command, only those variables are plotted. A 90% confidence interval is plotted around the mean trajectory. Use option `conf_sig` to change the level of the confidence interval.

*Options*

**periods = INTEGER**

Number of periods of the forecast. Default: `5`.

**conf_sig = DOUBLE**

Level of significance for confidence interval. Default: `0.90`.

**nograph**

See *nograph*.

**nodisplay**

See *nodisplay*.

**graph_format = FORMAT**

**graph_format = ( FORMAT, FORMAT... )**

See *graph_format = FORMAT*.

*Initial Values*

`forecast` computes the forecast taking as initial values the values specified in `histval` (see *histval*). When no `histval` block is present, the initial values are the one stated in `initval`. When `initval` is followed by command *steady*, the initial values are the steady state (see *steady*).

*Output*

The results are stored in `oo_.forecast`, which is described below.

*Example*

```
varexo_det tau;

varexo e;
...
shocks;
var e; stderr 0.01;
var tau;
periods 1:9;
values -0.15;
end;

stoch_simul(irf=0);

forecast;
```

**MATLAB/Octave variable: oo_.forecast**

Variable set by the *forecast* command, or by the *estimation* command if used with the *forecast* option and ML or if no Metropolis-Hastings has been computed (in that case, the forecast is computed for the posterior mode). Fields are of the form:

```
oo_.forecast.FORECAST_MOMENT.VARIABLE_NAME
```

where `FORECAST_MOMENT` is one of the following:

`HPDinf`

Lower bound of a 90% HPD interval[9] of forecast due to parameter uncertainty, but ignoring the effect of measurement error on observed variables. In case of ML, it stores the lower bound of the confidence interval.

`HPDsup`

Upper bound of a 90% HPD forecast interval due to parameter uncertainty, but ignoring the effect of measurement error on observed variables. In case of ML, it stores the upper bound of the confidence interval.

`HPDinf_ME`

Lower bound of a 90% HPD interval[10] of forecast for observed variables due to parameter uncertainty and measurement error. In case of ML, it stores the lower bound of the confidence interval.

`HPDsup_ME`

Upper bound of a 90% HPD interval of forecast for observed variables due to parameter uncertainty and measurement error. In case of ML, it stores the upper bound of the confidence interval.

`Mean`

Mean of the posterior distribution of forecasts.

[9] See option *conf_sig* to change the size of the HPD interval.
[10] See option *conf_sig* to change the size of the HPD interval.

**MATLAB/Octave variable: oo_.PointForecast**

Set by the *estimation* command, if it is used with the *forecast* option and if either *mh_replic* > 0 or the *load_mh_file* option are used.

Contains the distribution of forecasts taking into account the uncertainty about both parameters and shocks.

Fields are of the form:

```
oo_.PointForecast.MOMENT_NAME.VARIABLE_NAME
```

**MATLAB/Octave variable: oo_.MeanForecast**

Set by the *estimation* command, if it is used with the *forecast* option and if either *mh_replic* > 0 or *load_mh_file* option are used.

Contains the distribution of forecasts where the uncertainty about shocks is averaged out. The distribution of forecasts therefore only represents the uncertainty about parameters.

Fields are of the form:

```
oo_.MeanForecast.MOMENT_NAME.VARIABLE_NAME
```

**Command: conditional_forecast(OPTIONS...);**

This command computes forecasts on an estimated or calibrated model for a given constrained path of some future endogenous variables. This is done using the reduced-form first-order state-space representation of the DSGE model by finding the structural shocks that are needed to match the restricted paths. Consider the augmented state-space representation that stacks both predetermined and non-predetermined variables into a vector $y_t$:

$$y_t = Ty_{t-1} + R\varepsilon_t$$

Both $y_t$ and $\varepsilon_t$ are split up into controlled and uncontrolled ones, and we assume without loss of generality that the constrained endogenous variables and the controlled shocks come first :

$$\begin{pmatrix} y_{c,t} \\ y_{u,t} \end{pmatrix} = \begin{pmatrix} T_{c,c} & T_{c,u} \\ T_{u,c} & T_{u,u} \end{pmatrix} \begin{pmatrix} y_{c,t-1} \\ y_{u,t-1} \end{pmatrix} + \begin{pmatrix} R_{c,c} & R_{c,u} \\ R_{u,c} & R_{u,u} \end{pmatrix} \begin{pmatrix} \varepsilon_{c,t} \\ \varepsilon_{u,t} \end{pmatrix}$$

where matrices $T$ and $R$ are partitioned consistently with the vectors of endogenous variables and innovations. Provided that matrix $R_{c,c}$ is square and full rank (a necessary condition is that the number of free endogenous variables matches the number of free innovations), given $y_{c,t}$, $\varepsilon_{u,t}$ and $y_{t-1}$ the first block of equations can be solved for $\varepsilon_{c,t}$:

$$\varepsilon_{c,t} = R_{c,c}^{-1}\left(y_{c,t} - T_{c,c}y_{c,t-1} - T_{c,u}y_{u,t-1} - R_{c,u}\varepsilon_{u,t}\right)$$

and $y_{u,t}$ can be updated by evaluating the second block of equations:

$$y_{u,t} = T_{u,c}y_{c,t-1} + T_{u,u}y_{u,t-1} + R_{u,c}\varepsilon_{c,t} + R_{u,u}\varepsilon_{u,t}$$

By iterating over these two blocks of equations, we can build a forecast for all the endogenous variables in the system conditional on paths for a subset of the endogenous variables. If the distribution of the free innovations $\varepsilon_{u,t}$ is provided (*i.e.* some of them have positive variances) this exercise is replicated (the number of replication is controlled by the option *replic* described below) by drawing different sequences of free

innovations. The result is a predictive distribution for the uncontrolled endogenous variables, $y_{u,t}$, that Dynare will use to report confidence bands around the point conditional forecast.

A few things need to be noted. First, by making use of the first-order state-space solution, even if a higher-order approximation was performed, the conditional forecasts will be based on a first-order approximation. Second, since the controlled exogenous variables are identified on the basis of the reduced-form model (*i.e.* after solving for the expectations), they are unforeseen shocks from the perspective of the agents in the model. That is, agents expect the endogenous variables to return to their respective steady-state levels but are surprised in each period by the realisation of shocks keeping the endogenous variables along a predefined (unexpected) path. Third, if the structural innovations are correlated, because the calibrated or estimated covariance matrix has non-zero off-diagonal elements, the results of the conditional forecasts will depend on the ordering of the innovations (as declared after `varexo`). As in VAR models, a Cholesky decomposition is used to factorise the covariance matrix and identify orthogonal impulses. It is preferable to declare the correlations in the *model* block (explicitly imposing the identification restrictions), unless you are satisfied with the implicit identification restrictions implied by the Cholesky decomposition.

This command has to be called after *estimation* or *stoch_simul*.

Use *conditional_forecast_paths* block to give the list of constrained endogenous variables, their constrained future path, and optionally the exogenous variables used as instruments. If using the new syntax (with `exogenize` and `endogenize` keywords), the `controlled_varexo` option must not be passed. If using the old deprecated syntax (with `var` keyword), you must use the `controlled_varexo` option to specify the structural shocks which will be matched to generate the constrained path.

Use *plot_conditional_forecast* to graph the results.

The starting point of the conditional forecast depends on the value of `parameter_set`. If `parameter_set=calibration`, the starting point is the contents of the *histval* block if present, or the steady state otherwise (in particular, this means that a for a calibrated model, it is possible to use a smoothed state as the starting point of the conditional forecast, by running *smoother2histval* after *calib_smoother*). For any other value of `parameter_set`, a smoother is run with the corresponding parameter values and the last smoothed point is used as the starting point of the conditional forecast; it is the responsibility of the user to make sure that the specified parameter set has already been computed in cases where it refers to estimation.

*Options*

**`parameter_set = OPTION`**

See *parameter_set* for possible values. No default value, mandatory option.

**`controlled_varexo = (VARIABLE_NAME...)`**

Deprecated. Specify the exogenous variables to use as control variables. Only required when using the old syntax (with `var` keyword) in the *conditional_forecast_paths* block. Not allowed when using the new syntax (with `exogenize` and `endogenize` keywords).

**`periods = INTEGER`**

Number of periods of the forecast. Default: `40`. `periods` cannot be smaller than the number of constrained periods.

**`replic = INTEGER`**

Number of simulations used to compute the conditional forecast uncertainty. Default: `5000`. If set to 0, only the point forecast will be computed (stored in the `Point` field of the results structure).

**`conf_sig = DOUBLE`**

Level of significance for confidence interval. Default: `0.80`.

**`check_tol = DOUBLE`**

Tolerance for checking that conditioning succeeded (i.e. that the targets for controlled endogenous are reached with the computed controlled exogenous). Default: `1e-6`.

**`datafile = FILENAME`**

Specifies a CSV file containing the conditional forecast constraints. The CSV file should have 4 columns:

- Column 1: period index (numeric, starting from 1) or date (e.g. `2025Q1`)
- Column 2: name of endogenous variable to exogenize
- Column 3: value imposed on the endogenous variable
- Column 4: name of exogenous variable to endogenize

Each line describes a constraint for a single period. Empty lines are ignored. This option cannot be used together with a *conditional_forecast_paths* block. If dates are used in the first column, the initial date of the forecast must be determinable (typically inferred from the smoother output or the histval information). The CSV file can optionally have a first row with column headers (the content of these headers is not interpreted or checked).

*Output*

The results are stored in `oo_.conditional_forecast`, which is described below.

*Example*

```
var y a;
varexo e u;
...
estimation(...);

conditional_forecast_paths;
  exogenize y;
  periods 1:3, 4:5;
  values 2, 5;
  endogenize e;

  exogenize a;
  periods 1:5;
  values 3;
  endogenize u;
end;

conditional_forecast(parameter_set = calibration, replic = 3000);

plot_conditional_forecast(periods = 10) a y;
```

*Example (with datafile option, numeric periods)*

```
var y;
varexo e;
...
estimation(...);

% Create a CSV file with constraints (4 columns: period, endo_
↪name, value, exo_name)
% using numeric period indices. For example, to exogenize y with␣
↪value 2
% and endogenize e in periods 1-3, the CSV file would contain:
% 1,y,2,e
% 2,y,2,e
% 3,y,2,e

conditional_forecast(parameter_set = calibration, datafile = 'my_
↪constraints.csv');
```

*Example (with datafile option, dates)*

```
var a;
varexo u;
...
estimation(...);

% Create a CSV file with constraints (4 columns: period, endo_
↪name, value, exo_name)
% using dates. For example, to exogenize a with value 3 and␣
↪endogenize u
% in periods 2025Q1-2025Q2, the CSV file would contain:
% 2025Q1,a,3,u
% 2025Q2,a,3,u

conditional_forecast(parameter_set = calibration, datafile = 'my_
↪constraints.csv');
```

**MATLAB/Octave variable: oo_.conditional_forecast.cond**

Variable set by the `conditional_forecast` command. It stores the conditional forecasts, for both endogenous and exogenous variables. Fields for point and mean forecasts are `periods`+1 by 1 vectors storing the initial state (time 0) and the subsequent `periods` forecasts periods; for the confidence interval, fields are 2 by `periods`+1 matrices, with lower (resp. upper) band in the first (resp. second) row. Fields are of the form:

```
oo_.conditional_forecast.cond.FORECAST_MOMENT.VARIABLE_NAME
```

where FORECAST_MOMENT is one of the following:

`Point`

Point forecast of the conditional path. This field is always present.

`Mean`

Mean of the conditional forecast distribution. Only present if `replic > 0`.

`ci`

Confidence interval of the conditional forecast distribution. The size corresponds to `conf_sig`. Only present if `replic > 0`.

**MATLAB/Octave variable: oo_.conditional_forecast.uncond**

Variable set by the `conditional_forecast` command. It stores the unconditional forecasts, for both endogenous and exogenous variables. Fields are of the form:

```
oo_.conditional_forecast.uncond.FORECAST_MOMENT.VARIABLE_NAME
```

where FORECAST_MOMENT is one of the following:

`Point`

Point forecast of the unconditional path. This field is always present.

`Mean`

Mean of the unconditional forecast distribution. Only present if `replic > 0`.

`ci`

Confidence interval of the unconditional forecast distribution. Only present if `replic > 0`.

**MATLAB/Octave variable: oo_.conditional_forecast.initial_date**

A date object (see *Dates in a mod file*) corresponding to the date of the initial condition in the conditional

forecast. This object may be empty if no date information could be determined (the initial date is typically inferred from the smoother output or the histval information).

**MATLAB/Octave command:** `get_conditional_forecast('TYPE', 'MOMENT', 'VARIABLE_NAME' [, 'VARIABLE_NAME']...);`

Returns the conditional forecast for the given forecast type, moment and requested endogenous or exogenous variables, as they are stored in *oo_.conditional_forecast.cond* and *oo_.conditional_forecast.uncond*. *'TYPE'* can be either *'cond'* or *'uncond'*. *'MOMENT'* can be either *'Point'*, *'Mean'*, *'CI_inf'*, or *'CI_sup'* (the latter two for getting either the inferior or upper bound of the confidence interval).

**MATLAB/Octave command:** `get_conditional_forecast_dseries('TYPE', 'MOMENT');`

**MATLAB/Octave command:** `get_conditional_forecast_dseries('TYPE', 'MOMENT', 'VARIABLE_NAME' [, 'VARIABLE_NAME']...);`

Similar to *get_conditional_forecast*, but returns the conditional forecasts as a *dseries* object, using the date information stored in *oo_.conditional_forecast.initial_date*. If no variable name is given, returns a dseries object containing all endogenous and exogenous variables.

**Block:** `conditional_forecast_paths ;`

**Block:** `conditional_forecast_paths(overwrite);`

Describes the path of constrained endogenous variables and their instruments, before calling `conditional_forecast`.

The syntax uses `exogenize` and `endogenize` keywords. The `exogenize` keyword specifies which endogenous variable is being exogenized (i.e., constrained to follow a specific path), and the `endogenize` keyword specifies which exogenous variable (shock) is being used as an instrument to control that endogenous variable. This syntax allows for time-varying instruments, where different shocks can be used to control the same endogenous variable in different periods.

Each constraint is specified as follows:

```
exogenize ENDOGENOUS_NAME;
periods INTEGER[:INTEGER] | DATE[:DATE] [[,] INTEGER[:INTEGER] | DATE[:DATE]]...;
values EXPRESSION [, EXPRESSION ]...;
endogenize EXOGENOUS_NAME;
```

The `periods` line specifies the periods for which the constraint applies, and the `values` line specifies the values that the exogenized variable should take in those periods. The number of periods must match the number of values (or the value can be a scalar, which will be applied to all periods).

In case of the presence of `observation_trends`, the specified controlled path for these variables needs to include the trend component. When using the *loglinear* option, it is necessary to specify the logarithm of the controlled variables.

Multiple constraints can be declared in the same block. By default, if multiple `conditional_forecast_paths` blocks are used, they are concatenated. To replace the previous constraints instead of concatenating, use the `overwrite` option when declaring the block.

*Example*

```
var gy_obs gp_obs;
varexo e_a e_m;

...

xx = [0.05 0.04 0.04 0.02 0];

conditional_forecast_paths;
  exogenize gy_obs;
  periods 1, 2, 3:5;
```

(continues on next page)

(continued from previous page)

```
  values 0.01, -0.02, 0.03;
  endogenize e_a;

  exogenize gp_obs;
  periods 1:5, 6:10;
  values xx, xx-0.01;
  endogenize e_m;
end;

conditional_forecast(parameter_set=calibration);
```

*Example (with dates)*

```
var gy_obs;
varexo e_a;

...

// mydata.xlsx is supposed to have dates on the first column
calib_smoother(datafile = 'mydata.xlsx');

smoother2histval;

conditional_forecast_paths;
  exogenize gy_obs;
  periods 2025Q1, 2025Q2, 2025Q3:2026Q1;
  values 0.01, -0.02, 0.03;
  endogenize e_a;
end;

conditional_forecast(parameter_set=calibration);
```

**Deprecated syntax:**

For historical reasons, there also exist an alternative, deprecated syntax, that should not be used in new code, but which is maintained for backward compatibility. It is deprecated because it is less flexible when having a time-varying number of instruments: the selection of the exogenous variables which will *not* be endogenized in that case is not flexible, and moreover those variables will be set to zero for periods when they are not endogenized (this implies that there is no forecast uncertainty arising from these exogenous variables in uncontrolled periods).

The deprecated syntax uses the `var` keyword (instead of `exogenize`) and does not use `endogenize`. In this case, the instruments must be specified via the `controlled_varexo` option in the *`conditional_forecast`* command.

The deprecated syntax is similar to deterministic shocks in `shocks`, see *Shocks on exogenous variables* for the detailed syntax reference. Each constraint is specified as follows:

```
var ENDOGENOUS_NAME;
periods INTEGER[:INTEGER] [[,] INTEGER[:INTEGER]]...;
values DOUBLE | (EXPRESSION)  [[,] DOUBLE | (EXPRESSION) ]...;
```

It is possible to have a different number of controlled periods for different variables. In that case, the order of declaration of endogenous controlled variables and of exogenous in `controlled_varexo` option matters: if the second endogenous variable is controlled for fewer periods than the first one, then the second `controlled_varexo` is set to zeros during those periods.

When using the deprecated syntax, when there are multiple `conditional_forecast_paths` blocks, each new block overrides the previous one (*i.e.* they are not concatenated as when using the new syntax).

*Example (deprecated syntax)*

```
xx = [0.05 0.04 0.04 0.02 0];

conditional_forecast_paths;
  var gy_obs;
  periods 1, 2, 3:5;
  values 0.01, -0.02, 0.03;

  var gp_obs;
  periods 1:5, 6:10;
  values (xx), (xx-0.01);
end;

conditional_forecast(parameter_set=calibration, controlled_varexo=(e_a,␣
↪e_m));
```

**Command:** **plot_conditional_forecast** [VARIABLE_NAME...];

**Command:** **plot_conditional_forecast**(periods = INTEGER) [VARIABLE_NAME...];

Plots the conditional (plain lines) and unconditional (dashed lines) forecasts.

To be used after `conditional_forecast`.

*Options*

**periods = INTEGER**

Number of periods to be plotted. Default: equal to periods in `conditional_forecast`. The number of periods declared in `plot_conditional_forecast` cannot be greater than the one declared in `conditional_forecast`.

**Command:** **bvar_forecast** ;

This command computes (out-of-sample) forecasts for an estimated BVAR model, using Minnesota priors.

See `bvar-a-la-sims.pdf`, which comes with Dynare distribution, for more information on this command.

**Command:** **smoother2histval** ;

**Command:** **smoother2histval**(OPTIONS...);

The purpose of this command is to construct initial conditions (for a subsequent simulation) that are the smoothed values obtained through the *estimation* command (when used with the *smoother* option), or the *calib_smoother* command.

The `smoother2histval` command will extract the smoothed values from *oo_.SmoothedVariables* (and possibly from *oo_.SmoothedShocks* if there are lagged exogenous), and will use these values to construct initial conditions (as if they had been manually entered through *histval*).

If the command is run after a Metropolis, then the posterior mean of smoothed variables will be used.

*Options*

**period = INTEGER | DATE**

Period to use as the starting point for the subsequent simulation. If passed an integer, it should be between 1 and the number of observations that were used to produce the smoothed values. Default: the last observation.

**invars = ( VARIABLE_NAME [VARIABLE_NAME ...] )**

A list of variables to read from the smoothed values. It can contain state endogenous variables, and also exogenous variables having a lag. Default: all the state endogenous variables, and all the exogenous variables with a lag.

**outfile = FILENAME**

Write the initial conditions to a file. Default: write the initial conditions in the current workspace, so that a simulation can be performed.

**outvars = ( VARIABLE_NAME [VARIABLE_NAME ...] )**

A list of variables which will be given the initial conditions. This list must have the same length than the list given to `invars`, and there will be a one-to-one mapping between the two list. Default: same value as option `invars`.

*Use cases*

There are two possible ways of using this command:

- Everything in a single file: run an estimation with a smoother (or comm:*calib_smoother*), then run `smoother2histval` (without the `outfile` option), then run a stochastic simulation, *forecast*, or *conditional_forecast*.
- In two files: in the first file, run the smoother and then run `smoother2histval` with the `outfile` option; in the second file, run `histval_file` to load the initial conditions, and run a (deterministic or stochastic) simulation, *forecast*, or *conditional_forecast*.

## 4.22 Optimal policy

Dynare has tools to compute optimal policies for various types of objectives. You can either solve for optimal policy under commitment with *ramsey_model*, for optimal policy under discretion with *discretionary_policy* or for optimal simple rules with `osr` (also implying commitment).

**Command: planner_objective** MODEL_EXPRESSION ;

This command declares the policy maker objective, for use with *ramsey_model* or *discretionary_policy*.

You need to give the one-period objective, not the discounted lifetime objective. The discount factor is given by the *planner_discount* option of *ramsey_model* and *discretionary_policy*. The objective function can only contain current endogenous variables and no exogenous ones. This limitation is easily circumvented by defining an appropriate auxiliary variable in the model.

With *ramsey_model*, you are not limited to quadratic objectives: you can give any arbitrary nonlinear expression.

With *discretionary_policy*, the objective function must be quadratic.

**Command: evaluate_planner_objective ;**

**Command: evaluate_planner_objective**(OPTIONS...);

This command computes, displays, and stores the value of the planner objective function under Ramsey policy or discretion in `oo_.planner_objective_value`. It will provide both unconditional welfare and welfare conditional on the initial (i.e. period 0) values of the endogenous and exogenous state variables inherited by the planner. In a deterministic context, the respective initial values are set using `initval` or `histval` (depending on the exact context).

In a stochastic context, if no initial-state values have been specified with `histval`, their values are taken to be the steady-state values. Because conditional welfare is computed conditional on optimal policy by the planner in the first endogenous period (period 1), it is conditional on the information set in the period 1. This information set includes both the predetermined states inherited from period 0 (specified via `histval` for both endogenous and lagged exogenous states) as well as the period 1 values of the exogenous shocks. The latter are specified using the perfect foresight syntax of the `shocks` block.

At the current stage, the stochastic context does not support the `pruning` option. At `order>3`, only the computation of conditional welfare with steady-state Lagrange multipliers is supported. Note that at `order=2`, the output is based on the second-order accurate approximation of the variance stored in `oo_.var`.

*Options*

**periods = INTEGER**

The value of the option specifies the number of periods to use in the simulations in the computation of unconditional welfare at higher order.

Default: `10000`.

**drop = INTEGER**

The number of burn-in draws out of `periods` discarded before computing the unconditional welfare at higher order. Default: `1000`.

*Example (stochastic context)*

```
var a ...;
varexo u;

model;
a = rho*a(-1)+u+u(-1);
...
end;

histval;
u(0)=1;
a(0)=-1;
end;

shocks;
var u; stderr 0.008;
var u;
periods 1;
values 1;
end;

evaluate_planner_objective;
```

**MATLAB/Octave variable: oo_.planner_objective_value.unconditional**

Scalar storing the value of unconditional welfare. In a perfect foresight context, it corresponds to welfare in the long-run, approximated as welfare in the terminal simulation period.

**MATLAB/Octave variable: oo_.planner_objective_value.conditional**

In a non-Ramsey context, this field will be a scalar storing the value of welfare conditional on the specified initial condition. In a perfect foresight Ramsey context, this field will be a scalar storing the value of welfare conditional on the specified initial condition and zero initial Lagrange multipliers.

In a stochastic Ramsey context, it will have two subfields:

**MATLAB/Octave variable:**
**oo_.planner_objective_value.conditional.steady_initial_multiplier**

Stores the value of the planner objective when the initial Lagrange multipliers associated with the planner's problem are set to their steady-state values (see *ramsey_policy*).

**MATLAB/Octave variable:**
**oo_.planner_objective_value.conditional.zero_initial_multiplier**

Stores the value of the planner objective when the initial Lagrange multipliers associated with the planner's problem are set to 0, i.e. it is assumed that the planner exploits its ability to surprise private agents in the first period of implementing Ramsey policy. This value corresponds to the planner implementing optimal policy for the first time and committing not to re-optimize in the future.

### 4.22.1 Optimal policy under commitment (Ramsey)

Dynare allows to automatically compute optimal policy choices of a Ramsey planner who takes the specified private sector equilibrium conditions into account and commits to future policy choices. Doing so requires specifying the private sector equilibrium conditions in the `model` block and a `planner_objective` as well as potentially some `instruments` to facilitate computations.

**Warning**

Be careful when employing forward-looking auxiliary variables in the context of timeless perspective Ramsey computations. They may alter the problem the Ramsey planner will solve for the first period, although they seemingly leave the private sector equilibrium unaffected. The reason is the planner optimizes with respect to variables dated `t` and takes the value of time 0 variables as given, because they are predetermined. This set of initially predetermined variables will change with forward-looking definitions. Thus, users are strongly advised to use model-local variables instead.

*Example*

Consider a perfect foresight example where the Euler equation for the return to capital is given by

```
1/C=beta*1/C(+1)*(R(+1)+(1-delta))
```

The job of the Ramsey planner in period 1 is to choose $C_1$ and $R_1$, taking as given $C_0$. The above equation may seemingly equivalently be written as

```
1/C=beta*1/C(+1)*(R_cap);
R_cap=R(+1)+(1-delta);
```

due to perfect foresight. However, this changes the problem of the Ramsey planner in the first period to choosing $C_1$ and $R_1$, taking as given both $C_0$ and $R_0^{cap}$. Thus, the relevant return to capital in the Euler equation of the first period is not a choice of the planner anymore due to the forward-looking nature of the definition in the second line!

A correct specification would be to instead define `R_cap` as a model-local variable:

```
1/C=beta*1/C(+1)*(R_cap);
#R_cap=R(+1)+(1-delta);
```

**Command:** **`ramsey_model`**`(OPTIONS...);`

This command computes the first-order conditions for maximizing the policy maker objective function subject to the constraints provided by the equilibrium path of the private economy.

The planner objective must be declared with the *`planner_objective`* command.

This command only creates the expanded model, it doesn't perform any computations. It needs to be followed by other instructions to actually perform desired computations. Examples are calls to *`steady`* to compute the steady state of the Ramsey economy, to *`stoch_simul`* with various approximation orders to conduct stochastic simulations based on perturbation solutions, to *`estimation`* in order to estimate models under optimal policy with commitment, and to perfect foresight simulation routines.

See *Auxiliary variables*, for an explanation of how Lagrange multipliers are automatically created.

*Options*

This command accepts the following options:

**`planner_discount = EXPRESSION`**

Mandatory option declaring the discount factor of the central planner `optimal_policy_discount_factor`. The discount factor can be a numeric value or a mathematical expression involving constants and declared parameters. If it contains parameters that are estimated, the parameter dependency will be taken into account. Note that it is not possible to change the structure of the planner discount factor within the `<mod_file>.mod` file, e.g., it is not possible to have a discount factor that is equal to a declared parameter `beta` in the first part of the file and on a declared parameter `gamma` in the second part of the file.

**`instruments = (VARIABLE_NAME,...)`**

Declares instrument variables for the computation of the steady state under optimal policy. Requires a *`steady_state_model`* block or a `_steadystate.m` file. See below.

*Steady state*

Dynare takes advantage of the fact that the Lagrange multipliers appear linearly in the equations of the steady state of the model under optimal policy. Nevertheless, it is in general very difficult to compute the steady state with simply a numerical guess in `initval` for the endogenous variables.

It greatly facilitates the computation, if the user provides an analytical solution for the steady state (in *steady_state_model* block or in a `_steadystate.m` file). In this case, it is necessary to provide a steady-state solution CONDITIONAL on the value of the instruments in the optimal policy problem and declared with the option `instruments`. The initial value of the instrument for steady state finding in this case is set with `initval`. Note that computing and displaying steady-state values using the *steady* command or calls to *resid* must come after the *ramsey_model* statement and the `initval` block.

Note that choosing the instruments is partly a matter of interpretation and you can choose instruments that are handy from a mathematical point of view but different from the instruments you would refer to in the analysis of the paper. A typical example is choosing inflation or nominal interest rate as an instrument.

**Block: ramsey_constraints ;**

This block lets you define constraints on the variables in the Ramsey problem. The constraints take the form of a variable, an inequality operator (> or <) and a constant.

*Example*

```
ramsey_constraints;
i > 0;
end;
```

**Command: ramsey_policy** [VARIABLE_NAME...];

**Command: ramsey_policy**(OPTIONS...) [VARIABLE_NAME...];

This command is deprecated and formally equivalent to the calling sequence

```
ramsey_model;
stoch_simul;
evaluate_planner_objective;
```

It computes an approximation of the policy that maximizes the policy maker's objective function subject to the constraints provided by the equilibrium path of the private economy and under commitment to this optimal policy. The Ramsey policy is computed by approximating the equilibrium system around the perturbation point where the Lagrange multipliers are at their steady state, i.e. where the Ramsey planner acts as if the initial multipliers had been set to 0 in the distant past, giving them time to converge to their steady-state value. Consequently, the optimal decision rules are computed around this steady state of the endogenous variables and the Lagrange multipliers.

Note that the variables in the list after the `ramsey_policy` or *stoch_simul* command can also contain multiplier names, but in a case-sensititve way (e.g. `MULT_1`). In that case, Dynare will for example display the IRFs of the respective multipliers when `irf>0`.

The planner objective must be declared with the *planner_objective* command and the planner discount factor with the `planner_discount` option.

*Options*

This command accepts all options of *stoch_simul*, plus:

**planner_discount = EXPRESSION**

See *planner_discount*.

**instruments = (VARIABLE_NAME,...)**

Declares instrument variables for the computation of the steady state under optimal policy. Requires a *steady_state_model* block or a `_steadystate.m` file. See below.

*Output*

This command generates all the output variables of *stoch_simul*. For specifying the initial values for the endogenous state variables (except for the Lagrange multipliers), see above.

*Steady state*

See *Ramsey steady state*.

### 4.22.2 Optimal policy under discretion

**Command:** **discretionary_policy** [VARIABLE_NAME...];

**Command:** **discretionary_policy**(OPTIONS...) [VARIABLE_NAME...];

This command computes an approximation of the optimal policy under discretion. The algorithm implemented is essentially an LQ solver, and is described by Dennis (2007).

You must ensure that your objective is quadratic. Regarding the model, it must either be linear or solved at first order with an analytical steady state provided. In the first case, you should set the `linear` option of the *model* block or *model_options* command.

It is possible to use the *estimation* command after the *discretionary_policy* command, in order to estimate the model with optimal policy under discretion and *evaluate_planner_objective* to compute welfare.

*Options*

This command accepts the same options as `ramsey_policy`, plus:

**discretionary_tol = NON-NEGATIVE DOUBLE**

Sets the tolerance level used to assess convergence of the solution algorithm. Default: `1e-7`.

**maxit = INTEGER**

Maximum number of iterations. Default: `3000`.

### 4.22.3 Optimal Simple Rules (OSR)

**Command:** **osr** [VARIABLE_NAME...];

**Command:** **osr**(OPTIONS...) [VARIABLE_NAME...];

This command computes optimal simple policy rules. There are two different supported approaches. The first one chooses a set of parameters $\gamma$ to maximize the expected lifetime value of a *planner_objective*:

$$\max_{\gamma} E \sum_{t=0}^{\infty} \tilde{\beta}^t U_t$$

where:

- $E$ denotes the unconditional expectations operator;
- $\gamma$ are parameters to be optimized. They must be specified as parameters in the `params` command and be entered in the `model` block;
- $U$ is the period objective function of the planner, specified with *planner_objective*.
- $\tilde{\beta}$ is the discount factor the planner employs (see *planner_discount*).

The subset of the model parameters over which the optimal simple rule is to be optimized, $\gamma$, must be listed with *osr_params*.

Dynare will invoke this approach whenever it encounters an *osr* command in conjunction with a *planner_objective*.

The second (legacy) approach will be triggered if an *optim_weights* block is present instead of a *planner_objective*. Dynare then solves linear-quadratic problems of the form:

$$\min_{\gamma} E(y_t' W y_t)$$

such that:

$$A_1 E_t y_{t+1} + A_2 y_t + A_3 y_{t-1} + C e_t = 0$$

where:

- $E$ denotes the unconditional expectations operator;
- $\gamma$ are parameters to be optimized. They must be elements of the matrices $A_1$, $A_2$, $A_3$, i.e. be specified as parameters in the `params` command and be entered in the `model` block;
- $y$ are the endogenous variables, specified in the `var` command, whose (co)-variance enters the loss function;
- $e$ are the exogenous stochastic shocks, specified in the `varexo`- command;
- $W$ is the weighting matrix;

The linear quadratic problem consists of choosing a subset of model parameters to minimize the weighted (co)-variance of a specified subset of endogenous variables, subject to a linear law of motion implied by the first-order conditions of the model. A few things are worth mentioning. First, $y$ denotes the selected endogenous variables' deviations from their steady state, i.e. in case they are not already mean 0 the variables entering the loss function are automatically demeaned so that the centered second moments are minimized. Second, `osr` with this syntax only solves linear quadratic problems of the type resulting from combining the specified quadratic loss function with a first-order approximation to the model's equilibrium conditions. The reason is that the first-order state-space representation is used to compute the unconditional (co)-variances. Hence, `osr` without a *`planner_objective`* is only compatible with `order=1`. Third, because the objective involves minimizing a weighted sum of unconditional second moments, those second moments must be finite. In particular, unit roots in $y$ are not allowed.

The weighting matrix $W$ used for the quadratic objective function is specified in the *`optim_weights`* block. By attaching weights to endogenous variables, the subset of endogenous variables entering the objective function, $y$, is implicitly specified.

The linear quadratic problem is solved using the numerical optimizer specified with *`opt_algo`*.

*Options*

The `osr` command will subsequently run *`stoch_simul`* and accepts the same options, including restricting the endogenous variables by listing them after the command, as *`stoch_simul`* (see *Stochastic solution and simulation*) plus

**`opt_algo = INTEGER`**

Specifies the optimizer for minimizing the objective function. The same solvers as for `mode_compute` (see *`mode_compute`*) are available, except for `5`, `6`, and `10`.

**`optim = (NAME, VALUE, ...)`**

A list of NAME`` and VALUE pairs. Can be used to set options for the optimization routines. The set of available options depends on the selected optimization routine (i.e. on the value of option *`opt_algo`*). See *`optim`*.

**`maxit = INTEGER`**

Determines the maximum number of iterations used in `opt_algo=4`. This option is now deprecated and will be removed in a future release of Dynare. Use `optim` instead to set optimizer-specific values. Default: `1000`.

**`tolf = DOUBLE`**

Convergence criterion for termination based on the function value used in `opt_algo=4`. Iteration will cease when it proves impossible to improve the function value by more than tolf. This option is now deprecated and will be removed in a future release of Dynare. Use `optim` instead to set optimizer-specific values. Default: `1e-7`.

**`analytic_derivation`**

**analytic_derivation = BOOLEAN**

Triggers estimation with analytic gradient of the objective function. It is only supported for the linear-quadratic approach at `order=1`

**analytic_derivation_mode = INTEGER**

Different ways to compute derivatives either analytically or numerically. Possible values are:

- `0`: efficient Sylvester equation method to compute analytical derivatives
- `1`: Kronecker products method to compute analytical derivatives (only at order=1)
- `-1`: numerical two-sided finite difference method to compute all identification Jacobians (numerical tolerance level is equal to `options_.dynatol.x`)
- `-2`: numerical two-sided finite difference method to compute derivatives of steady-state and dynamic model numerically, the identification Jacobians are then computed analytically (numerical tolerance level is equal to `options_.dynatol.x`)

Default: `0`.

It is only supported for the linear-quadratic approach at `order=1`.

**silent_optimizer**

See *silent_optimizer*.

**huge_number = DOUBLE**

Value for replacing the infinite bounds on parameters by finite numbers. Used by some optimizers for numerical reasons (see *huge_number*). Users need to make sure that the optimal parameters are not larger than this value. Default: `1e7`.

The value of the objective is stored in the variable `oo_.osr.objective_function` and the value of parameters at the optimum is stored in `oo_.osr.optim_params`. See below for more details.

After running `osr` the parameters entering the simple rule will be set to their optimal value so that subsequent runs of *stoch_simul* will be conducted at these values.

**Command: osr_params** `PARAMETER_NAME...;`

This command declares parameters to be optimized by `osr`.

**Block: optim_weights ;**

This block specifies quadratic objectives for optimal policy problems if the legacy linear-quadratic approach without a *planner_objective* is used.

More precisely, this block specifies the nonzero elements of the weight matrix $W$ used in the quadratic form of the objective function in `osr`.

An element of the diagonal of the weight matrix is given by a line of the form:

```
VARIABLE_NAME EXPRESSION;
```

An off-the-diagonal element of the weight matrix is given by a line of the form:

```
VARIABLE_NAME,  VARIABLE_NAME EXPRESSION;
```

*Example*

```
var y inflation r;
varexo y_ inf_;

parameters delta sigma alpha kappa gammarr gammax0 gammac0 gamma_y_ gamma_
 ↪inf_;

delta =  0.44;
kappa =  0.18;
```

(continues on next page)

(continued from previous page)

```
alpha =  0.48;
sigma = -0.06;

gammarr = 0;
gammax0 = 0.2;
gammac0 = 1.5;
gamma_y_ = 8;
gamma_inf_ = 3;

model(linear);
y  = delta * y(-1)  + (1-delta)*y(+1)+sigma *(r - inflation(+1)) + y_;
inflation  =   alpha * inflation(-1) + (1-alpha) * inflation(+1) + kappa*y␣
↪+ inf_;
r = gammax0*y(-1)+gammac0*inflation(-1)+gamma_y_*y_+gamma_inf_*inf_;
end;

shocks;
var y_; stderr 0.63;
var inf_; stderr 0.4;
end;

optim_weights;
inflation 1;
y 1;
y, inflation 0.5;
end;

osr_params gammax0 gammac0 gamma_y_ gamma_inf_;
osr y;
```

**Block: osr_params_bounds ;**

This block declares lower and upper bounds for parameters in the optimal simple rule. If not specified the optimization is unconstrained.

Each line has the following syntax:

```
PARAMETER_NAME, LOWER_BOUND, UPPER_BOUND;
```

Note that the use of this block requires the use of a constrained optimizer, i.e. setting *opt_algo* to 1, 2, 5 or 9.

*Example*

```
osr_params_bounds;
gamma_inf_, 0, 2.5;
end;

osr(opt_algo=9) y;
```

**MATLAB/Octave variable: oo_.osr.objective_function**

After an execution of the `osr` command, this variable contains the value of the objective under optimal policy.

**MATLAB/Octave variable: oo_.osr.optim_params**

After an execution of the `osr` command, this variable contains the value of parameters at the optimum, stored in fields of the form `oo_.osr.optim_params.PARAMETER_NAME`.

**MATLAB/Octave variable: M_.osr.param_names**

After an execution of the `osr` command, this cell contains the names of the parameters.

**MATLAB/Octave variable: M_.osr.param_indices**

After an execution of the `osr` command, this vector contains the indices of the OSR parameters in `M_.params`.

**MATLAB/Octave variable: M_.osr.param_bounds**

After an execution of the `osr` command, this two by number of OSR parameters matrix contains the lower and upper bounds of the parameters in the first and second column, respectively.

**MATLAB/Octave variable: M_.osr.variable_weights**

After an execution of the `osr` command, this sparse matrix contains the weighting matrix associated with the variables in the objective function. The field is only set if the legacy linear-quadratic approach without a *planner_objective* is used.

**MATLAB/Octave variable: M_.osr.variable_indices**

After an execution of the `osr` command, this vector contains the indices of the variables entering the objective function in `M_.endo_names`. The field is only set if the legacy linear-quadratic approach without a *planner_objective* is used.

## 4.23 Sensitivity and identification analysis

Dynare provides an interface to the global sensitivity analysis (GSA) toolbox (developed by the Joint Research Center (JRC) of the European Commission), which is now part of the official Dynare distribution. The GSA toolbox can be used to answer the following questions:

1. What is the domain of structural coefficients assuring the stability and determinacy of a DSGE model?
2. Which parameters mostly drive the fit of, e.g., GDP and which the fit of inflation? Is there any conflict between the optimal fit of one observed series versus another?
3. How to represent in a direct, albeit approximated, form the relationship between structural parameters and the reduced form of a rational expectations model?

The discussion of the methodologies and their application is described in Ratto (2008).

With respect to the previous version of the toolbox, in order to work properly, the GSA toolbox no longer requires that the Dynare estimation environment is set up.

### 4.23.1 Performing sensitivity analysis

**Command: sensitivity ;**

**Command: sensitivity(OPTIONS...);**

This command triggers sensitivity analysis on a DSGE model.

*Sampling Options*

**Nsam = INTEGER**

Size of the Monte Carlo sample. Default: `2048`.

**ilptau = INTEGER**

If equal to `1`, use $LP_\tau$ quasi-Monte Carlo. If equal to `0`, use LHS Monte Carlo. Default: `1`.

**pprior = INTEGER**

If equqal to `1`, sample from the prior distributions. If equal to `0`, sample from the multivariate normal $N(\bar{\theta}, \Sigma)$, where $\bar{\theta}$ is the posterior mode and $\Sigma = H^{-1}$, $H$ is the Hessian at the mode. Default: `1`.

**prior_range = INTEGER**

If equal to `1`, sample uniformly from prior ranges. If equal to `0`, sample from prior distributions. Default: `1`.

**morris = INTEGER**

If equal to 0, ANOVA mapping (Type I error) If equal to 1, Screening analysis (Type II error). If equal to 2, Analytic derivatives (similar to Type II error, only valid when identification=1). The ANOVA mapping requires the SS-ANOVA-R MATLAB Toolbox available at https://joint-research-centre.ec.europa.eu/system/files/2025-01/ss_anova_recurs.zip Default: 1 when identification=1, 0 otherwise.

**morris_nliv = INTEGER**

Number of levels in Morris design. Default: 6.

**morris_ntra = INTEGER**

Number trajectories in Morris design. Default: 20.

**ppost = INTEGER**

If equal to 1, use Metropolis posterior sample. If equal to 0, do not use Metropolis posterior sample. Default: 0.

NB: This overrides any other sampling option.

**neighborhood_width = DOUBLE**

When pprior=0 and ppost=0, allows for the sampling of parameters around the value specified in the mode_file, in the range xparam1 $\pm$ |xparam1 $\times$ neighborhood_width|. Default: 0.

*Stability Mapping Options*

**stab = INTEGER**

If equal to 1, perform stability mapping. If equal to 0, do not perform stability mapping. Default: 1.

**load_stab = INTEGER**

If equal to 1, load a previously created sample. If equal to 0, generate a new sample. Default: 0.

**alpha2_stab = DOUBLE**

Critical value for correlations $\rho$ in filtered samples: plot couples of parmaters with $|\rho| >$ alpha2_stab. Default: 0.

**pvalue_ks = DOUBLE**

The threshold $pvalue$ for significant Kolmogorov-Smirnov test (i.e. plot parameters with $pvalue <$ pvalue_ks). Default: 0.001.

**pvalue_corr = DOUBLE**

The threshold $pvalue$ for significant correlation in filtered samples (i.e. plot bivariate samples when $pvalue <$ pvalue_corr). Default: 1e-5.

*Reduced-Form Mapping Options*

**redform = INTEGER**

If equal to 1, prepare Monte Carlo sample of reduced-form matrices. If equal to 0, do not prepare Monte Carlo sample of reduced-form matrices. Default: 0.

**load_redform = INTEGER**

If equal to 1, load previously estimated mapping. If equal to 0, estimate the mapping of the reduced-form model. Default: 0.

**logtrans_redform = INTEGER**

If equal to 1, use log-transformed entries. If equal to 0, use raw entries. Default: 0.

**threshold_redform = [DOUBLE DOUBLE]**

The range over which the filtered Monte Carlo entries of the reduced-form coefficients should be analyzed. The first number is the lower bound and the second is the upper bound. An empty vector indicates that these entries will not be filtered. Default: empty.

**ksstat_redform = DOUBLE**

Critical value for Smirnov statistics $d$ when reduced-form entries are filtered. Default: 0.001.

**alpha2_redform = DOUBLE**

Critical value for correlations $\rho$ when reduced-form entries are filtered. Default: `1e-5`.

**namendo = (VARIABLE_NAME...)**

List of endogenous variables. ':' indicates all endogenous variables. Default: empty.

**namlagendo = (VARIABLE_NAME...)**

List of lagged endogenous variables. ':' indicates all lagged endogenous variables. Analyze entries [namendo $\times$ namlagendo] Default: empty.

**namexo = (VARIABLE_NAME...)**

List of exogenous variables. ':' indicates all exogenous variables. Analyze entries [namendo $\times$ namexo]. Default: empty.

*RMSE Options*

**rmse = INTEGER**

If equal to 1, perform RMSE analysis. If equal to `0`, do not perform RMSE analysis. Default: `0`.

**load_rmse = INTEGER**

If equal to 1, load previous RMSE analysis. If equal to `0`, make a new RMSE analysis. Default: `0`.

**lik_only = INTEGER**

If equal to 1, compute only likelihood and posterior. If equal to `0`, compute RMSE's for all observed series. Default: `0`.

**var_rmse = (VARIABLE_NAME...)**

List of observed series to be considered. ':' indicates all observed variables. Default: `varobs`.

**pfilt_rmse = DOUBLE**

Filtering threshold for RMSE's. Default: `0.1`.

**istart_rmse = INTEGER**

Value at which to start computing RMSE's (use `2` to avoid big intitial error). Default: `presample+1`.

**alpha_rmse = DOUBLE**

Critical value for Smirnov statistics $d$: plot parameters with $d >$ `alpha_rmse`. Default: `0.001`.

**alpha2_rmse = DOUBLE**

Critical value for correlation $\rho$: plot couples of parmaters with $|\rho| =$ `alpha2_rmse`. Default: `1e-5`.

**datafile = FILENAME**

See *datafile*.

**nobs = INTEGER**

**nobs = [INTEGER1:INTEGER2]**

See *nobs*.

**first_obs = INTEGER**

See *first_obs*.

**prefilter = INTEGER**

See *prefilter*.

**presample = INTEGER**

See *presample*.

**nograph**

See *nograph*.

**nodisplay**

See *nodisplay*.

**graph_format = FORMAT**

**graph_format = ( FORMAT, FORMAT... )**

See *graph_format*.

**conf_sig = DOUBLE**

See *conf_sig*.

**loglinear**

See *loglinear*.

**mode_file = FILENAME**

See *mode_file*.

**kalman_algo = INTEGER**

See *kalman_algo*.

*Identification Analysis Options*

**identification = INTEGER**

If equal to 1, performs identification analysis (forcing `redform=0` and `morris=1`) If equal to `0`, no identification analysis. Default: `0`.

**morris = INTEGER**

See *morris*.

**morris_nliv = INTEGER**

See *morris_nliv*.

**morris_ntra = INTEGER**

See *morris_ntra*.

**load_ident_files = INTEGER**

Loads previously performed identification analysis. Default: `0`.

**useautocorr = INTEGER**

Use autocorrelation matrices in place of autocovariance matrices in moments for identification analysis. Default: `0`.

**ar = INTEGER**

Maximum number of lags for moments in identification analysis. Default: 1.

**diffuse_filter = INTEGER**

See *diffuse_filter*.

**kalman_tol = DOUBLE**

See *kalman_tol = DOUBLE*.

**diffuse_kalman_tol = DOUBLE**

See *diffuse_kalman_tol = DOUBLE*.

**use_univariate_smoother_if_singularity_is_detected**

**use_univariate_smoother_if_singularity_is_detected = BOOLEAN**

See *use_univariate_smoother_if_singularity_is_detected*.

### 4.23.2 IRF/Moment calibration

The `irf_calibration` and `moment_calibration` blocks allow imposing implicit "endogenous" priors about IRFs and moments on the model. The way it works internally is that any parameter draw that is inconsistent with the "calibration" provided in these blocks is discarded, i.e. assigned a prior density of `0`. In the context of *sensitivity*, these restrictions allow tracing out which parameters are driving the model to satisfy or violate the given restrictions.

IRF and moment calibration can be defined in `irf_calibration` and `moment_calibration` blocks:

**Block: irf_calibration ;**

**Block:** **irf_calibration**(OPTIONS...);

This block allows defining IRF calibration criteria and is terminated by `end;`. To set IRF sign restrictions, the following syntax is used:

```
VARIABLE_NAME(INTEGER), EXOGENOUS_NAME, -;
VARIABLE_NAME(INTEGER:INTEGER), EXOGENOUS_NAME, +;
```

To set IRF restrictions with specific intervals, the following syntax is used:

```
VARIABLE_NAME(INTEGER), EXOGENOUS_NAME, [EXPRESSION, EXPRESSION];
VARIABLE_NAME(INTEGER:INTEGER), EXOGENOUS_NAME, [EXPRESSION, EXPRESSION];
```

When `(INTEGER:INTEGER)` is used, the restriction is considered to be fulfilled by a logical OR. A list of restrictions must always be fulfilled with logical AND.

*Options*

**relative_irf**

See *relative_irf*.

*Example*

```
irf_calibration;
y(1:4), e_ys, [-50, 50]; //[first year response with logical OR]
@#for ilag in 21:40
R_obs(@{ilag}), e_ys, [0, 6]; //[response from 5th to 10th years with␣
↪logical AND]
@#endfor
end;
```

**Block:** **moment_calibration** ;

**Block:** **moment_calibration**(OPTIONS...);

This block allows defining moment calibration criteria. This block is terminated by `end;`, and contains lines of the form:

```
VARIABLE_NAME1, VARIABLE_NAME2(+/-INTEGER), [EXPRESSION, EXPRESSION];
VARIABLE_NAME1, VARIABLE_NAME2(+/-INTEGER), +/-;
VARIABLE_NAME1, VARIABLE_NAME2(+/-(INTEGER:INTEGER)), [EXPRESSION, EXPRESSION];
VARIABLE_NAME1, VARIABLE_NAME2((-INTEGER:+INTEGER)), [EXPRESSION, EXPRESSION];
```

When `(INTEGER:INTEGER)` is used, the restriction is considered to be fulfilled by a logical OR. A list of restrictions must always be fulfilled with logical AND. The moment restrictions generally apply to auto- and cross-correlations between variables. The only exception is a restriction on the unconditional variance of an endogenous variable, specified as shown in the example below.

*Example*

```
moment_calibration;
y_obs,y_obs, [0.5, 1.5]; //[unconditional variance]
y_obs,y_obs(-(1:4)), +; //[sign restriction for first year␣
↪autocorrelation with logical OR]
@#for ilag in -2:2
y_obs,R_obs(@{ilag}), -; //[-2:2 cross correlation with logical AND]
@#endfor
@#for ilag in -4:4
y_obs,pie_obs(@{ilag}), -; //[-4_4 cross correlation with logical AND]
@#endfor
end;
```

### 4.23.3 Performing identification analysis

**Command:** `identification ;`

**Command:** `identification(OPTIONS...);`

This command triggers:

1. Theoretical identification analysis based on
   - moments as in Iskrev (2010)
   - spectral density as in Qu and Tkachenko (2012)
   - minimal system as in Komunjer and Ng (2011)
   - reduced-form solution and linear rational expectation model as in Ratto and Iskrev (2011)

   Note that for orders 2 and 3, all identification checks are based on the pruned state-space system as in Mutschler (2015). That is, theoretical moments and spectrum are computed from the pruned ABCD-system, whereas the minimal system criteria is based on the first-order system, but augmented by the theoretical (pruned) mean at order 2 or 3.
2. Identification strength analysis based on (theoretical or simulated) curvature of moment information matrix as in Ratto and Iskrev (2011)
3. Parameter checks based on nullspace and multicorrelation coefficients to determine which (combinations of) parameters are involved

*General Options*

**order = 1|2|3**

Order of approximation. At orders 2 and 3 identification is based on the pruned state-space system. Note that the order set in other functions does not overwrite the default. Default: `1`.

**parameter_set = OPTION**

See *parameter_set* for possible values. Default: `prior_mean`.

**prior_mc = INTEGER**

Size of Monte Carlo sample. Default: `1`.

**prior_range = INTEGER**

Triggers uniform sample within the range implied by the prior specifications (when `prior_mc>1`). Default: `0`.

**advanced = INTEGER**

If set to `1`, shows a more detailed analysis, comprised of an analysis for the linearized rational expectation model as well as the associated reduced-form solution. Further performs a bruteforce search of the groups of parameters best reproducing the behavior of each single parameter. The maximum dimension of the group searched is triggered by `max_dim_cova_group`. Default: `0`.

**max_dim_cova_group = INTEGER**

In the brute force search (performed when `advanced=1`) this option sets the maximum dimension of groups of parameters that best reproduce the behavior of each single model parameter. Default: `2`.

**gsa_sample_file = INTEGER|FILENAME**

If equal to `0`, do not use sample file. If equal to `1`, triggers gsa prior sample. If equal to `2`, triggers gsa Monte Carlo sample (i.e. loads a sample corresponding to `pprior=0` and `ppost=0` in the *sensitivity* options). If equal to `FILENAME` uses the provided path to a specific user defined sample file. Default: `0`.

*Kalman-filtering Options*

**diffuse_filter**

Deals with non-stationary cases. See *diffuse_filter*.

**lik_init = INTEGER**

See *lik_init*.

**kalman_algo = INTEGER**

See *kalman_algo*.

**kalman_tol = DOUBLE**

See *kalman_tol = DOUBLE*.

**diffuse_kalman_tol = DOUBLE**

See *diffuse_kalman_tol*.

**use_univariate_filters_if_singularity_is_detected**

**use_univariate_filters_if_singularity_is_detected = BOOLEAN**

See *use_univariate_filters_if_singularity_is_detected*.

*Numerical Options*

**analytic_derivation_mode = INTEGER**

See *analytic_derivation_mode = INTEGER*.

**normalize_jacobians = INTEGER**

If set to 1: Normalize Jacobian matrices by rescaling each row by its largest element in absolute value. Normalize Gram (or Hessian-type) matrices by transforming into correlation-type matrices. Default: 1

**tol_rank = DOUBLE**

Tolerance level used for rank computations. Default: `1.e-10`.

**tol_deriv = DOUBLE**

Tolerance level for selecting non-zero columns in Jacobians. Default: `1.e-8`.

**tol_sv = DOUBLE**

Tolerance level for selecting non-zero singular values. Default: `1.e-3`.

**schur_vec_tol = DOUBLE**

See *schur_vec_tol*.

*Identification Strength Options*

**no_identification_strength**

Disables computations of identification strength analysis based on sample information matrix.

**periods = INTEGER**

When the analytic Hessian is not available (i.e. with missing values or diffuse Kalman filter or univariate Kalman filter), this triggers the length of stochastic simulation to compute Simulated Moments Uncertainty. Default: `300`.

**replic = INTEGER**

When the analytic Hessian is not available, this triggers the number of replicas to compute Simulated Moments Uncertainty. Default: `100`.

*Moments Options*

**no_identification_moments**

Disables computations of identification check based on Iskrev (2010)'s J, i.e. derivative of first two moments.

**ar = INTEGER**

Number of lags of computed autocovariances/autocorrelations (theoretical moments) in Iskrev (2010)'s J criteria. Default: 1.

**useautocorr = INTEGER**

If equal to 1, compute derivatives of autocorrelation. If equal to 0, compute derivatives of autocovariances. Default: 0.

*Spectrum Options*

**no_identification_spectrum**

Disables computations of identification check based on Qu and Tkachenko (2012)'s G, i.e. Gram matrix of derivatives of first moment plus outer product of derivatives of spectral density.

**grid_nbr = INTEGER**

Number of grid points in [-pi;pi] to approximate the integral to compute Qu and Tkachenko (2012)'s G criteria. Default: 5000.

*Minimal State-Space System Options*

**no_identification_minimal**

Disables computations of identification check based on Komunjer and Ng (2011)'s D, i.e. minimal state-space system and observational equivalent spectral density transformations.

*Misc Options*

**nograph**

See *nograph*.

**nodisplay**

See *nodisplay*.

**graph_format = FORMAT**

**graph_format = ( FORMAT, FORMAT... )**

See *graph_format*.

**tex**

See *tex*.

*Debug Options*

**load_ident_files = INTEGER**

If equal to 1, allow Dynare to load previously computed analyzes. Default: 0.

**no_identification_reducedform**

Disables computations of identification check based on steady state and reduced-form solution.

**checks_via_subsets = INTEGER**

If equal to 1: finds problematic parameters in a bruteforce fashion: It computes the rank of the Jacobians for all possible parameter combinations. If the rank condition is not fullfilled, these parameter sets are flagged as non-identifiable. The maximum dimension of the group searched is triggered by max_dim_subsets_groups. Default: 0.

**max_dim_subsets_groups = INTEGER**

Sets the maximum dimension of groups of parameters for which the above bruteforce search is performed. Default: 4.

### 4.23.4 Types of analysis and output files

The sensitivity analysis toolbox includes several types of analyses. Sensitivity analysis results are saved locally in <mod_file>/gsa, where <mod_file>.mod is the name of the Dynare model file.

#### 4.23.4.1 Sampling

The following binary files are produced:

- <mod_file>_prior.mat: this file stores information about the analyses performed sampling from the prior, i.e. pprior=1 and ppost=0;

- <mod_file>_mc.mat: this file stores information about the analyses performed sampling from multivariate normal, i.e. pprior=0 and ppost=0;
- <mod_file>_post.mat: this file stores information about analyses performed using the Metropolis posterior sample, i.e. ppost=1.

#### 4.23.4.2 Stability Mapping

Figure files produced are of the form <mod_file>_prior_*.fig and store results for stability mapping from prior Monte Carlo samples:

- <mod_file>_prior_stable.fig: plots of the Smirnov test and the correlation analyses confronting the cdf of the sample fulfilling Blanchard-Kahn conditions (blue color) with the cdf of the rest of the sample (red color), i.e. either instability or indeterminacy or the solution could not be found (e.g. the steady-state solution could not be found by the solver);
- <mod_file>_prior_indeterm.fig: plots of the Smirnov test and the correlation analyses confronting the cdf of the sample producing indeterminacy (red color) with the cdf of the rest of the sample (blue color);
- <mod_file>_prior_unstable.fig: plots of the Smirnov test and the correlation analyses confronting the cdf of the sample producing explosive roots (red color) with the cdf of the rest of the sample (blue color);
- <mod_file>_prior_wrong.fig: plots of the Smirnov test and the correlation analyses confronting the cdf of the sample where the solution could not be found (e.g. the steady-state solution could not be found by the solver - red color) with the cdf of the rest of the sample (blue color);
- <mod_file>_prior_calib.fig: plots of the Smirnov test and the correlation analyses splitting the sample fulfilling Blanchard-Kahn conditions, by confronting the cdf of the sample where IRF/moment restrictions are matched (blue color) with the cdf where IRF/moment restrictions are NOT matched (red color);

Similar conventions apply for <mod_file>_mc_*.fig files, obtained when samples from multivariate normal are used.

#### 4.23.4.3 IRF/Moment restrictions

The following binary files are produced:

- <mod_file>_prior_restrictions.mat: this file stores information about the IRF/moment restriction analysis performed sampling from the prior ranges, i.e. pprior=1 and ppost=0;
- <mod_file>_mc_restrictions.mat: this file stores information about the IRF/moment restriction analysis performed sampling from multivariate normal, i.e. pprior=0 and ppost=0;
- <mod_file>_post_restrictions.mat: this file stores information about IRF/moment restriction analysis performed using the Metropolis posterior sample, i.e. ppost=1.

Figure files produced are of the form <mod_file>_prior_irf_calib_*.fig and <mod_file>_prior_moment_calib_*.fig and store results for mapping restrictions from prior Monte Carlo samples:

- <mod_file>_prior_irf_calib_<ENDO_NAME>_vs_<EXO_NAME>_<PERIOD>.fig: plots of the Smirnov test and the correlation analyses splitting the sample fulfilling Blanchard-Kahn conditions, by confronting the cdf of the sample where the individual IRF restriction <ENDO_NAME> vs. <EXO_NAME> at period(s) <PERIOD> is matched (blue color) with the cdf where the IRF restriction is NOT matched (red color)
- <mod_file>_prior_irf_calib_<ENDO_NAME>_vs_<EXO_NAME>_ALL.fig: plots of the Smirnov test and the correlation analyses splitting the sample fulfilling Blanchard-Kahn conditions, by confronting the cdf of the sample where ALL the individual IRF restrictions for the same couple <ENDO_NAME> vs. <EXO_NAME> are matched (blue color) with the cdf where the IRF restriction is NOT matched (red color)
- <mod_file>_prior_irf_restrictions.fig: plots visual information on the IRF restrictions compared to the actual Monte Carlo realization from prior sample.
- <mod_file>_prior_moment_calib_<ENDO_NAME1>_vs_<ENDO_NAME2>_<LAG>.fig: plots of the Smirnov test and the correlation analyses splitting the sample fulfilling Blanchard-Kahn conditions, by confronting the cdf of the sample where the individual acf/ccf moment restriction <ENDO_NAME1> vs.

<ENDO_NAME2> at lag(s) <LAG> is matched (blue color) with the cdf where the IRF restriction is NOT matched (red color)

- <mod_file>_prior_moment_calib_<ENDO_NAME>_vs_<EXO_NAME>_ALL.fig: plots of the Smirnov test and the correlation analyses splitting the sample fulfilling Blanchard-Kahn conditions, by confronting the cdf of the sample where ALL the individual acf/ccf moment restrictions for the same couple <ENDO_NAME1> vs. <ENDO_NAME2> are matched (blue color) with the cdf where the IRF restriction is NOT matched (red color)
- <mod_file>_prior_moment_restrictions.fig: plots visual information on the moment restrictions compared to the actual Monte Carlo realization from prior sample.

Similar conventions apply for <mod_file>_mc_*.fig and <mod_file>_post_*.fig files, obtained when samples from multivariate normal or from posterior are used.

#### 4.23.4.4 Reduced-Form Mapping

When the option threshold_redform is not set, or it is empty (the default), this analysis estimates a multivariate smoothing spline ANOVA model (the 'mapping') for the selected entries in the transition matrix of the shock matrix of the reduced-form first-order solution of the model. This mapping is done either with prior samples or with MC samples with neighborhood_width. Unless neighborhood_width is set with MC samples, the mapping of the reduced-form solution forces the use of samples from prior ranges or prior distributions, i.e.: pprior=1 and ppost=0. It uses 250 samples to optimize smoothing parameters and 1000 samples to compute the fit. The rest of the sample is used for out-of-sample validation. One can also load a previously estimated mapping with a new Monte Carlo sample, to look at the forecast for the new Monte Carlo sample.

The following synthetic figures are produced:

- <mod_file>_redform_<endo name>_vs_lags_*.fig: shows bar charts of the sensitivity indices for the ten most important parameters driving the reduced-form coefficients of the selected endogenous variables (namendo) versus lagged endogenous variables (namlagendo); suffix log indicates the results for log-transformed entries;
- <mod_file>_redform_<endo name>_vs_shocks_*.fig: shows bar charts of the sensitivity indices for the ten most important parameters driving the reduced-form coefficients of the selected endogenous variables (namendo) versus exogenous variables (namexo); suffix log indicates the results for log-transformed entries;
- <mod_file>_redform_gsa(_log).fig: shows bar chart of all sensitivity indices for each parameter: this allows one to notice parameters that have a minor effect for any of the reduced-form coefficients.

Detailed results of the analyses are shown in the subfolder <mod_file>/gsa/redform_prior for prior samples and in <mod_file>/gsa/redform_mc for MC samples with option neighborhood_width, where the detailed results of the estimation of the single functional relationships between parameters $\theta$ and reduced-form coefficient (denoted as $y$ hereafter) are stored in separate directories named as:

- <namendo>_vs_<namlagendo>, for the entries of the transition matrix;
- <namendo>_vs_<namexo>, for entries of the matrix of the shocks.

The following files are stored in each directory (we stick with prior sample, but similar conventions are used for MC samples):

- <mod_file>_prior_<namendo>_vs_<namexo>.fig: histogram and CDF plot of the MC sample of the individual entry of the shock matrix, in sample and out of sample fit of the ANOVA model;
- <mod_file>_prior_<namendo>_vs_<namexo>_map_SE.fig: for entries of the shock matrix it shows graphs of the estimated first-order ANOVA terms $y = f(\theta_i)$ for each deep parameter $\theta_i$;
- <mod_file>_prior_<namendo>_vs_<namlagendo>.fig: histogram and CDF plot of the MC sample of the individual entry of the transition matrix, in sample and out of sample fit of the ANOVA model;
- <mod_file>_prior_<namendo>_vs_<namlagendo>_map_SE.fig: for entries of the transition matrix it shows graphs of the estimated first-order ANOVA terms $y = f(\theta_i)$ for each deep parameter $\theta_i$;
- <mod_file>_prior_<namendo>_vs_<namexo>_map.mat, <mod_file>_<namendo>_vs_<namlagendo>_map.mat: these files store info in the estimation;

When option `logtrans_redform` is set, the ANOVA estimation is performed using a log-transformation of each y. The ANOVA mapping is then transformed back onto the original scale, to allow comparability with the baseline estimation. Graphs for this log-transformed case, are stored in the same folder in files denoted with the `_log` suffix.

When the option `threshold_redform` is set, the analysis is performed via Monte Carlo filtering, by displaying parameters that drive the individual entry `y` inside the range specified in `threshold_redform`. If no entry is found (or all entries are in the range), the MCF algorithm ignores the range specified in `threshold_redform` and performs the analysis splitting the MC sample of `y` into deciles. Setting `threshold_redform=[-inf inf]` triggers this approach for all `y`'s.

Results are stored in subdirectories of `<mod_file>/gsa/redform_prior` named

- `<mod_file>_prior_<namendo>_vs_<namlagendo>_threshold`, for the entries of the transition matrix;
- `<mod_file>_prior_<namendo>_vs_<namexo>_threshold`, for entries of the matrix of the shocks.

The files saved are named:

- `<mod_file>_prior_<namendo>_vs_<namexo>_threshold.fig`, `<mod_file>_<namendo>_vs_<namlagendo>_thresh fig`: graphical outputs;
- `<mod_file>_prior_<namendo>_vs_<namexo>_threshold.mat`, `<mod_file>_<namendo>_vs_<namlagendo>_thresh mat`: info on the analysis;

#### 4.23.4.5 RMSE

The RMSE analysis can be performed with different types of sampling options:

1. When `pprior=1` and `ppost=0`, the toolbox analyzes the RMSEs for the Monte Carlo sample obtained by sampling parameters from their prior distributions (or prior ranges): this analysis provides some hints about what parameter drives the fit of which observed series, prior to the full estimation;
2. When `pprior=0` and `ppost=0`, the toolbox analyzes the RMSEs for a multivariate normal Monte Carlo sample, with covariance matrix based on the inverse Hessian at the optimum: this analysis is useful when maximum likelihood estimation is done (i.e. no Bayesian estimation);
3. When `ppost=1` the toolbox analyzes the RMSEs for the posterior sample obtained by Dynare's Metropolis procedure.

The use of cases 2 and 3 requires an estimation step beforehand. To facilitate the sensitivity analysis after estimation, the *sensitivity* command also allows you to indicate some options of the *estimation* command. These are:

- `datafile`
- `nobs`
- `first_obs`
- `prefilter`
- `presample`
- `nograph`
- `nodisplay`
- `graph_format`
- `conf_sig`
- `loglinear`
- `mode_file`

Binary files produced my RMSE analysis are:

- `<mod_file>_prior_*.mat`: these files store the filtered and smoothed variables for the prior Monte Carlo sample, generated when doing RMSE analysis (`pprior=1` and `ppost=0`);
- `<mode_file>_mc_*.mat`: these files store the filtered and smoothed variables for the multivariate normal Monte Carlo sample, generated when doing RMSE analysis (`pprior=0` and `ppost=0`).

Figure files <mod_file>_rmse_*.fig store results for the RMSE analysis.

- `<mod_file>_rmse_prior*.fig`: save results for the analysis using prior Monte Carlo samples;
- `<mod_file>_rmse_mc*.fig`: save results for the analysis using multivariate normal Monte Carlo samples;
- `<mod_file>_rmse_post*.fig`: save results for the analysis using Metropolis posterior samples.

The following types of figures are saved (we show prior sample to fix ideas, but the same conventions are used for multivariate normal and posterior):

- `<mod_file>_rmse_prior_params_*.fig`: for each parameter, plots the cdfs corresponding to the best 10% RMSEs of each observed series (only those cdfs below the significance threshold `alpha_rmse`);
- `<mod_file>_rmse_prior_<var_obs>_*.fig`: if a parameter significantly affects the fit of `var_obs`, all possible trade-off's with other observables for same parameter are plotted;
- `<mod_file>_rmse_prior_<var_obs>_map.fig`: plots the MCF analysis of parameters significantly driving the fit the observed series `var_obs`;
- `<mod_file>_rmse_prior_lnlik*.fig`: for each observed series, plots in BLUE the cdf of the log-likelihood corresponding to the best 10% RMSEs, in RED the cdf of the rest of the sample and in BLACK the cdf of the full sample; this allows one to see the presence of some idiosyncratic behavior;
- `<mod_file>_rmse_prior_lnpost*.fig`: for each observed series, plots in BLUE the cdf of the log-posterior corresponding to the best 10% RMSEs, in RED the cdf of the rest of the sample and in BLACK the cdf of the full sample; this allows one to see idiosyncratic behavior;
- `<mod_file>_rmse_prior_lnprior*.fig`: for each observed series, plots in BLUE the cdf of the log-prior corresponding to the best 10% RMSEs, in RED the cdf of the rest of the sample and in BLACK the cdf of the full sample; this allows one to see idiosyncratic behavior;
- `<mod_file>_rmse_prior_lik.fig`: when `lik_only=1`, this shows the MCF tests for the filtering of the best 10% log-likelihood values;
- `<mod_file>_rmse_prior_post.fig`: when `lik_only=1`, this shows the MCF tests for the filtering of the best 10% log-posterior values.

#### 4.23.4.6 Screening Analysis

Screening analysis does not require any additional options with respect to those listed in *Sampling Options*. The toolbox performs all the analyses required and displays results.

The results of the screening analysis with Morris sampling design are stored in the subfolder `<mod_file>/gsa/screen`. The data file `<mod_file>_prior` stores all the information of the analysis (Morris sample, reduced-form coefficients, etc.).

Screening analysis merely concerns reduced-form coefficients. Similar synthetic bar charts as for the reduced-form analysis with Monte Carlo samples are saved:

- `<mod_file>_redform_<endo name>_vs_lags_*.fig`: shows bar charts of the elementary effect tests for the ten most important parameters driving the reduced-form coefficients of the selected endogenous variables (`namendo`) versus lagged endogenous variables (`namlagendo`);
- `<mod_file>_redform_<endo name>_vs_shocks_*.fig`: shows bar charts of the elementary effect tests for the ten most important parameters driving the reduced-form coefficients of the selected endogenous variables (`namendo`) versus exogenous variables (`namexo`);
- `<mod_file>_redform_screen.fig`: shows bar chart of all elementary effect tests for each parameter: this allows one to identify parameters that have a minor effect for any of the reduced-form coefficients.

#### 4.23.4.7 Identification Analysis

Setting the option `identification=1`, an identification analysis based on theoretical moments is performed. Sensitivity plots are provided that allow to infer which parameters are most likely to be less identifiable.

Prerequisite for properly running all the identification routines, is the keyword `identification`; in the Dynare model file. This keyword triggers the computation of analytic derivatives of the model with respect to estimated

parameters and shocks. This is required for option `morris=2`, which implements Iskrev (2010) identification analysis.

For example, the placing:

```
identification;
sensitivity(identification=1, morris=2);
```

in the Dynare model file triggers identification analysis using analytic derivatives as in Iskrev (2010), jointly with the mapping of the acceptable region.

The identification analysis with derivatives can also be triggered by the single command:

```
identification;
```

This does not do the mapping of acceptable regions for the model and uses the standard random sampler of Dynare. Additionally, using only `identification;` adds two additional identification checks: namely, of Qu and Tkachenko (2012) based on the spectral density and of Komunjer and Ng (2011) based on the minimal state-space system. It completely offsets any use of the sensitivity analysis toolbox.

## 4.24 Markov-switching SBVAR

Given a list of variables, observed variables and a data file, Dynare can be used to solve a Markov-switching SBVAR model according to Sims *et al.* (2008).[11] Having done this, you can create forecasts and compute the marginal data density, regime probabilities, IRFs, and variance decomposition of the model.

The commands have been modularized, allowing for multiple calls to the same command within a `<mod_file>.mod` file. The default is to use `<mod_file>` to tag the input (output) files used (produced) by the program. Thus, to call any command more than once within a `<mod_file>.mod` file, you must use the `*_tag` options described below.

**Command:** **`markov_switching`**`(OPTIONS...);`

Declares the Markov state variable information of a Markov-switching SBVAR model.

*Options*

**`chain = INTEGER`**

The Markov chain considered. Default: `none`.

**`number_of_regimes = INTEGER`**

Specifies the total number of regimes in the Markov Chain. This is a required option.

**`duration = DOUBLE | [ROW VECTOR OF DOUBLES]`**

The duration of the regimes or regimes. This is a required option. When passed a scalar real number, it specifies the average duration for all regimes in this chain. When passed a vector of size equal `number_of_regimes`, it specifies the average duration of the associated regimes (`1:number_of_regimes`) in this chain. An absorbing state can be specified through the *`restrictions`* option.

**`restrictions = [[ROW VECTOR OF 3 DOUBLES],[ROW VECTOR OF 3 DOUBLES],...]`**

Provides restrictions on this chain's regime transition matrix. Its vector argument takes three inputs of the form: `[current_period_regime, next_period_regime, transition_probability]`.

The first two entries are positive integers, and the third is a non-negative real in the set [0,1]. If restrictions are specified for every transition for a regime, the sum of the probabilities must be 1. Otherwise, if restrictions are not provided for every transition for a given regime the sum of the provided transition probabilities msut be <1. Regardless of the number of lags, the restrictions are specified for parameters at time `t` since the transition probability for a parameter at t is equal to that of the parameter at `t-1`.

In case of estimating a MS-DSGE model,[12] in addition the following options are allowed:

[11] If you want to align the paper with the description herein, please note that $A$ is $A^0$ and $F$ is $A^+$.

[12] An example can be found at https://git.dynare.org/Dynare/dynare/blob/master/tests/ms-dsge/test_ms_dsge.mod.

**parameters = [LIST OF PARAMETERS]**

This option specifies which parameters are controlled by this Markov Chain.

**number_of_lags = DOUBLE**

Provides the number of lags that each parameter can take within each regime in this chain.

*Example*

```
markov_switching(chain=1, duration=2.5, restrictions=[[1,3,0],[3,1,0]]);
```

Specifies a Markov-switching BVAR with a first chain with 3 regimes that all have a duration of 2.5 periods. The probability of directly going from regime 1 to regime 3 and vice versa is 0.

*Example*

```
markov_switching(chain=2, number_of_regimes=3, duration=[0.5, 2.5, 2.5],
parameter=[alpha, rho], number_of_lags=2, restrictions=[[1,3,0],[3,3,
↪1]]);
```

Specifies a Markov-switching DSGE model with a second chain with 3 regimes that have durations of 0.5, 2.5, and 2.5 periods, respectively. The switching parameters are `alpha` and `rho`. The probability of directly going from regime 1 to regime 3 is 0, while regime 3 is an absorbing state.

**Command:** **svar**(OPTIONS...);

Each Markov chain can control the switching of a set of parameters. We allow the parameters to be divided equation by equation and by variance or slope and intercept.

*Options*

**coefficients**

Specifies that only the slope and intercept in the given equations are controlled by the given chain. One, but not both, of `coefficients` or `variances` must appear. Default: `none`.

**variances**

Specifies that only variances in the given equations are controlled by the given chain. One, but not both, of `coefficients` or `variances` must appear. Default: `none`.

**equations**

Defines the equation controlled by the given chain. If not specified, then all equations are controlled by `chain`. Default: `none`.

**chain = INTEGER**

Specifies a Markov chain defined by *markov_switching*. Default: `none`.

**Command:** **sbvar**(OPTIONS...);

To be documented. For now, see the wiki: https://git.dynare.org/Dynare/dynare/-/wikis/MS-SBVAR

*Options*

`datafile`, `freq`, `initial_year`, `initial_subperiod`, `final_year`, `final_subperiod`, `data`, `vlist`, `vlistlog`, `vlistper`, `restriction_fname`, `nlags`, `cross_restrictions`, `contemp_reduced_form`, `real_pseudo_forecast`, `no_bayesian_prior`, `dummy_obs`, `nstates`, `indxscalesstates`, `alpha`, `beta`, `gsig2_lmdm`, `q_diag`, `flat_prior`, `ncsk`, `nstd`, `ninv`, `indxparr`, `indxovr`, `aband`, `indxap`, `apband`, `indximf`, `indxfore`, `foreband`, `indxgforhat`, `indxgimfhat`, `indxestima`, `indxgdls`, `eq_ms`, `cms`, `ncms`, `eq_cms`, `tlindx`, `tlnumber`, `cnum`, `forecast`, `coefficients_prior_hyperparameters`

**Block:** **svar_identification** ;

This block is terminated by `end;` and contains lines of the form:

```
UPPER_CHOLESKY;
LOWER_CHOLESKY;
EXCLUSION CONSTANTS;
EXCLUSION LAG INTEGER; EQUATION INTEGER, VARIABLE_NAME [[,] VARIABLE_NAME...];
RESTRICTION EQUATION INTEGER, EXPRESSION = EXPRESSION;
```

To be documented. For now, see the wiki: https://git.dynare.org/Dynare/dynare/-/wikis/MS-SBVAR

**Command:** `ms_estimation(OPTIONS...);`

Triggers the creation of an initialization file for, and the estimation of, a Markov-switching SBVAR model. At the end of the run, the $A^0$, $A^+$, $Q$ and $\zeta$ matrices are contained in the `oo_.ms` structure.

*General Options*

**`file_tag = FILENAME`**

The portion of the filename associated with this run. This will create the model initialization file, `init_<file_tag>.dat`. Default: `<mod_file>`.

**`output_file_tag = FILENAME`**

The portion of the output filename that will be assigned to this run. This will create, among other files, `est_final_<output_file_tag>.out`, `est_intermediate_<output_file_tag>.out`. Default: `<file_tag>`.

**`no_create_init`**

Do not create an initialization file for the model. Passing this option will cause the *Initialization Options* to be ignored. Further, the model will be generated from the output files associated with the previous estimation run (i.e. `est_final_<file_tag>.out`, `est_intermediate_<file_tag>.out` or `init_<file_tag>.dat`, searched for in sequential order). This functionality can be useful for continuing a previous estimation run to ensure convergence was reached or for reusing an initialization file. NB: If this option is not passed, the files from the previous estimation run will be overwritten. Default: off (i.e. create initialization file)

*Initialization Options*

**`coefficients_prior_hyperparameters = [DOUBLE1 DOUBLE2 ... DOUBLE6]`**

Sets the hyper parameters for the model. The six elements of the argument vector have the following interpretations:

`1`

Overall tightness for $A^0$ and $A^+$.

`2`

Relative tightness for $A^+$.

`3`

Relative tightness for the constant term.

`4`

Tightness on lag decay (range: 1.2 - 1.5); a faster decay produces better inflation process.

`5`

Weight on nvar sums of coeffs dummy observations (unit roots).

`6`

Weight on single dummy initial observation including constant.

Default: `[1.0 1.0 0.1 1.2 1.0 1.0]`

**`freq = INTEGER | monthly | quarterly | yearly`**

Frequency of the data (e.g. `monthly, 12`). Default: `4`.

**initial_year = INTEGER**

The first year of data. Default: none.

**initial_subperiod = INTEGER**

The first period of data (i.e. for quarterly data, an integer in `[1,4]`). Default: `1`.

**final_year = INTEGER**

The last year of data. Default: Set to encompass entire dataset.

**final_subperiod = INTEGER**

The final period of data (i.e. for monthly data, an integer in `[1,12]`. Default: When final_year is also missing, set to encompass entire dataset; when `final_year` is indicated, set to the maximum number of subperiods given the frequency (i.e. 4 for quarterly data, 12 for monthly,...).

**datafile = FILENAME**

See *datafile*.

**xls_sheet = QUOTED_STRING**

See *xls_sheet*.

**xls_range = RANGE**

See *xls_range*.

**nlags = INTEGER**

The number of lags in the model. Default: `1`.

**cross_restrictions**

Use cross $A^0$ and $A^+$ restrictions. Default: `off`.

**contemp_reduced_form**

Use contemporaneous recursive reduced form. Default: `off`.

**no_bayesian_prior**

Do not use Bayesian prior. Default: `off` (i.e. use Bayesian prior).

**alpha = INTEGER**

Alpha value for squared time-varying structural shock lambda. Default: `1`.

**beta = INTEGER**

Beta value for squared time-varying structural shock lambda. Default: `1`.

**gsig2_lmdm = INTEGER**

The variance for each independent $\lambda$ parameter under `SimsZha` restrictions. Default: `50^2`.

**specification = sims_zha | none**

This controls how restrictions are imposed to reduce the number of parameters. Default: `Random Walk`.

*Estimation Options*

**convergence_starting_value = DOUBLE**

This is the tolerance criterion for convergence and refers to changes in the objective function value. It should be rather loose since it will gradually be tightened during estimation. Default: `1e-3`.

**convergence_ending_value = DOUBLE**

The convergence criterion ending value. Values much smaller than square root machine epsilon are probably overkill. Default: `1e-6`.

**convergence_increment_value = DOUBLE**

Determines how quickly the convergence criterion moves from the starting value to the ending value. Default: `0.1`.

**max_iterations_starting_value = INTEGER**

This is the maximum number of iterations allowed in the hill-climbing optimization routine and should be rather small since it will gradually be increased during estimation. Default: `50`.

**max_iterations_increment_value = DOUBLE**

Determines how quickly the maximum number of iterations is increased. Default: `2`.

**max_block_iterations = INTEGER**

The parameters are divided into blocks and optimization proceeds over each block. After a set of blockwise optimizations are performed, the convergence criterion is checked and the blockwise optimizations are repeated if the criterion is violated. This controls the maximum number of times the blockwise optimization can be performed. Note that after the blockwise optimizations have converged, a single optimization over all the parameters is performed before updating the convergence value and maximum number of iterations. Default: `100`.

**max_repeated_optimization_runs = INTEGER**

The entire process described by *max_block_iterations* is repeated until improvement has stopped. This is the maximum number of times the process is allowed to repeat. Set this to `0` to not allow repetitions. Default: `10`.

**function_convergence_criterion = DOUBLE**

The convergence criterion for the objective function when `max_repeated_optimizations_runs` is positive. Default: `0.1`.

**parameter_convergence_criterion = DOUBLE**

The convergence criterion for parameter values when `max_repeated_optimizations_runs` is positive. Default: `0.1`.

**number_of_large_perturbations = INTEGER**

The entire process described by *max_block_iterations* is repeated with random starting values drawn from the posterior. This specifies the number of random starting values used. Set this to `0` to not use random starting values. A larger number should be specified to ensure that the entire parameter space has been covered. Default: `5`.

**number_of_small_perturbations = INTEGER**

The number of small perturbations to make after the large perturbations have stopped improving. Setting this number much above `10` is probably overkill. Default: `5`.

**number_of_posterior_draws_after_perturbation = INTEGER**

The number of consecutive posterior draws to make when producing a small perturbation. Because the posterior draws are serially correlated, a small number will result in a small perturbation. Default: `1`.

**max_number_of_stages = INTEGER**

The small and large perturbation are repeated until improvement has stopped. This specifies the maximum number of stages allowed. Default: `20`.

**random_function_convergence_criterion = DOUBLE**

The convergence criterion for the objective function when `number_of_large_perturbations` is positive. Default: `0.1`.

**random_parameter_convergence_criterion = DOUBLE**

The convergence criterion for parameter values when `number_of_large_perturbations` is positive. Default: `0.1`.

*Example*

```
ms_estimation(datafile=data, initial_year=1959, final_year=2005,
nlags=4, max_repeated_optimization_runs=1, max_number_of_stages=0);

ms_estimation(file_tag=second_run, datafile=data, initial_year=1959,
final_year=2005, nlags=4, max_repeated_optimization_runs=1,
```

(continues on next page)

(continued from previous page)

```
max_number_of_stages=0);

ms_estimation(file_tag=second_run, output_file_tag=third_run,
no_create_init, max_repeated_optimization_runs=5,
number_of_large_perturbations=10);
```

**Command:** **ms_simulation** ;

**Command:** **ms_simulation**(OPTIONS...);

Simulates a Markov-switching SBVAR model.

*Options*

**file_tag = FILENAME**

The portion of the filename associated with the `ms_estimation` run. Default: `<mod_file>`.

**output_file_tag = FILENAME**

The portion of the output filename that will be assigned to this run. Default: `<file_tag>`.

**mh_replic = INTEGER**

The number of draws to save. Default: `10,000`.

**drop = INTEGER**

The number of burn-in draws. Default: `0.1*mh_replic*thinning_factor`.

**thinning_factor = INTEGER**

The total number of draws is equal to `thinning_factor*mh_replic+drop`. Default: `1`.

**adaptive_mh_draws = INTEGER**

Tuning period for Metropolis-Hastings draws. Default: `30,000`.

**save_draws**

Save all elements of $A^0$, $A^+$, $Q$, and $\zeta$, to a file named `draws_<<file_tag>>.out` with each draw on a separate line. A file that describes how these matrices are laid out is contained in `draws_header_<<file_tag>>.out`. A file called `load_flat_file.m` is provided to simplify loading the saved files into the corresponding variables `A0`, `Aplus`, `Q`, and `Zeta` in your MATLAB/Octave workspace. Default: `off`.

*Example*

```
ms_simulation(file_tag=second_run);
ms_simulation(file_tag=third_run, mh_replic=5000, thinning_factor=3);
```

**Command:** **ms_compute_mdd** ;

**Command:** **ms_compute_mdd**(OPTIONS...);

Computes the marginal data density of a Markov-switching SBVAR model from the posterior draws. At the end of the run, the Muller and Bridged log marginal densities are contained in the `oo_.ms` structure.

*Options*

**file_tag = FILENAME**

See *file_tag*.

**output_file_tag = FILENAME**

See *output_file_tag*.

**simulation_file_tag = FILENAME**

The portion of the filename associated with the simulation run. Default: `<file_tag>`.

**proposal_type = INTEGER**

The proposal type:

1

Gaussian.

2

Power.

3

Truncated Power.

4

Step.

5

Truncated Gaussian.

Default: 3

**proposal_lower_bound = DOUBLE**

The lower cutoff in terms of probability. Not used for proposal_type in [1,2]. Required for all other proposal types. Default: 0.1.

**proposal_upper_bound = DOUBLE**

The upper cutoff in terms of probability. Not used for proposal_type equal to 1. Required for all other proposal types. Default: 0.9.

**mdd_proposal_draws = INTEGER**

The number of proposal draws. Default: 100,000.

**mdd_use_mean_center**

Use the posterior mean as center. Default: off.

**Command: ms_compute_probabilities ;**

**Command: ms_compute_probabilities**(OPTIONS...);

Computes smoothed regime probabilities of a Markov-switching SBVAR model. Output .eps files are contained in <output_file_tag/Output/Probabilities>.

*Options*

**file_tag = FILENAME**

See *file_tag*.

**output_file_tag = FILENAME**

See *output_file_tag*.

**filtered_probabilities**

Filtered probabilities are computed instead of smoothed. Default: off.

**real_time_smoothed**

Smoothed probabilities are computed based on time t information for $0 \leq t \leq nobs$. Default: off

**Command: ms_irf ;**

**Command: ms_irf**(OPTIONS...);

Computes impulse response functions for a Markov-switching SBVAR model. Output .eps files are contained in <output_file_tag/Output/IRF>, while data files are contained in <output_file_tag/IRF>.

*Options*

**file_tag = FILENAME**

See *file_tag*.

**output_file_tag = FILENAME**

See *output_file_tag*.

**simulation_file_tag = FILENAME**

See *simulation_file_tag*.

**horizon = INTEGER**

The forecast horizon. Default: `12`.

**filtered_probabilities**

Uses filtered probabilities at the end of the sample as initial conditions for regime probabilities. Only one of `filtered_probabilities`, `regime` and `regimes` may be passed. Default: `off`.

**error_band_percentiles = [DOUBLE1 ...]**

The percentiles to compute. Default: `[0.16 0.50 0.84]`. If `median` is passed, the default is `[0.5]`.

**shock_draws = INTEGER**

The number of regime paths to draw. Default: `10,000`.

**shocks_per_parameter = INTEGER**

The number of regime paths to draw under parameter uncertainty. Default: `10`.

**thinning_factor = INTEGER**

Only 1/`thinning_factor` of the draws in posterior draws file are used. Default: `1`.

**free_parameters = NUMERICAL_VECTOR**

A vector of free parameters to initialize theta of the model. Default: use estimated parameters

**parameter_uncertainty**

Calculate IRFs under parameter uncertainty. Requires that `ms_simulation` has been run. Default: `off`.

**regime = INTEGER**

Given the data and model parameters, what is the ergodic probability of being in the specified regime. Only one of `filtered_probabilities`, `regime` and `regimes` may be passed. Default: `off`.

**regimes**

Describes the evolution of regimes. Only one of `filtered_probabilities`, `regime` and `regimes` may be passed. Default: `off`.

**median**

A shortcut to setting `error_band_percentiles=[0.5]`. Default: `off`.

**Command: ms_forecast ;**

**Command: ms_forecast**`(OPTIONS...);`

Generates forecasts for a Markov-switching SBVAR model. Output `.eps` files are contained in `<output_file_tag/Output/Forecast>`, while data files are contained in `<output_file_tag/Forecast>`.

*Options*

**file_tag = FILENAME**

See *file_tag*.

**output_file_tag = FILENAME**

See *output_file_tag*.

**simulation_file_tag = FILENAME**

See *simulation_file_tag*.

**data_obs_nbr = INTEGER**

The number of data points included in the output. Default: `0`.

**error_band_percentiles = [DOUBLE1 ...]**

See *error_band_percentiles*.

**shock_draws = INTEGER**

See *shock_draws*.

**shocks_per_parameter = INTEGER**

See *shocks_per_parameter*.

**thinning_factor = INTEGER**

See *thinning_factor*.

**free_parameters = NUMERICAL_VECTOR**

See *free_parameters*.

**parameter_uncertainty**

See *parameter_uncertainty*.

**regime = INTEGER**

See *regime*.

**regimes**

See *regimes*.

**median**

See *median*.

**horizon = INTEGER**

See *horizon*.

**Command: ms_variance_decomposition ;**

**Command: ms_variance_decomposition**(OPTIONS...);

Computes the variance decomposition for a Markov-switching SBVAR model. Output `.eps` files are contained in <output_file_tag/Output/Variance_Decomposition>, while data files are contained in <output_file_tag/Variance_Decomposition>.

*Options*

**file_tag = FILENAME**

See *file_tag*.

**output_file_tag = FILENAME**

See *output_file_tag*.

**simulation_file_tag = FILENAME**

See *simulation_file_tag*.

**horizon = INTEGER**

See *horizon*.

**filtered_probabilities**

See *filtered_probabilities*.

**no_error_bands**

Do not output percentile error bands (i.e. compute mean). Default: `off` (i.e. output error bands)

**error_band_percentiles = [DOUBLE1 ...]**

See *error_band_percentiles*.

**shock_draws = INTEGER**

See *shock_draws*.

**shocks_per_parameter = INTEGER**

See *shocks_per_parameter*.

**thinning_factor = INTEGER**

See *thinning_factor*.

**free_parameters = NUMERICAL_VECTOR**

See *free_parameters*.

**parameter_uncertainty**

See *parameter_uncertainty*.

**regime = INTEGER**

See *regime*.

**regimes**

See *regimes*.

## 4.25 Epilogue Variables

**Block: epilogue ;**

The epilogue block is useful for computing output variables of interest that may not be necessarily defined in the model (e.g. various kinds of real/nominal shares or relative prices, or annualized variables out of a quarterly model).

It can also provide several advantages in terms of computational efficiency and flexibility:

- You can calculate variables in the epilogue block after smoothers/simulations have already been run without adding the new definitions and equations and rerunning smoothers/simulations. Even posterior smoother subdraws can be recycled for computing epilogue variables without rerunning subdraws with the new definitions and equations.
- You can also reduce the state-space dimension in data filtering/smoothing. Assume, for example, you want annualized variables as outputs. If you define an annual growth rate in a quarterly model, you need lags up to order 7 of the associated quarterly variable; in a medium/large scale model this would just blow up the state dimension and increase by a huge amount the computing time of a smoother.

The `epilogue` block is terminated by `end;` and contains lines of the form:

NAME = EXPRESSION;

The equations are evaluated in the order in which they are written, each one over the whole sample (in a single vectorized operation when it is not a recursion). As a consequence, `EXPRESSION` may involve lags of the endogenous, exogenous and epilogue variables, subject to the following restrictions:

- leads are forbidden;
- an epilogue variable must have been defined earlier in the block to be referenced;
- only the lags of the variable being defined may appear on the right hand side of its own equation, not its contemporaneous value (which would make the equation implicit).

An equation is skipped, and the variable it defines left unchanged, if one of the variables it references has no observation at all. Otherwise it is evaluated from the first period at which all the referenced variables are available, given the lags at which they appear; the earlier periods are filled with `NaN`s. In particular, a variable defined by a recursion (such as `x` in the example below) must be initialized by the caller, otherwise its equation is skipped.

***Example***

```
epilogue;
// annualized level of y
ya = exp(y)+exp(y(-1))+exp(y(-2))+exp(y(-3));
// annualized growth rate of y
gya = ya/ya(-4)-1;
// autoregressive variable, must be initialized by the caller
x = .7*x(-1)+gya;
end;
```

Dynare evaluates the epilogue block where it needs it, for example in the shock decomposition when the *with_epilogue* option is used. The block can also be applied afterwards to a dataset of your own:

**MATLAB/Octave command:** `ds = evaluate_epilogue(ds, M_, oo_)`

Evaluates the epilogue block on the `dseries` object `ds` and returns a new object in which the epilogue variables have been added (or replaced, if they were already present). The input object is left untouched.

***Example***

```
ds = dseries('mydata.csv');
ds = evaluate_epilogue(ds, M_, oo_);
```

## 4.26 Heterogeneity

Dynare provides tools for solving models with microeconomic heterogeneity, where a continuum of agents differs in wealth, income, or employment status. Aggregate dynamics arise from the interaction between individual decisions and the distribution of agents' states.

The implementation notably handles incomplete-markets macroeconomic models (such as HANK models) and combines elements from Bhandari *et al.* (2023) and Auclert *et al.* (2021).

Examples throughout this section draw on Krusell and Smith (1998), a benchmark for economies with idiosyncratic and aggregate shocks.

The details of the solution algorithms implemented in Dynare are given in Rion (2026).

### 4.26.1 Declaring Heterogeneous Agent Models

#### 4.26.1.1 Heterogeneity dimension

**Command:** `heterogeneity_dimension NAME;`

Declares a heterogeneity dimension for the model. This command must appear before any variable declarations that use this dimension.

Currently, Dynare supports only **one** heterogeneity dimension per model (for example, `households`).

*Example*:

```
heterogeneity_dimension households;
```

#### 4.26.1.2 Heterogeneous Variable Declarations

Variables that vary across heterogeneous agents must be declared with the `heterogeneity` option:

**Command:** `var(heterogeneity=NAME) VAR_NAME...;`

**Command:** `varexo(heterogeneity=NAME) VAR_NAME...;`

Declares endogenous or exogenous variables defined for each agent in the specified heterogeneity dimension.

*Example*:

```
heterogeneity_dimension households;

var(heterogeneity=households)
    c  (long_name='consumption')
    a  (long_name='assets')
    Va (long_name='marginal value of assets')
;

varexo(heterogeneity=households)
    e  (long_name='idiosyncratic productivity shock')
;
```

#### 4.26.1.3 Heterogeneous Agent Model Block

**Block:** `model(heterogeneity=NAME);`

Declares equations describing the behavior of heterogeneous agents. The `heterogeneity` option specifies which heterogeneity dimension these equations pertain to.

**Allowed equations**

In this block, equations are subject to the following restrictions:

- Heterogeneous exogenous variables declared in *varexo(heterogeneity=NAME)* may only appear at time $t$ (no leads or lags).
- Heterogeneous endogenous variables declared in *var(heterogeneity=NAME)* may appear at times $t-1$, $t$, and $t+1$ only (maximum lag is $-1$, maximum lead is $+1$).
- Expressions that combine forward-looking heterogeneous variables (lead $\geq 1$) with lagged state variables (lag $= -1$) must be separable. Specifically, the following rules apply when an expression contains both leads and lags:
  - `+`, `-`, `=`: always separable.
  - `*`: separable if one factor contains all the leads and the other contains none (both orders are accepted).
  - `/`: separable only when the **numerator** contains the leads; e.g., `c(+1)/a(-1)` is accepted while `a(-1)/c(+1)` is rejected.
  - Unary functions (`log`, `exp`, etc.): non-separable if the argument contains both leads and lags; e.g., `log(k(-1) + c(+1))` is rejected.
  - `^` and other nonlinear binary operators: non-separable if operands span both leads and lags.

  For example, `k(-1) + c(+1)` and `c(+1) * a(-1)` are valid because leads and lags appear in separable terms, while `log(k(-1) + c(+1))` or `(k(-1) + c(+1))^2` are rejected.

**Complementarity conditions**

The complementarity operator $\perp$ (Unicode U+27C2) or its alternative pure-ASCII syntax `_|_` can be used to specify inequality constraints on variables. Equations with complementarity conditions should follow the same conventions as described in *lmmcp*.

*Example*:

```
model(heterogeneity=households);
    c^(-1/eis)-beta*Va(+1)=0 ⊥ a>=0;
    (1+r)*a(-1)+w*e=c+a;
    Va = (1+r)*c^(-1/eis);
end;
```

#### 4.26.1.4 Heterogeneous Shocks Block

**Block:** `shocks(heterogeneity=NAME);`

Declares variances of exogenous shocks for a specific heterogeneity dimension. The `heterogeneity` option specifies which heterogeneity dimension these shocks belong to.

*Example*:

```
shocks(heterogeneity=households);
    var eps_e; stderr 0.01;
end;
```

#### 4.26.1.5 Aggregate Variable Declarations

Standard `var` and `varexo` declarations (without the `heterogeneity` option) are used for aggregate variables that apply to the economy as a whole.

*Example*:

```
var
    Y  (long_name = 'aggregate output')
    r  (long_name = 'rate of return on capital')
    w  (long_name = 'wage rate')
    K  (long_name = 'aggregate capital')
;

varexo Z  (long_name = 'aggregate productivity shock');
```

#### 4.26.1.6 Aggregate Shocks Block

The standard `shocks` block (without the `heterogeneity` option) declares variances of aggregate exogenous shocks.

*Example*:

```
shocks;
    var Z; stderr 0.01;
end;
```

#### 4.26.1.7 Aggregate Model Block

The standard `model` block (without the `heterogeneity` option) describes aggregate dynamics and equilibrium conditions linking aggregate and heterogeneous variables.

Dynare provides the `SUM` operator for aggregating heterogeneous variables within the aggregate model block:

**SUM(var)**

Computes the integral of a heterogeneous variable over the stationary distribution:

$$\begin{aligned} textSUM(x) = \\ intx(s) \\ ,d \\ mu(s) \end{aligned}$$

where
$mu$ is the distribution of agents over individual states $s$.

*Example*:

```
model;
    Y = (Z_ss+Z) * K(-1)^alpha * L^(1-alpha);
    r = alpha * (Z_ss+Z) * (K(-1)/L)^(alpha-1) + delta;
    w = (1-alpha) * (Z_ss+Z) * (K(-1)/L)^alpha;
    K - SUM(a) = 0;
end;
```

### 4.26.2 Solving Heterogeneous Agent Models

Dynare provides two approaches for initializing the steady state of a heterogeneous-agent model:

- *heterogeneity_load_steady_state*: loads a pre-computed steady state from a MAT file or a workspace variable.
- *heterogeneity_compute_steady_state*: computes the steady state numerically.

#### 4.26.2.1 Loading a pre-computed steady state

**Command: heterogeneity_load_steady_state ;**

**Command: heterogeneity_load_steady_state**(OPTIONS...);

Loads the steady state for a heterogeneous-agent model. The steady state represents the set of policy functions of heterogeneous agents and their stationary distribution when aggregate variables are at their steady-state values. It must be computed externally and provided either from a MAT file or from a workspace variable.

This command verifies that the residuals on aggregate equations are small enough (up to the tolerance set with the `tolf` option). On the other hand, it does not check the residuals of the heterogeneous equations, so it is up to the user to ensure that those are within an acceptable boundary for the problem under consideration.

*Options*

**filename = FILENAME**

Path to the MAT file containing the steady-state structure. It must be included in quotes if the filename contains a path or an extension. If the `.mat` extension is omitted, it will be added automatically. If not provided, the steady-state structure is loaded from the base workspace using the variable name specified by *variable*.

**variable = STRING**

When *filename* is provided, this is the variable name within the MAT file to load. When *filename* is not provided, this is the name of a variable in the current workspace containing the steady-state structure. This is useful when the structure is built in a `verbatim` block and does not need to be saved to disk. Default: `'steady_state'`

**tolf = DOUBLE**

Tolerance for the check over residuals of the aggregate equations. Default: `1e-6`.

*Steady-State Structure*

The steady-state structure (whether loaded from a MAT file or from the workspace) must contain the following fields.

**Key Terminology:**

- **States**: Pre-determined endogenous heterogeneous variables that appear with a time lag (e.g., `a(-1)`) in the `model(heterogeneity=...)` block equations. These are the state variables over which policy functions are defined.
- **Shocks**: Discretized idiosyncratic exogenous processes, typically AR(1) processes with i.i.d. Gaussian innovations. These must be declared in `varexo(heterogeneity=...)` and discretized into finite-state Markov chains with grids and transition matrices.
- **Policy functions**: Decision rules that map current shocks and states to agents' choices and values (e.g., consumption `c(e,a)`, next-period assets `a'(e,a)`, marginal value `Va(e,a)`).

**Structure Fields:**

**steady_state.agg (structure)**

Steady-state values for all the endogenous aggregate variables declared in the `var` statement. Each field should be a scalar real number corresponding to an aggregate variable declared in `var`. When endogenous or exogenous aggregate variables appear with more than one lead or lag in the aggregate model block, Dynare internally transforms the model into an augmented aggregate state space with aggregate auxiliary variables. Users only need to provide steady-state values for the original aggregate variables declared in the `var` statement. Dynare automatically computes the steady-state values of the auxiliary aggregate variables.

*Example (Krusell-Smith model)*:

```
steady_state.agg.r = 0.0245      % Interest rate
steady_state.agg.w = 1.0231      % Wage
steady_state.agg.Y = 0.5531      % Output
steady_state.agg.K = 10.4667     % Capital
```

**steady_state.pol (structure)**

Policy functions with subfields:

- `pol.grids`: structure containing all the state variable grids as column vectors. Each field corresponds to a state variable (pre-determined endogenous heterogeneous variable).

  *Example*:

  ```
  steady_state.pol.grids.a  % Column vector [500×1] containing asset grid␣
  ↪points
  ```

- `pol.values`: structure containing policy functions as multidimensional arrays for all the variables declared using the `var(heterogeneity=...)` statement. Each field corresponds to a heterogeneous endogenous variable. Dimensions are ordered according to `pol.order`.

  *Example*:

  ```
  steady_state.pol.values.c   % Consumption policy [7×500]: dimension␣
  ↪1=shocks (e), dimension 2=states (a)
  steady_state.pol.values.a   % Next-period assets policy [7×500]
  steady_state.pol.values.Va  % Marginal value of assets [7×500]
  ```

- `pol.order`: cell array specifying the dimension ordering for policy function arrays. The first dimensions correspond to shocks, followed by states.

  *Example*:

  ```
  steady_state.pol.order = {'e', 'a'}  % Dimension 1 = shock e, Dimension␣
  ↪2 = state a
  ```

**steady_state.shocks (structure)**

Discretized idiosyncratic shocks with subfields:

- `shocks.grids`: structure containing discretized shock values as column vectors. Each field corresponds to an exogenous shock declared in `varexo(heterogeneity=...)`.

  *Example*:

  ```
  steady_state.shocks.grids.e  % Column vector [7×1] containing␣
  ↪productivity shock values
  ```

- `shocks.Pi`: structure containing Markov transition matrices. Each field is a matrix where element (i,j) gives the probability of transitioning from shock state i to shock state j. Each row must sum to 1.

  *Example*:

```
steady_state.shocks.Pi.e  % Matrix [7×7] with row sums = 1, all elements␣
↪>= 0
```

Both `grids` and `Pi` must be provided for all shocks to ensure consistency with the policy function values defined on the tensor-product grid. See the *rouwenhorst* helper function for a convenient way to produce these objects.

**steady_state.d (structure)**
The stationary distribution:

- `d.grids`: structure with distribution state grids (optional; defaults to `pol.grids`). Allows using different grids for the distribution than for policy functions. When specified, each grid must have the same bounds as its counterpart in `pol.grids`.
- `d.hist`: multidimensional array containing the histogram of the stationary distribution. Must sum to 1.0. Dimensions are ordered according to `d.order`.

  *Example*:

  ```
  steady_state.d.hist  % Array [7×500] representing fraction of agents at␣
  ↪each (e,a) point
            % sum(ss.d.hist(:)) == 1.0
  ```

- `d.order`: cell array specifying the dimension ordering of the distribution. This ensures that `d.hist` and `pol.values` are aligned, even if their grid dimensions differ. If not specified, defaults to `pol.order`.

  *Example*:

  ```
  steady_state.d.order = {'e', 'a'}  % Same ordering as pol.order
  ```

*Output*

The command outputs various objects into `oo_.heterogeneity`.

**MATLAB/Octave variable: oo_.heterogeneity**

Structure storing the objects related to the heterogeneous-agent model solution. Populated progressively by *heterogeneity_load_steady_state* or *heterogeneity_compute_steady_state*, then *heterogeneity_solve*. Contains the steady state (policy functions, stationary distribution, aggregate values) and, after solving, the linearized decision rules in the `dr` subfield.

**MATLAB/Octave variable: oo_.heterogeneity.steady_state**

Structure containing the steady-state solution for the heterogeneous-agent block. Fields `agg`, `pol`, `shocks`, and `d` follow the format described under *heterogeneity_load_steady_state*.

#### 4.26.2.2 Computing the steady state

**Command: heterogeneity_compute_steady_state ;**

**Command: heterogeneity_compute_steady_state(OPTIONS...);**

Computes the steady state of a heterogeneous-agent model. The steady state consists of policy functions, a stationary distribution, and aggregate variable values such that all equilibrium conditions hold. Free parameters can be calibrated simultaneously to satisfy market-clearing or other aggregate conditions.

**Algorithm**

The command solves for free parameter values (if any free parameter is specified) by finding zeros of a system of aggregate residual equations using a Broyden solver. Each residual evaluation proceeds in three steps:

1. **Time iteration:** For given free parameter values, compute the policy functions using time iteration on the equations in the heterogeneous-agent model block. MCP constraints (e.g., borrowing limits) are handled via a Fischer-Burmeister complementarity function. At each iteration, a nonlinear system is solved at every grid point using a trust-region solver.

2. **Forward iteration:** The stationary distribution consistent with the converged policy functions is computed by iterating the transition operator forward until convergence.
3. **Aggregation:** Aggregated heterogeneous variables (i.e., the auxiliary variables defined by `SUM` operators) are computed by integrating policy functions against the distribution.

The Broyden solver then evaluates the residuals of the target aggregate equations. When no free parameters are specified, the command simply evaluates these three steps once and reports the residuals.

Note that this command takes as given the value of aggregate endogenous variables, and does not try to recompute them. Hence, it is the responsibility of the user to provide aggregate endogenous values that are consistent with a zero residual on non-target aggregate equations.

**Target equations:** By default, the aggregate equations containing a `SUM` operator are used as target equations. This can be overridden with the `calibration_target_equations` option. The number of target equations must equal the number of free parameters.

**Initial guess**

An initial guess must be provided via a MAT file or a workspace variable (see the *filename* and *variable* options). The structure must contain the same fields as described in *heterogeneity_load_steady_state* (`agg`, `pol`, `shocks`, `d`), plus an optional `free_parameters` field for parameter calibration. The `d.hist` field is ignored here since the stationary distribution is recomputed by the forward iteration step; only `d.grids` (and optionally `d.order`) are used.

**steady_state.free_parameters (structure, optional)**

Parameters to calibrate. Each field name must match a parameter declared in *parameters* and contain a structure with:

- `initial_guess`: *Required.* Starting value for the Broyden solver.
- `lower_bound`: Optional lower bound on the parameter.
- `upper_bound`: Optional upper bound on the parameter.

*Example*:

```
steady_state.free_parameters.beta.initial_guess = 0.986;
steady_state.free_parameters.beta.lower_bound = 0.9;
steady_state.free_parameters.beta.upper_bound = 0.999;
```

*Options*

**`filename = FILENAME`**

See *filename* in *heterogeneity_load_steady_state*.

**`variable = STRING`**

See *variable* in *heterogeneity_load_steady_state*.

**`calibration_target_equations = [QUOTED_STRING | INTEGER[, QUOTED_STRING | INTEGER[, ...]]]`**

List of aggregate equations to use as calibration targets. Equations can be specified by their `name` tag (as a quoted string) or by their index (as an integer). When not specified, all aggregate equations containing a `SUM` operator are used automatically.

> **Warning**
>
> Using equation `name` tags is recommended over integer indices, as the latter are sensitive to equation reordering.

*Example*:

```
heterogeneity_compute_steady_state(filename=myfile,
    calibration_target_equations=['capital_market_clearing']);
```

**calibration_tolf = DOUBLE**

Convergence tolerance for the Broyden solver on aggregate residuals. Also used to check the residual on non-target aggregate equations. Default: `1e-4`

**calibration_max_iter = INTEGER**

Maximum number of Broyden iterations. Default: `50`

**calibration_verbosity = INTEGER**

Verbosity level for calibration output. `0`: silent; `1`: summary; `2`: iteration details. Default: `2`

**time_iteration_max_iter = INTEGER**

Maximum number of time-iteration (backward) steps per residual evaluation. Default: `1000`

**time_iteration_tol = DOUBLE**

Convergence tolerance on policy functions (sup-norm). Default: `1e-8`

**time_iteration_learning_rate = DOUBLE**

Dampening factor for policy function updates. A value of `1` means no dampening. Reducing this below `1` improves stability at the cost of slower convergence. Default: `1`

**time_iteration_early_stopping = INTEGER**

Number of consecutive iterations with increasing residuals before the time-iteration loop is stopped early (to avoid divergence). Set to `0` to disable. Default: `3`

**time_iteration_verbosity = INTEGER**

Verbosity level for time iteration. `0`: silent; `2`: iteration details. Default: `2`

**time_iteration_solver_tolf = DOUBLE**

Tolerance on the residual for the trust-region solver used at each grid point within time iteration. Default: `1e-10`

**time_iteration_solver_tolx = DOUBLE**

Tolerance on the step size for the trust-region solver. Default: `1e-10`

**time_iteration_solver_factor = DOUBLE**

Initial step bound factor for the trust-region solver. Default: `100`

**time_iteration_solver_max_iter = INTEGER**

Maximum iterations per grid point for the trust-region solver. Default: `1000`

**time_iteration_solver_stop_on_error**

If present, the solver stops immediately when the trust-region solver fails to converge at any grid point. Without this option, the solver continues with the best iterate.

**forward_max_iter = INTEGER**

Maximum number of forward iterations for the stationary distribution. Default: `10000`

**forward_tol = DOUBLE**

Convergence tolerance for the stationary distribution (L1 norm). Default: `1e-10`

**forward_check_every = INTEGER**

Check distribution convergence every `INTEGER` iterations. Default: `100`

**forward_verbosity = INTEGER**

Verbosity level for distribution computation. `0`: silent; `1`: non-convergence warnings only; `2`: iteration details. Default: `1`

*Output*

Same output as *heterogeneity_load_steady_state*. Additionally updates *M_.params* with calibrated parameter values.

**Practical advice**

The quality of the initial guess is critical. Poor starting values for policy functions or parameters may cause divergence or convergence to a spurious solution. In particular, numerical convergence does not ensure that the obtained solution is economically meaningful. The following strategies can help:

- **Grid homotopy:** Start with a coarse grid (e.g., 50 points for states), solve the steady state, then use the result as the initial guess for a finer grid (e.g., 200 or 500 points). Repeat until the desired resolution is reached.

  To save and reuse results across grid resolutions, add a `verbatim` block after the coarse-grid run. For example, in the Krusell and Smith (1998) model with a coarse 30-point asset grid:

```
verbatim;
% Save the converged steady state
my_ss = oo_.heterogeneity.steady_state;

% Store calibrated parameter values for later reuse
my_ss.free_parameters.beta.initial_guess = get_param_by_name('beta');

save('ks_coarse.mat', 'my_ss');
end;
```

  The saved `.mat` file can be reloaded as-is with *heterogeneity_load_steady_state*:

```
heterogeneity_load_steady_state(filename='ks_coarse.mat', variable=my_ss);
```

  To refine the grid and re-solve, load the coarse result, build a finer asset grid, interpolate the policy functions onto it, and pass the updated structure to *heterogeneity_compute_steady_state*. The calibrated parameter values and bounds saved above are carried over automatically. The following example illustrates with the Krusell and Smith (1998) model (state: `a`; shock: `e`; policies: `c`, `a`):

```
verbatim;
% Load the coarse-grid result (30 points)
load('ks_coarse.mat', 'my_ss');

% Build a finer asset grid (500 points, same bounds)
a_coarse = my_ss.pol.grids.a;
a_fine   = logspace(log10(a_coarse(1)+0.25), ...
                    log10(a_coarse(end)+0.25), 500) - 0.25;

% Interpolate policy functions from coarse to fine grid
% Each row of pol.values corresponds to one shock state
for ie = 1:size(my_ss.pol.values.c, 1)
    my_ss.pol.values.c(ie,:) = interp1(a_coarse, ...
        my_ss.pol.values.c(ie,:), a_fine, 'linear', 'extrap');
    my_ss.pol.values.a(ie,:) = interp1(a_coarse, ...
        my_ss.pol.values.a(ie,:), a_fine, 'linear', 'extrap');
end

% Update grids
my_ss.pol.grids.a = a_fine;
my_ss.d.grids.a   = a_fine;
end;

heterogeneity_compute_steady_state(variable=my_ss);
```

- **Asymmetric grids:** Using a coarse grid for policy functions (`pol.grids`) and a denser grid for the distribution (`d.grids`) is often an effective compromise between accuracy and speed.
- **Dampening:** If time iteration oscillates, reduce `time_iteration_learning_rate` (e.g., to `0.5` or `0.8`). This slows convergence but stabilizes updates.
- **Solver tolerances:** If the trust-region solver at individual grid points fails, try relaxing `time_iteration_solver_tolf` or increasing `time_iteration_solver_max_iter`. Use `time_iteration_solver_stop_on_error` during debugging to pinpoint problematic grid regions.
- **Calibration tolerance:** The outer Broyden loop tolerance `calibration_tolf` should be looser than the inner time-iteration tolerance `time_iteration_tol`, otherwise the calibration step may attempt to match noise in the inner solver.

*Examples*

Compute steady state without parameter calibration (residual check only):

```
heterogeneity_compute_steady_state(filename=ks_sp);
```

Compute steady state with parameter calibration and user-specified target equation:

```
heterogeneity_compute_steady_state(filename=hank_1a_sp,
    calibration_target_equations=['capital_market_clearing'],
    time_iteration_solver_stop_on_error);
```

Compute steady state with fine-tuned convergence settings:

```
heterogeneity_compute_steady_state(filename=hank_2a_sp,
    calibration_target_equations=['wage_nkpc',
        'liquid_asset_market_clearing',
        'illiquid_asset_market_clearing'],
    time_iteration_tol=1e-10,
    time_iteration_learning_rate=0.79,
    time_iteration_solver_tolf=1e-12,
    time_iteration_solver_tolx=1e-14);
```

#### 4.26.2.3 Helper functions

The following MATLAB/Octave functions are available for use in steady-state helper scripts.

**MATLAB/Octave command: `[y, p_vec, P_mat] = rouwenhorst(rho, sigma, N, tol, max_iter)`**

Constructs an `N`-point Markov chain approximation, using the Rouwenhorst method (Rouwenhorst, 1995), to a positive stationary process $y_t$ whose logarithm is an AR(1):

$$\log y_t = \mu + \rho \log y_{t-1} + \varepsilon_t,$$

where $\rho$ is the persistence parameter, $\sigma$ is the stationary standard deviation of $\log y_t$ (**not** of $\varepsilon_t$), and the constant $\mu$ is chosen so that $\mathbb{E}y_t = 1$.

The outputs can directly populate the `steady_state.shocks` structure required by *`heterogeneity_load_steady_state`* and *`heterogeneity_compute_steady_state`*: `y` provides `shocks.grids.VARNAME` and `P_mat` provides `shocks.Pi.VARNAME`.

*Inputs*

`rho` *(double)*

Persistence parameter of $\log y_t$. Must satisfy $0 \leq \rho < 1$.

`sigma` *(double)*

Unconditional standard deviation of $\log y_t$.

`N` *(integer)*

Number of grid points for the discretization.

`tol` *(double)*

Convergence tolerance for the stationary distribution computation.

`max_iter` *(integer)*

Maximum number of iterations for the stationary distribution computation.

*Outputs*

`y` *(N×1 vector)*

Grid points in levels. Internally, a symmetric log-grid on $[-1, 1]$ is scaled so that its variance matches `sigma`, then exponentiated and normalized so that $\sum_i \texttt{p_vec}(i)\,\texttt{y}(i) = 1$.

`p_vec` *(N×1 vector)*

Stationary distribution probabilities. Element $i$ gives the long-run fraction of time spent in state $i$.

`P_mat` *(N×N matrix)*

Transition probability matrix. Element $(i, j)$ gives the probability of moving from state $i$ to state $j$. Each row sums to 1.

*Example*

```
rho    = 0.966;
sigma = 0.5;
N      = 7;
tol    = 1e-10;
max_iter = 1000;
[y, p_vec, P_mat] = rouwenhorst(rho, sigma, N, tol, max_iter);
```

#### 4.26.2.4 Solving for the dynamics

**Command: heterogeneity_solve ;**

**Command: heterogeneity_solve(OPTIONS...);**

Computes the linearized solution. *heterogeneity_load_steady_state* or *heterogeneity_compute_steady_state* must be called first.

*Options*

**truncation_horizon = INTEGER**

Time horizon for Jacobian computations. Default: `300`

*Output*

Populates *oo_.heterogeneity.dr*.

**MATLAB/Octave variable: oo_.heterogeneity.dr**

Structure storing the linearized solution of the heterogeneous-agent block, computed by *heterogeneity_solve*. The user-facing subfield is `G` (see *oo_.heterogeneity.dr.G*); other subfields are for internal purpose.

**MATLAB/Octave variable: oo_.heterogeneity.dr.G**

Nested structure of sequence-space Jacobian matrices. `oo_.heterogeneity.dr.G.VARIABLE_NAME.SHOCK_NAME` is a $T \times T$ matrix (where $T$ is *truncation_horizon*) whose entry $(t, s)$ gives the linear response of `VARIABLE_NAME` in period $t$ to a unit deviation of `SHOCK_NAME` in period $s$, treated as known at time 0.

*Example*

`oo_.heterogeneity.dr.G.Y.Z` contains the response of output `Y` to the exogenous shock `Z`.

#### 4.26.2.5 Simulating

**Command:** **heterogeneity_simulate ;**

**Command:** **heterogeneity_simulate**(OPTIONS...) [VARIABLE_NAME...];

Computes linear impulse response functions (IRFs) or simulations for heterogeneous-agent models. The command automatically detects the appropriate simulation mode based on model contents and options. *heterogeneity_solve* must be called first.

**Simulation Modes**

Dynare distinguishes two simulation modes based on shock anticipation:

1. **Stochastic simulation:** Computes responses to unanticipated shocks drawn from distributions specified in the `shocks` block. Always produces impulse response functions (IRFs). When `periods > 0`, also produces stochastic simulation paths with random shocks drawn each period from a multivariate normal law with mean zero and variance-covariance matrix `M_.Sigma_e`.
2. **News shock sequence:** Triggered automatically if the model file includes a `shocks` block with `periods` and `values` keywords. Simulates the model's response to anticipated shocks known at date 0. Following *Auclert et al. (2021)*: households learn at t=0 about the sequence of future shocks. News shock mode is mutually exclusive with stochastic simulation options `irf`, `periods`, `irf_shocks`, and `relative_irf`.

*Options*

The following options are available. See *stoch_simul* for detailed descriptions.

**irf = INTEGER**

Horizon for impulse response functions. See *irf*. Cannot be used with news shock sequence mode. Default: `40`

**periods = INTEGER**

Number of periods for stochastic simulation paths. See *periods*. When `0` (default): only IRFs are computed. When >`0`: also computes stochastic simulation with random shocks. Cannot be used with news shock sequence mode.

**irf_shocks = ( SHOCK_NAME [[,] SHOCK_NAME ...] )**

List of exogenous shocks for which to compute IRFs. See *irf_shocks*. Cannot be used with news shock sequence mode. Default: all aggregate shocks

**relative_irf**

Requests IRFs as percentage deviations from steady state in response to a unit shock. See *relative_irf*. Cannot be used with news shock sequence mode.

**nograph**

Suppresses graph generation. See *nograph*.

**nodisplay**

Prevents display of graphs. See *nodisplay*.

**graph_format = FORMAT**

Specifies output format(s) for graphs. See *graph_format*.

**tex**

Requests generation of TeX files for including IRF figures in LaTeX documents. See *tex*.

**irf_plot_threshold = DOUBLE**

Threshold for plotting IRFs. See *irf_plot_threshold*. Cannot be used with news shock sequence mode.

**print**

Enables printing of results. See *print*.

**noprint**

Suppresses printing of results. See *noprint*.

*Output*

The simulation results are stored in the same format as standard Dynare simulations: impulse responses in *oo_.irfs*, simulated endogenous variables in *oo_.endo_simul*, and exogenous paths in *oo_.exo_simul*.

**Stochastic simulation:**

- *oo_.irfs*: impulse response vectors for each variable-shock pair `(var)_(shock)`
- *oo_.endo_simul*: simulated paths in levels (only when `periods > 0`)

**News shock sequence:**

- *oo_.endo_simul*: simulated paths in levels
- *oo_.exo_simul*: realized news shock paths

*Examples*:

```
% Impulse responses for all aggregate shocks
heterogeneity_simulate;

% IRFs only for aggregate TFP shock
heterogeneity_simulate(irf_shocks=(Z));

% Stochastic simulation over 1000 periods
heterogeneity_simulate(periods=1000);

% Deterministic sequence simulation (when declared in shocks block)
heterogeneity_simulate(periods=40);

% IRFs for specific variables only
heterogeneity_simulate Y K;
```

## 4.27 Semi-structural models

Dynare provides tools for semi-structural models, in the vein of the FRB/US model (see Brayton and Tinsley (1996)), where expectations are not necessarily model consistent but based on a VAR auxiliary model. In the following, it is assumed that each equation is written as `VARIABLE = EXPRESSION` or `T(VARIABLE) = EXPRESSION` where `T(VARIABLE)` stands for a transformation of an endogenous variable (`log` or `diff`). This representation, where each equation determines the endogenous variable on the LHS, can be exploited when simulating the model (see algorithms `12` and `14` in *solve_algo*) and is mandatory to define auxiliary models used for computing expectations (see below).

### 4.27.1 Auxiliary models

The two auxiliary models defined in this section are linear backward-looking models used to form expectations. Both models can be recast as VAR(1)-processes and therefore offer isomorphic ways of specifying the expectations process, but differ in their convenience of specifying features like cointegration and error correction. `var_model` directly specifies a VAR, while `trend_component_model` allows to define a trend target to which the endogenous variables may be attracted in the long-run (i.e. an error correction model).

**Command:** **`var_model`**`(OPTIONS...);`

Picks equations in the `model` block to form a VAR model. This model can be used as an auxiliary model in `var_expectation_model` or `pac_model`. It must be of the following form:

$$Y_t = \mathbf{c} + \sum_{i=1}^{p} A_i Y_{t-i} + \varepsilon_t$$

or

$$A_0 Y_t = \mathbf{c} + \sum_{i=1}^{p} A_i Y_{t-i} + \varepsilon_t$$

if the VAR is structural (see below), where $Y_t$ and $\varepsilon_t$ are $n \times 1$ vectors, $\mathbf{c}$ is a $n \times 1$ vector of parameters, $A_i$ $(i = 0, \ldots, p)$ are $n \times n$ matrices of parameters, and $A_0$ is non singular square matrix. Vector $\mathbf{c}$ and matrices $A_i$ $(i = 0, \ldots, p)$ are set by parsing the equations in the `model` block. Then, Dynare builds a VAR(1)-companion form model for $\mathcal{Y}_t = (1, Y_t, \ldots, Y_{t-p+1})'$ as:

$$\begin{pmatrix} 1 \\ Y_t \\ Y_{t-1} \\ \vdots \\ \vdots \\ Y_{t-p+1} \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0_n' & \ldots & \ldots & \ldots & 0_n' \\ \mathbf{c} & A_1 & A_2 & \ldots & \ldots & A_p \\ 0_n & I_n & O_n & \ldots & \ldots & O_n \\ 0_n & O_n & I_n & O_n & \ldots & O_n \\ \vdots & O_n & \ddots & \ddots & \ddots & \vdots \\ 0_n & O_n & \ldots & O_n & I_n & O_n \end{pmatrix}}_{\mathcal{C}} \begin{pmatrix} 1 \\ Y_{t-1} \\ Y_{t-2} \\ \vdots \\ \vdots \\ Y_{t-p} \end{pmatrix} + \underbrace{\begin{pmatrix} 0 \\ \varepsilon_t \\ 0_n \\ \vdots \\ \vdots \\ 0_n \end{pmatrix}}_{\epsilon_t}$$

assuming that we are dealing with a reduced-form VAR (otherwise, the right-hand side would additionally be premultiplied by $A_0^{-1}$. to obtain the reduced-form representation). If the VAR does not have a constant, we remove the first line of the system and the first column of the companion matrix $\mathcal{C}$. Dynare only saves the companion matrix, since that is the only information required to compute the expectations.

**MATLAB/Octave variable: oo_.var.MODEL_NAME.CompanionMatrix**

Reduced-form companion matrix of the `var_model`.

*Options*

**model_name = STRING**

Name of the VAR model, which will be referenced in `var_expectation_model` or `pac_model` as an `auxiliary_model`. Needs to be a valid MATLAB field name.

**eqtags = [QUOTED_STRING[, QUOTED_STRING[, ...]]]**

List of equations in the `model` block (referenced using the equation tag `name`) used to build the VAR model.

**structural**

By default the VAR model is not structural, *i.e.* each equation must contain exactly one contemporaneous variable (on the LHS). If the `structural` option is provided then any variable defined in the system can appear at time $t$ in each equation. Internally Dynare will rewrite this model as a reduced-form VAR (by inverting the implied matrix $A_0$).

*Example*

```
var_model(model_name = toto, eqtags = [ 'X', 'Y', 'Z' ]);

model;

[ name = 'X' ]
x = a*x(-1) + b*x(-2) + c*z(-2) + e_x;

[ name = 'Z' ]
z = f*z(-1) + e_z;
```

(continues on next page)

(continued from previous page)

```
[ name = 'Y' ]
y = d*y(-2) + e*z(-1) + e_y;

end;
```

**Command:** **trend_component_model**(OPTIONS...);

Picks equations in the model block to form a trend component model. This model can be used as an auxiliary model in `var_expectation_model` or `pac_model`. It must be of the following form:

$$\begin{cases} \Delta X_t &= A_0(X_{t-1} - C_0 Z_{t-1}) + \sum_{i=1}^{p} A_i \Delta X_{t-i} + \varepsilon_t \\ Z_t &= Z_{t-1} + \eta_t \end{cases}$$

where $X_t$ and $Z_t$ are $n \times 1$ and $m \times 1$ vectors of endogenous variables. $Z_t$ defines the trend target to whose linear combination $C_0 Z_t$ the endogenous variables $X_t$ will be attracted, provided the implied error correction matrix $A_0$ is negative definite. $\varepsilon_t$ and $\eta_t$ are $n \times 1$ and $m \times 1$ vectors of exogenous variables, $A_i$ ($i = 0, \ldots, p$) are $n \times n$ matrices of parameters, and $C_0$ is a $n \times m$ matrix. This model can also be cast into a VAR(1) model by first rewriting it in levels. Let $Y_t = (X_t', Z_t')'$ and $\zeta_t = (\varepsilon_t', \eta_t')'$. Then we have:

$$Y_t = \sum_{i=1}^{p+1} B_i Y_{t-i} + \zeta_t$$

with

$$B_1 = \begin{pmatrix} I_n + A_0 + A_1 & -\Lambda \\ O_{m,n} & I_m \end{pmatrix}$$

where $\Lambda = A_0 C_0$,

$$B_i = \begin{pmatrix} A_i - A_{i-1} & O_{n,m} \\ O_{m,n} & O_m \end{pmatrix}$$

for $i = 2, \ldots, p$, and

$$B_{p+1} = \begin{pmatrix} -A_p & O_{n,m} \\ O_{m,n} & O_m \end{pmatrix}$$

This VAR(p+1) in levels can again be rewritten as a VAR(1)-companion model form.

**MATLAB/Octave variable:** **oo_.trend_component.MODEL_NAME.CompanionMatrix**

Reduced-form companion matrix of the `trend_component_model`.

*Options*

**model_name = STRING**

Name of the trend component model, will be referenced in `var_expectation_model` or `pac_model` as an `auxiliary_model`. Needs to be a valid MATLAB field name.

**eqtags = [QUOTED_STRING[, QUOTED_STRING[, ...]]]**

List of equations in the `model` block (referenced using the equation tag `name`) used to build the trend component model.

`targets = [QUOTED_STRING[, QUOTED_STRING[, ...]]]`

List of targets, corresponding to the variables in vector $Z_t$, referenced using the equation tag `name`) of the associated equation in the `model` block. `target` must be a subset of `eqtags`.

*Example*

```
trend_component_model(model_name=toto, eqtags=['eq:x1', 'eq:x2',
↪'eq:x1bar', 'eq:x2bar'], targets=['eq:x1bar', 'eq:x2bar']);

model;

[name='eq:x1']
diff(x1) = a_x1_0*(x1(-1)-x1bar(-1))+a_x1_0_*(x2(-1)-x2bar(-1)) + a_x1_
↪1*diff(x1(-1)) + a_x1_2*diff(x1(-2)) + + a_x1_x2_1*diff(x2(-1)) + a_
↪x1_x2_2*diff(x2(-2)) + ex1;

[name='eq:x2']
diff(x2) = a_x2_0*(x2(-1)-x2bar(-1)) + a_x2_1*diff(x1(-1)) + a_x2_
↪2*diff(x1(-2)) + a_x2_x1_1*diff(x2(-1)) + a_x2_x1_2*diff(x2(-2)) +␣
↪ex2;

[name='eq:x1bar']
x1bar = x1bar(-1) + ex1bar;

[name='eq:x2bar']
x2bar = x2bar(-1) + ex2bar;

end;
```

### 4.27.2 VAR expectations

Suppose we wish to forecast a variable $y_t$ and that $y_t$ is an element of vector of variables $\mathcal{Y}_t$ whose law of motion is described by a VAR(1) model $\mathcal{Y}_t = \mathcal{C}\mathcal{Y}_{t-1} + \epsilon_t$. More generally, $y_t$ may be a linear combination of the scalar variables in $\mathcal{Y}_t$. Let the vector $\alpha$ be such that $y_t = \alpha'\mathcal{Y}_t$ ($\alpha$ is a selection vector if $y_t$ is a variable in $\mathcal{Y}_t$, *i.e.* a column of an identity matrix, or an arbitrary vector defining the weights of a linear combination). Then the best prediction, in the sense of the minimization of the RMSE, for $y_{t+h}$ given the information set at $t-\tau$ (which we assume to include all observables up to time $t-\tau$, $\mathcal{Y}_{\underline{t-\tau}}$) is:

$$y_{t+h|t-\tau} = \mathbb{E}[y_{t+h}|\mathcal{Y}_{\underline{t-\tau}}] = \alpha\mathcal{C}^{h+\tau}\mathcal{Y}_{t-\tau}$$

In a semi-structural model, variables appearing in $t+h$ (*e.g.* the expected output gap in a dynamic IS curve or expected inflation in a New Keynesian Phillips curve) will be replaced by the expectation implied by an auxiliary VAR model. Another use case is for the computation of permanent incomes. Typically, consumption will depend on something like:

$$\sum_{h=0}^{\infty}\beta^h y_{t+h|t-\tau}$$

Assuming that $0 < \beta < 1$ and knowing the limit of geometric series, the conditional expectation of this variable can be evaluated based on the same auxiliary model:

$$\mathbb{E}\left[\sum_{h=0}^{\infty}\beta^h y_{t+h}\middle|\mathcal{Y}_{\underline{t-\tau}}\right] = \alpha\mathcal{C}^{\tau}(I-\beta\mathcal{C})^{-1}\mathcal{Y}_{t-\tau}$$

Finite discounted sums can also be considered.

**Command:** **var_expectation_model**(OPTIONS...);

Declares a model used to forecast an endogenous variable or linear combination of variables in $t+h$. More generally, the same model can be used to forecast the discounted flow of a variable or a linear expression of variables:

$$\sum_{h=a}^{b} \beta^{h-\tau} \mathbb{E}[y_{t+h}|\mathcal{Y}_{\underline{t-\tau}}]$$

where $(a,b) \in \mathbb{N}^2$ with $a < b$ and $a < \infty$, $\beta \in (0,1]$ is a discount factor, and $\tau$ is a finite positive integer.

*Options*

**model_name = STRING**

Name of the VAR based expectation model, which will be referenced in the `model` block.

**auxiliary_model = STRING**

Name of the associated auxiliary model, defined with `var_model` or `trend_component_model`.

**expression = VARIABLE_NAME | EXPRESSION**

Name of the variable or expression (linear combination of variables) to be expected.

**discount = PARAMETER_NAME | DOUBLE**

Discount factor ($\beta$).

**horizon = INTEGER | [INTEGER:INTEGER]**

If the value of `horizon` is a finite integer scalar, the following expectation is computed:

$$\beta^{h-\tau} \mathbb{E}[y_{t+h}|\mathcal{Y}_{\underline{t-\tau}}]$$

otherwise the value is a range of periods $a:b$ over which the expected discounted sum is computed (the upper bound can be `Inf`).

**time_shift = INTEGER**

Shift of the information set ($\tau$), default value is 0.

**Operator:** **var_expectation (NAME_OF_VAR_EXPECTATION_MODEL);**

This operator is used instead of a leaded variable, e.g. `X(1)`, in the `model` block to substitute a model-consistent forecast with a forecast based on a VAR model.

*Example*

```
var_model(model_name=toto, eqtags=['X', 'Y', 'Z']);

var_expectation_model(model_name=varexp, expression=x, auxiliary_model_
↪name=toto, horizon=1, discount=beta);


model;

[name='X']
x = a*x(-1) + b*x(-2) + c*z(-2) + e_x;

[name='Z']
```

(continues on next page)

(continued from previous page)

```
z = f*z(-1) + e_z;

[name='Y']
y = d*y(-2) + e*z(-1) + e_y;

foo = .5*foo(-1) + var_expectation(varexp);

end;
```

In this example `var_expectation(varexp)` stands for the one step ahead expectation of `x`, as a replacement for `x(1)`.

**MATLAB/Octave command: `var_expectation.initialize(NAME_OF_VAR_EXPECTATION_MODEL);`**

Initialize the `var_expectation_model` by building the companion matrix of the associated auxiliary `var_model`. Needs to be executed before attempts to simulate or estimate the model.

**MATLAB/Octave command: `var_expectation.update(NAME_OF_VAR_EXPECTATION_MODEL);`**

Update/compute the reduced-form parameters of `var_expectation_model`. Needs to be executed before attempts to simulate or estimate the model and requires the auxiliary `var_model` to have previously been initialized.

*Example (continued)*

```
var_expectation.initialize('varexp');

var_expectation.update('varexp');
```

**Warning**

Changes to the parameters of the underlying auxiliary `var_model` require calls to `var_expectation.initialize` and `var_expectation.update` to become effective. Changes to the `var_expectation_model` or its associated parameters require a call to `var_expectation.update`.

### 4.27.3 PAC equation

In its simplest form, a PAC equation breaks down changes in a variable of interest $y$ into three contributions: (*i*) the lagged deviation from a target $y^\star$, (*ii*) the lagged changes in the variable $y$, and (*iii*) the expected changes in the target $y^\star$:

$$\Delta y_t = a_0(y^\star_{t-1} - y_{t-1}) + \sum_{i=1}^{m-1} a_i \Delta y_{t-i} + \sum_{i=0}^{\infty} d_i \Delta y^\star_{t+i} + \varepsilon_t$$

Brayton *et al.* (2000) shows how such an equation can be derived from the minimization of a quadratic cost function penalizing expected deviations from the target and non-smoothness of $y$, where future costs are discounted (with discount factor $\beta$). They also show that the parameters $(d_i)_{i\in\mathbb{N}}$ are non-linear functions of the $m$ parameters $a_i$ and the discount factor $\beta$. To simulate or estimate this equation we need to figure out how to determine the expected changes of the target. This can be done as in the previous section using VAR based expectations, or considering model consistent expectations (MCE).

To ensure that the endogenous variable $y$ is equal to its target $y^\star$ in the (deterministic) long run, *i.e.* that the error correction term is zero in the long run, we can optionally add a growth neutrality correction to this equation. Suppose that $g$ is the long run growth rate, for $y$ and $y^\star$, then in the long run (assuming that the data are in logs) we must have:

$$g = a_0(y^\star_\infty - y_\infty) + g\sum_{i=1}^{m-1} a_i + g\sum_{i=0}^{\infty} d_i$$
$$\Leftrightarrow a_0(y^\star_\infty - y_\infty) = \left(1 - \sum_{i=1}^{m-1} a_i - \sum_{i=0}^{\infty} d_i\right) g$$

Unless additional restrictions are placed on the coefficients $(a_i)_{i=0}^{m-1}$, i.e. on the form of the minimized cost function, there is no reason for the right-hand side to be zero. Instead, we can optionally add the right-hand side to the PAC equation, to ensure that the error correction term is asymptotically zero.

The PAC equations can be generalized by adding exogenous variables. This can be done in two, non-exclusive, manners. We can replace the PAC equation by a convex combination of the original PAC equation (derived from an optimization program) and a linear expression involving exogenous variables (referred as the rule of thumb part as opposed to the part derived from the minimization of a cost function; not to be confused with exogenous shocks):

$$\Delta y_t = \lambda\left(a_0(y^\star_{t-1} - y_{t-1}) + \sum_{i=1}^{m-1} a_i \Delta y_{t-i} + \sum_{i=0}^{\infty} d_i \Delta y^\star_{t+i}\right) + (1-\lambda)\gamma' X_t + \varepsilon_t$$

where $\lambda \in [0,1]$ is the weight of the pure PAC equation, $\gamma$ is a $k \times 1$ vector of parameters, and $X_t$ a $k \times 1$ vector of variables in the rule of thumb part. Or we can simply add the exogenous variables to the PAC equation (without the weight $\lambda$):

$$\Delta y_t = a_0(y^\star_{t-1} - y_{t-1}) + \sum_{i=1}^{m-1} a_i \Delta y_{t-i} + \sum_{i=0}^{\infty} d_i \Delta y^\star_{t+i} + \gamma' X_t + \varepsilon_t$$

**Command:** **pac_model**(OPTIONS...);

Declares a PAC model. A `.mod` file can have more than one PAC model or PAC equation, but each PAC equation must be associated to a different PAC model.

*Options*

**model_name = STRING**

Name of the PAC model, will be referenced in the `model` block.

**auxiliary_model = STRING**

Name of the associated auxiliary model, defined with `var_model` or `trend_component_model`, to compute the VAR based expectations for the expected changes in the target, *i.e.* to evaluate $\sum_{i=0}^{\infty} d_i \Delta y^\star_{t+i}$. The infinite sum will then be replaced by a linear combination, defined by a vector $h$, of the variables involved in the companion representation of the auxiliary model. The weights defining the linear combination are nonlinear functions of the $(a_i)_{i=0}^{m-1}$ coefficients in the PAC equation. This option is not mandatory, if absent Dynare understands that the expected changes of the target have to be computed under the MCE assumption. This is done by rewriting recursively the infinite sum as shown in equation 10 of Brayton *et al.* (2000).

**discount = PARAMETER_NAME | DOUBLE**

Discount factor ($\beta$) for future expected costs appearing in the definition of the cost function.

**growth = PARAMETER_NAME | VARIABLE_NAME | EXPRESSION | DOUBLE**

If present a growth neutrality correction is added to the PAC equation. The user must ensure that the provided value (or long term level if a variable or expression is given) is consistent with the asymptotic growth rate of the endogenous variable.

**kind = dd | dl**

Instructs Dynare how to compute the vector $h$, the weights defining the linear combination of the companion VAR variables. The default value dd must be used if the target appears in first difference in the auxiliary model, see equation (A.79) in Brayton *et al.* (2000), while value dl must be used if the target shows up in level in the auxiliary model, equation (A.74) in Brayton *et al.* (2000).

**auxname = STRING**

Name the auxiliary variable, created by the preprocessor, that will define the expectation term in the PAC equation.

**Operator: pac_expectation (NAME_OF_PAC_MODEL);**

This operator is used instead of the infinite sum, $\sum_{i=0}^{\infty} d_i \Delta y^{\star}_{t+i}$, in a PAC equation defined in the model block. Depending on the assumption regarding the formation of expectations, it will be replaced by a linear combination of the variables involved in the companion representation of the auxiliary model or by a recursive forward equation.

The PAC equation target can be composite and defined as a weighted sum of stationary and non-stationary components. Such a target requires an additional equation in the model block, with the target variable on the left hand-side and the components in the right hand-side. Each component must be an endogenous variable in the auxiliary model. The characteristics of each component must be described in the pac_target_info block, see below, and the pac_target_nonstationary operator must be used in the error correction term of the PAC equation to link the target to the provided description. Note that composite targets make only sense if the auxiliary model is not a trend component model (where all the variables are non-stationary).

**Block: pac_target_info(NAME_OF_PAC_MODEL);**

This block enables the user to provide the properties of each component of a target in PAC models with a composite target. The NAME_OF_PAC_MODEL argument refers to a PAC model (must match the value of option model_name in the declaration of a PAC model).

On the first line of the block, the name of the composite target variable must be provided using the following syntax:

```
target VARIABLE_NAME ;
```

where VARIABLE_NAME is a declared endogenous variable, its associated equation is not part of the auxiliary model but all the components (the variables on the right hand-side) must be defined in the auxiliary model. Next, the following line declares the name of the auxiliary variable that will appear in the error correction term, this variable contains only the non-stationary components of the target:

```
auxname_target_nonstationary NAME ;
```

The block should contain the following group of lines for each stationary component:

```
component STATIONARY_VARIABLE_NAME ;
kind ll ;
auxname AUX_VAR_NAME ;
```

where STATIONARY_VARIABLE_NAME is the name of a stationary variable appearing in the right hand-side of the equation defining the target VARIABLE_NAME. The second line instructs Dynare that the component appears in levels in the auxiliary model and in the PAC expectations. The third line specifies the name of the auxiliary variable created by Dynare for the component of the PAC expectation related to STATIONARY_VARIABLE_NAME.

The block should contain the following group of lines for each nonstationary component:

```
component NONSTATIONARY_VARIABLE_NAME ;
kind dd | dl ;
auxname AUX_VAR_NAME ;
growth PARAMETER_NAME | VARIABLE_NAME | EXPRESSION | DOUBLE ;
```

where NONSTATIONARY_VARIABLE_NAME is the name of a nonstationary variable appearing in the right hand-side of the equation defining the target VARIABLE_NAME. The second line instructs Dynare on how to calculate the weights that define the linear combination of the companion VAR variables. Use value dd if the target appears in first difference in the auxiliary model, or dl if the target shows up in level in the auxiliary model. The third line sets the name of the auxiliary variable created by Dynare for the component of the PAC expectation related to NONSTATIONARY_VARIABLE_NAME. The fourth line is mandatory if a growth neutrality correction is required. The provided value for this option must be consistent with the asymptotic growth rate of the PAC endogenous variable.

**Operator: pac_target_nonstationary (NAME_OF_PAC_MODEL);**

This operator is only required in presence of a composite target in the PAC equation. The operator, used in the error correction term of the PAC equation, selects the non-stationary components of the target.

**MATLAB/Octave command: pac.initialize(NAME_OF_PAC_MODEL);**

**MATLAB/Octave command: pac.update(NAME_OF_PAC_MODEL);**

Same as in the previous section for the VAR expectations, initialize the PAC model, by building the companion matrix of the auxiliary model, and computes the reduced-form parameters of the PAC equation (the weights in the linear combination of the variables involved in the companion representation of the auxiliary model, or the parameters of the recursive representation of the infinite sum in the MCE case).

*Example (trend component auxiliary model)*

```
trend_component_model(model_name=toto, eqtags=['eq:x1', 'eq:x2', 'eq:x1bar',
↪ 'eq:x2bar'], targets=['eq:x1bar', 'eq:x2bar']);

pac_model(auxiliary_model_name=toto, discount=beta, model_name=pacman);

model;

  [name='eq:y']
  y = (1-rho_1-rho_2)*diff(x2(-1)) + rho_1*y(-1) + rho_2*y(-2) + ey;

  [name='eq:x1']
  diff(x1) = a_x1_0*(x1(-1)-x1bar(-1)) + a_x1_1*diff(x1(-1)) + a_x1_
↪2*diff(x1(-2)) + a_x1_x2_1*diff(x2(-1)) + a_x1_x2_2*diff(x2(-2)) + ex1;

  [name='eq:x2']
  diff(x2) = a_x2_0*(x2(-1)-x2bar(-1)) + a_x2_1*diff(x1(-1)) + a_x2_
↪2*diff(x1(-2)) + a_x2_x1_1*diff(x2(-1)) + a_x2_x1_2*diff(x2(-2)) + ex2;

  [name='eq:x1bar']
  x1bar = x1bar(-1) + ex1bar;

  [name='eq:x2bar']
  x2bar = x2bar(-1) + ex2bar;

  [name='zpac']
  diff(z) = e_c_m*(x1(-1)-z(-1)) + c_z_1*diff(z(-1))  + c_z_2*diff(z(-2)) +␣
↪pac_expectation(pacman) + ez;

end;

pac.initialize('pacman');

pac.update.expectation('pacman');
```

*Example (VAR auxiliary model and composite target)*

```
var_model(model_name=toto, eqtags=['eq:x', 'eq:y']);

pac_model(auxiliary_model_name=toto, discount=beta, model_name=pacman);

pac_target_info(pacman);

  target v;
  auxname_target_nonstationary vns;

  component y;
  auxname pv_y_;
  kind ll;

  component x;
  growth diff(x(-1));
  auxname pv_dx_;
  kind dd;

end;

model;

  [name='eq:y']
  y = a_y_1*y(-1) + a_y_2*diff(x(-1)) + b_y_1*y(-2) + b_y_2*diff(x(-2)) +␣
↪ey ;


  [name='eq:x']
  diff(x) = b_x_1*y(-2) + b_x_2*diff(x(-1)) + ex ;

  [name='eq:v']
  v = x + d_y*y ;  // Composite PAC target, no residuals here only␣
↪variables defined in the auxiliary model.

  [name='zpac']
  diff(z) = e_c_m*(pac_target_nonstationary(pacman)-z(-1)) + c_z_1*diff(z(-
↪1))  + c_z_2*diff(z(-2)) + pac_expectation(pacman) + ez;

end;

pac.initialize('pacman');

pac.update.expectation('pacman');
```

### 4.27.4 Estimation of a PAC equation

The PAC equation, introduced in the previous section, can be estimated. This equation is nonlinear with respect to the estimated parameters $(a_i)_{i=0}^{m-1}$, since the reduced-form parameters (in the computation of the infinite sum) are nonlinear functions of the autoregressive parameters and the error correction parameter. Brayton *et al.* (2000) shows how to estimate the PAC equation by iterative OLS. Although this approach is implemented in Dynare, mainly for comparison purposes, we also propose NLS estimation, which is much preferable (asymptotic properties of NLS being more solidly grounded).

Note that it is currently not feasible to estimate the PAC equation jointly with the remaining parameters of the model using e.g. Bayesian techniques. Thus, estimation of the PAC equation can only be conducted conditional on the values of the parameters of the auxiliary model.

**Warning**

The estimation routines described below require the option `json=compute` be passed to the preprocessor (via the command line or at the top of the `.mod` file, see *Dynare invocation*).

**MATLAB/Octave command:** `pac.estimate.nls(EQNAME, GUESS, DATA, RANGE[, ALGO]);`

**MATLAB/Octave command:** `pac.estimate.iterative_ols(EQNAME, GUESS, DATA, RANGE);`

Trigger the NLS or iterative OLS estimation of a PAC equation. `EQNAME` is a row char array designating the PAC equation to be estimated (the PAC equation must have a name specified with an equation tag). `DATA` is a `dseries` object containing the data required for the estimation (*i.e.* data for all the endogenous and exogenous variables in the equation). The residual values of the PAC equation (which correspond to a defined *varexo*) must also be a member of `DATA`, but filled with `NaN` values. `RANGE` is a `dates` object defining the time span of the sample. `ALGO` is a row char array used to select the method (or minimization algorithm) for NLS. Possible values are : `'fmincon'`, `'fminunc'`, `'fminsearch'`, `'lsqnonlin'`, `'particleswarm'`, `'csminwel'`, `'simplex'`, `'annealing'`, and `'GaussNewton'`. The first four algorithms require the Mathworks Optimisation toolbox. The fifth algorithm requires the Mathworks Global Optimisation toolbox. When the optimisation algorithm allows it, we impose constraints on the error correction parameter, which must be positive and smaller than 1 (it the case for `'fmincon'`, `'lsqnonlin'`, `'particleswarm'`, and `'annealing'`). The default optimisation algorithm is `'csminwel'`. `GUESS` is a structure containing the initial guess values for the estimated parameters. Each field is the name of a parameter in the PAC equation and holds the initial guess for this parameter. If some parameters are calibrated, then they should not be members of the `GUESS` structure (and values have to be provided in the `.mod` file before the call to the estimation routine).

For the NLS routine the estimation results are displayed in a table after the estimation. For both the NLS and iterative OLS routines, the results are saved in `oo_` (under the fields `nls` or `iterative_ols`). Also, the values of the parameters are updated in `M_.params`.

*Example (continued)*

```
// Set the initial guess for the estimated parameters
eparams.e_c_m =  .9;
eparams.c_z_1 =  .5;
eparams.c_z_2 =  .2;

// Define the dataset used for estimation
edata = TrueData;
edata.ez = dseries(NaN); // Set to NaN the residual of the equation.

pac.estimate.nls('zpac', eparams, edata, 2005Q1:2005Q1+200, 'annealing');
```

**Warning**

The specification of *GUESS* and *DATA* involves the use of structures. As such, their subfields will not be cleared across Dynare runs as the structures stay in the workspace. Be careful to clear these structures from the memory (e.g. within the `.mod` file) when e.g. changing which parameters are calibrated.

## 4.28 Displaying and saving results

Dynare has comments to plot the results of a simulation and to save the results.

**Command:** `rplot VARIABLE_NAME...;`

Plots the simulated path of one or several variables, as stored in `oo_.endo_simul` by either *perfect_foresight_solver*, `simul` (see *Deterministic simulation*) or *stoch_simul* with option `periods` (see *Stochastic solution and simulation*). The variables are plotted in levels.

**Command:** `dynatype(FILENAME) [VARIABLE_NAME...];`

This command prints the listed endogenous or exogenous variables in a text file named FILENAME. If no VARIABLE_NAME is listed, all endogenous variables are printed.

**Command:** `dynasave(FILENAME) [VARIABLE_NAME...];`

This command saves the listed endogenous or exogenous variables in a binary file named FILENAME. If no VARIABLE_NAME is listed, all endogenous variables are saved.

In MATLAB or Octave, variables saved with the `dynasave` command can be retrieved by the command:

```
load(FILENAME,'-mat')
```

## 4.29 Macro processing language

It is possible to use "macro" commands in the `.mod` file for performing tasks such as: including modular source files, replicating blocks of equations through loops, conditionally executing some code, writing indexed sums or products inside equations...

The Dynare macro-language provides a new set of *macro-commands* which can be used in `.mod` files. It features:

- File inclusion
- Loops (`for` structure)
- Conditional inclusion (`if/then/else` structures)
- Expression substitution

This macro-language is totally independent of the basic Dynare language, and is processed by a separate component of the Dynare pre-processor. The macro processor transforms a `.mod` file with macros into a `.mod` file without macros (doing expansions/inclusions), and then feeds it to the Dynare parser. The key point to understand is that the macro processor only does text substitution (like the C preprocessor or the PHP language). Note that it is possible to see the output of the macro processor by using the `savemacro` option of the `dynare` command (see *Dynare invocation*).

The macro processor is invoked by placing *macro directives* in the `.mod` file. Directives begin with an at-sign followed by a pound sign (`@#`). They produce no output, but give instructions to the macro processor. In most cases, directives occupy exactly one line of text. If needed, two backslashes (\\) at the end of the line indicate that the directive is continued on the next line. Macro directives following `//` are not interpreted by the macro processor. For historical reasons, directives in commented blocks, *i.e.* surrounded by `/*` and `*/`, are interpreted by the macro processor. The user should not rely on this behavior. The main directives are:

- `@#includepath`, paths to search for files that are to be included,
- `@#include`, for file inclusion,
- `@#define`, for defining a macro processor variable,
- `@#if`, `@#ifdef`, `@#ifndef`, `@#elseif`, `@#else`, `@#endif` for conditional statements,
- `@#for`, `@#endfor` for constructing loops.

The macro processor maintains its own list of variables (distinct from model variables and MATLAB/Octave variables). These macro-variables are assigned using the `@#define` directive and can be of the following basic types: boolean, real, string, tuple, function, and array (of the previous types).

### 4.29.1 Macro expressions

Macro-expressions can be used in two places:

- Inside macro directives, directly;
- In the body of the `.mod` file, between an @-sign and curly braces (like `@{expr}`): the macro processor will substitute the expression with its value

It is possible to construct macro-expressions that can be assigned to macro-variables or used within a macro-directive. The expressions are constructed using literals (*i.e.*fixed values) of the basic types (boolean, real, string, tuple, array), comprehensions, macro-variables, macro-functions, and standard operators.

**Note**

Elsewhere in the manual, MACRO_EXPRESSION designates an expression constructed as explained in this section.

### Boolean

The following operators can be used on Booleans:

- Comparison operators: ==, !=
- Logical operators: &&, ||, !

### Real

The following operators can be used on reals:

- Arithmetic operators: +, -, *, /, ^
- Comparison operators: <, >, <=, >=, ==, !=
- Logical operators: &&, ||, !
- Ranges with an increment of 1: `REAL1:REAL2` (for example, `1:4` is equivalent to real array `[1, 2, 3, 4]`).

  Changed in version 4.6: Previously, putting brackets around the arguments to the colon operator (e.g. `[1:4]`) had no effect. Now, `[1:4]` will create an array containing an array (i.e. `[ [1, 2, 3, 4] ]`).
- Ranges with user-defined increment: `REAL1:REAL2:REAL3` (for example, `6:-2.1:-1` is equivalent to real array `[6, 3.9, 1.8, -0.3]`).
- Functions: `max`, `min`, `mod`, `exp`, `log`, `log10`, `sin`, `cos`, `tan`, `asin`, `acos`, `atan`, `sqrt`, `cbrt`, `sign`, `floor`, `ceil`, `trunc`, `erf`, `erfc`, `gamma`, `lgamma`, `round`, `normpdf`, `normcdf`. NB `ln` can be used instead of `log`

### String

String literals have to be enclosed by **double** quotes (like `"name"`).

The following operators can be used on strings:

- Comparison operators: <, >, <=, >=, ==, !=
- Concatenation of two strings: +
- Extraction of substrings: if `s` is a string, then `s[3]` is a string containing only the third character of `s`, and `s[4:6]` contains the characters from 4th to 6th
- Function: `length`

### Tuple

Tuples are enclosed by parentheses and elements are separated by commas (like `(a,b,c)` or `(1,2,3)`).

The following operators can be used on tuples:

- Comparison operators: ==, !=
- Functions: `empty`, `length`

### Array

Arrays are enclosed by brackets, and their elements are separated by commas (like `[1,[2,3],4]` or `["US", "FR"]`).

The following operators can be used on arrays:

- Comparison operators: `==`, `!=`
- Dereferencing: if `v` is an array, then `v[2]` is its 2nd element
- Concatenation of two arrays: `+`
- Set union of two arrays: `|`
- Set intersection of two arrays: `&`
- Difference `-`: returns the first operand from which the elements of the second operand have been removed.
- Cartesian product of two arrays: `*`
- Cartesian product of one array N times: `^N`
- Extraction of sub-arrays: e.g. `v[4:6]`
- Testing membership of an array: `in` operator (for example: `"b"` in `["a", "b", "c"]` returns 1)
- Functions: `empty`, `sum`, `length`

### Comprehension

Comprehension syntax is a shorthand way to make arrays from other arrays. There are three different ways the comprehension syntax can be employed: *filtering*, *mapping*, and *filtering and mapping*.

#### Filtering

Filtering allows one to choose those elements from an array for which a certain condition hold.

*Example*

Create a new array, choosing the even numbers from the array `1:5`:

```
[ i in 1:5 when mod(i,2) == 0 ]
```

would result in:

```
[2, 4]
```

#### Mapping

Mapping allows you to apply a transformation to every element of an array.

*Example*

Create a new array, squaring all elements of the array `1:5`:

```
[ i^2 for i in 1:5 ]
```

would result in:

```
[1, 4, 9, 16, 25]
```

#### Filtering and Mapping

Combining the two preceding ideas would allow one to apply a transformation to every selected element of an array.

*Example*

Create a new array, squaring all even elements of the array `1:5`:

```
[ i^2 for i in 1:5 when mod(i,2) == 0]
```

would result in:

```
[4, 16]
```

*Further Examples*

```
[ (j, i+1) for (i,j) in (1:2)^2 ]
[ (j, i+1) for (i,j) in (1:2)*(1:2) when i < j ]
```

would result in:

```
[(1, 2), (2, 2), (1, 3), (2, 3)]
[(2, 2)]
```

### Function

Functions can be defined in the macro processor using the `@#define` directive (see below). A function is evaluated at the time it is invoked during the macroprocessing stage, not at define time. Functions can be included in expressions and the operators that can be combined with them depend on their return type.

### Checking variable type

Given a variable name or literal, you can check the type it evaluates to using the following functions: `isboolean`, `isreal`, `isstring`, `istuple`, and `isarray`.

*Examples*

| Code | Output |
|---|---|
| `isboolean(0)` | `false` |
| `isboolean(true)` | `true` |
| `isreal("str")` | `false` |

### Casting between types

Variables and literals of one type can be cast into another type. Some type changes are straightforward (e.g. changing a *real* to a *string*) whereas others have certain requirements (e.g. to cast an *array* to a *real* it must be a one element array containing a type that can be cast to *real*).

*Examples*

| Code | Output |
|---|---|
| `(bool) -1.1` | `true` |
| `(bool) 0` | `false` |
| `(real) "2.2"` | `2.2` |
| `(tuple) [3.3]` | `(3.3)` |
| `(array) 4.4` | `[4.4]` |
| `(real) [5.5]` | `5.5` |
| `(real) [6.6, 7.7]` | `error` |
| `(real) "8.8 in a string"` | `error` |

Casts can be used in expressions:

*Examples*

| Code | Output |
|---|---|
| (bool) 0 && true | false |
| (real) "1" + 2 | 3 |
| (string) (3 + 4) | "7" |
| (array) 5 + (array) 6 | [5, 6] |

### 4.29.2 Macro directives

**Macro directive:** **@#includepath** "PATH"

**Macro directive:** **@#includepath** MACRO_EXPRESSION

This directive adds the path contained in PATH to the list of those to search when looking for a `.mod` file specified by `@#include`. If provided with a MACRO_EXPRESSION argument, the argument must evaluate to a string. Note that these paths are added *after* any paths passed using *-I*.

*Example*

```
@#includepath "/path/to/folder/containing/modfiles"
@#includepath folders_containing_mod_files
```

**Macro directive:** **@#include** "FILENAME"

**Macro directive:** **@#include** MACRO_EXPRESSION

This directive simply includes the content of another file in its place; it is exactly equivalent to a copy/paste of the content of the included file. If provided with a MACRO_EXPRESSION argument, the argument must evaluate to a string. Note that it is possible to nest includes (i.e. to include a file from an included file). The file will be searched for in the current directory. If it is not found, the file will be searched for in the folders provided by *-I* and `@#includepath`.

*Example*

```
@#include "modelcomponent.mod"
@#include location_of_modfile
```

**Macro directive:** **@#define** MACRO_VARIABLE

**Macro directive:** **@#define** MACRO_VARIABLE = MACRO_EXPRESSION

**Macro directive:** **@#define** MACRO_FUNCTION = MACRO_EXPRESSION

Defines a macro-variable or macro function.

*Example*

```
@#define var                     // Equals true
@#define x = 5                   // Real
@#define y = "US"                // String
@#define v = [ 1, 2, 4 ]         // Real array
@#define w = [ "US", "EA" ]      // String array
@#define u = [ 1, ["EA"] ]       // Mixed array
@#define z = 3 + v[2]            // Equals 5
@#define t = ("US" in w)         // Equals true
@#define f(x) = " " + x + y      // Function `f` with argument `x`
                                // returns the string ' ' + x + 'US'
```

*Example*

```
@#define x = 1
@#define y = [ "B", "C" ]
@#define i = 2
@#define f(x) = x + " " + y[i]
```

(continues on next page)

(continued from previous page)

```
@#define i = 1

model;
  A = @{y[i] + f("D")};
end;
```

The latter is strictly equivalent to:

```
model;
  A = BD + B;
end;
```

**Macro directive:** **@#if** MACRO_EXPRESSION
**Macro directive:** **@#ifdef** MACRO_VARIABLE
**Macro directive:** **@#ifndef** MACRO_VARIABLE
**Macro directive:** **@#elseif** MACRO_EXPRESSION
**Macro directive:** **@#else**
**Macro directive:** **@#endif**

Conditional inclusion of some part of the `.mod` file. The lines between `@#if`, `@#ifdef`, or `@#ifndef` and the next `@#elseif`, `@#else` or `@#endif` is executed only if the condition evaluates to `true`. Following the `@#if` body, zero or more `@#elseif` branches are allowed. An `@#elseif` condition is only evaluated if the preceding `@#if` or `@#elseif` condition(s) evaluated to `false`. The `@#else` branch is optional and only evaluated if all `@#if` and `@#elseif` statements evaluate to false.

Note that when using `@#ifdef`, the condition will evaluate to `true` if the MACRO_VARIABLE has been previously defined, regardless of its value. Conversely, `@#ifndef` will evaluate to true if the MACRO_VARIABLE has not yet been defined.

Note that when using `@#elseif` you can check whether or not a variable has been defined by using the `defined` operator. Hence, to enter the body of an `@#elseif` branch if the variable `X` has not been defined, you would write: `@#elseif !defined(X)`.

Note that if a real appears as the result of the MACRO_EXPRESSION, it will be interpreted as a boolean; a value of `0` is interpreted as `false`, otherwise it is interpreted as `true`. Further note that because of the imprecision of reals, extra care must be taken when testing them in the MACRO_EXPRESSION. For example, `exp(log(5)) == 5` will evaluate to `false`. Hence, when comparing real values, you should generally use a non-zero tolerance around the value desired, e.g. `exp(log(5)) > 5-1e-14 && exp(log(5)) < 5+1e-14`

*Example*

Choose between two alternative monetary policy rules using a macro-variable:

```
@#define linear_mon_pol = false // 0 would be treated the same
...
model;
@#if linear_mon_pol
  i = w*i(-1) + (1-w)*i_ss + w2*(pie-piestar);
@#else
  i = i(-1)^w * i_ss^(1-w) * (pie/piestar)^w2;
@#endif
...
end;
```

This would result in:

```
...
model;
```

(continues on next page)

(continued from previous page)

```
  i = i(-1)^w * i_ss^(1-w) * (pie/piestar)^w2;
...
end;
```

*Example*

Choose between two alternative monetary policy rules using a macro-variable. The only difference between this example and the previous one is the use of `@#ifdef` instead of `@#if`:

```
@#define linear_mon_pol = false // 0 would be treated the same
...
model;
@#ifdef linear_mon_pol
  i = w*i(-1) + (1-w)*i_ss + w2*(pie-piestar);
@#else
  i = i(-1)^w * i_ss^(1-w) * (pie/piestar)^w2;
@#endif
...
end;
```

Although `linear_mon_pol` contains the value `false` because `@#ifdef` only checks that the variable has been defined, the linear monetary policy is output::This would result in:

```
...
model;
  i = w*i(-1) + (1-w)*i_ss + w2*(pie-piestar);
...
end;
```

**Macro directive:** `@#for` MACRO_VARIABLE in MACRO_EXPRESSION

**Macro directive:** `@#for` MACRO_VARIABLE in MACRO_EXPRESSION when MACRO_EXPRESSION

**Macro directive:** `@#for` MACRO_TUPLE in MACRO_EXPRESSION

**Macro directive:** `@#for` MACRO_TUPLE in MACRO_EXPRESSION when MACRO_EXPRESSION

**Macro directive:** `@#endfor`

Loop construction for replicating portions of the `.mod` file. Note that this construct can enclose variable/parameter declarations, computational tasks, but not a model declaration.

*Example*

```
model;
@#for country in [ "home", "foreign" ]
  GDP_@{country} = A * K_@{country}^a * L_@{country}^(1-a);
@#endfor
end;
```

The latter is equivalent to:

```
model;
  GDP_home = A * K_home^a * L_home^(1-a);
  GDP_foreign = A * K_foreign^a * L_foreign^(1-a);
end;
```

*Example*

```
model;
@#for (i, j) in ["GDP"] * ["home", "foreign"]
  @{i}_@{j} = A * K_@{j}^a * L_@{j}^(1-a);
```

(continues on next page)

(continued from previous page)

```
@#endfor
end;
```

The latter is equivalent to:

```
model;
  GDP_home = A * K_home^a * L_home^(1-a);
  GDP_foreign = A * K_foreign^a * L_foreign^(1-a);
end;
```

*Example*

```
@#define countries = ["US", "FR", "JA"]
@#define nth_co = "US"
model;
@#for co in countries when co != nth_co
  (1+i_@{co}) = (1+i_@{nth_co}) * E_@{co}(+1) / E_@{co};
@#endfor
  E_@{nth_co} = 1;
end;
```

The latter is equivalent to:

```
model;
  (1+i_FR) = (1+i_US) * E_FR(+1) / E_FR;
  (1+i_JA) = (1+i_US) * E_JA(+1) / E_JA;
  E_US = 1;
end;
```

**Macro directive:** **@#echo** MACRO_EXPRESSION

Asks the preprocessor to display some message on standard output. The argument must evaluate to a string.

**Macro directive:** **@#error** MACRO_EXPRESSION

Asks the preprocessor to display some error message on standard output and to abort. The argument must evaluate to a string.

**Macro directive:** **@#echomacrovars**

**Macro directive:** **@#echomacrovars** MACRO_VARIABLE_LIST

**Macro directive:** **@#echomacrovars**(save) MACRO_VARIABLE_LIST

Asks the preprocessor to display the value of all macro variables up until this point. If the `save` option is passed, then values of the macro variables are saved to `options_.macrovars_line_<<line_numbers>>`. If `NAME_LIST` is passed, only display/save variables and functions with that name.

*Example*

```
@#define A = 1
@#define B = 2
@#define C(x) = x*2
@#echomacrovars A C D
```

The output of the command above is:

```
Macro Variables:
  A = 1
Macro Functions:
  C(x) = (x * 2)
```

### 4.29.3 Typical usages

#### 4.29.3.1 Modularization

The `@#include` directive can be used to split `.mod` files into several modular components.

Example setup:

`modeldesc.mod`

Contains variable declarations, model equations, and shocks declarations.

`simul.mod`

Includes `modeldesc.mod`, calibrates parameter,s and runs stochastic simulations.

`estim.mod`

Includes `modeldesc.mod`, declares priors on parameters, and runs Bayesian estimation.

Dynare can be called on `simul.mod` and `estim.mod`, but it makes no sense to run it on `modeldesc.mod`.

The main advantage is that you don't have to copy/paste the whole model (during initial development) or changes to the model (during development).

#### 4.29.3.2 Indexed sums of products

The following example shows how to construct a moving average:

```
@#define window = 2

var x MA_x;
...
model;
...
MA_x = @{1/(2*window+1)}*(
@#for i in -window:window
        +x(@{i})
@#endfor
       );
...
end;
```

After macro processing, this is equivalent to:

```
var x MA_x;
...
model;
...
MA_x = 0.2*(
        +x(-2)
        +x(-1)
        +x(0)
        +x(1)
        +x(2)
       );
...
end;
```

#### 4.29.3.3 Multi-country models

Here is a bare-bones example for a multi-country model:

```
@#define countries = [ "US", "EA", "AS", "JP", "RC" ]
@#define nth_co = "US"

@#for co in countries
var Y_@{co} K_@{co} L_@{co} i_@{co} E_@{co} ...;
parameters a_@{co} ...;
varexo ...;
@#endfor

model;
@#for co in countries
 Y_@{co} = K_@{co}^a_@{co} * L_@{co}^(1-a_@{co});
...
@#if co != nth_co
 (1+i_@{co}) = (1+i_@{nth_co}) * E_@{co}(+1) / E_@{co}; // UIP relation
@#else
 E_@{co} = 1;
@#endif
@#endfor
end;
```

#### 4.29.3.4 Endogeneizing parameters

When calibrating the model, it may be useful to pin down parameters by targeting endogenous objects.

For example, suppose production is defined by a CES function:

$$y_t = \left(\alpha^{1/\xi}\ell_t^{1-1/\xi} + (1-\alpha)^{1/\xi}k_t^{1-1/\xi}\right)^{\xi/(\xi-1)}$$

and the labor share in GDP is defined as:

$$\text{lab_rat}_t = (w_t\ell_t)/(p_t y_t)$$

In the model, $\alpha$ is a (share) parameter and $lab_rat_t$ is an endogenous variable.

It is clear that setting a value for $\alpha$ is not straightforward. But we have real world data for $lab_rat_t$, and it is clear that these two objects are economically linked.

The solution is to use a method called *variable flipping*, which consists in changing the way of computing the steady state. During this computation, $\alpha$ will be made an endogenous variable and the steady-state value $lab_rat$ of the dynamic variable $lab_rat_t$ will be made a parameter. An economically sensible value will be calibrated for $lab_rat$, and the solution algorithm will deduce the implied value for $\alpha$.

An implementation could consist of the following files:

`modeqs.mod`

> This file contains variable declarations and model equations. The code for the declaration of $\alpha$ and `lab_rat` would look like:
>
> ```
> @#if steady
>   var alpha;
>   parameters lab_rat;
> ```

(continues on next page)

(continued from previous page)

```
@#else
  parameters alpha;
  var lab_rat;
@#endif
```

`steadystate.mod`

This file computes the steady state. It begins with:

```
@#define steady = 1
@#include "modeqs.mod"
```

Then it initializes parameters (including `lab_rat`, excluding $\alpha$), computes the steady state (using guess values for endogenous, including $\alpha$), then saves values of parameters and variables at steady state in a file, using the `save_params_and_steady_state` command.

`simulate.mod`

This file computes the simulation. It begins with:

```
@#define steady = 0
@#include "modeqs.mod"
```

Then it loads values of parameters and variables at steady state from file, using the `load_params_and_steady_state` command, and computes the simulations.

### 4.29.4 MATLAB/Octave loops versus macro processor loops

Suppose you have a model with a parameter $\rho$ and you want to run simulations for three values: $\rho = 0.8, 0.9, 1$. There are several ways of doing this:

*With a MATLAB/Octave loop*

```
rhos = [ 0.8, 0.9, 1];
for i = 1:length(rhos)
  set_param_value('rho',rhos(i));
  stoch_simul(order=1);
  if info(1)~=0
    error('Simulation failed for parameter draw')
  end
end
```

Here the loop is not unrolled, MATLAB/Octave manages the iterations. This is interesting when there are a lot of iterations. It is strongly advised to always check whether the error flag `info(1)==0` to prevent erroneously relying on stale results from previous iterations.

*With a macro processor loop (case 1)*

```
rhos = [ 0.8, 0.9, 1];
@#for i in 1:3
  set_param_value('rho',rhos(@{i}));
  stoch_simul(order=1);
  if info(1)~=0
    error('Simulation failed for parameter draw')
  end
@#endfor
```

This is very similar to the previous example, except that the loop is unrolled. The macro processor manages the loop index but not the data array (`rhos`).

*With a macro processor loop (case 2)*

```
@#for rho_val in [ 0.8, 0.9, 1]
  rho = @{rho_val};
  stoch_simul(order=1);
  if info(1)~=0
    error('Simulation failed for parameter draw')
  end
@#endfor
```

The advantage of this method is that it uses a shorter syntax, since the list of values is directly given in the loop construct. The inconvenience is that you can not reuse the macro array in MATLAB/Octave.

## 4.30 Verbatim inclusion

Pass everything contained within the verbatim block to the <mod_file>.m file.

**Block:** `verbatim ;`

By default, whenever Dynare encounters code that is not understood by the parser, it is directly passed to the preprocessor output.

In order to force this behavior you can use the `verbatim` block. This is useful when the code you want passed to the driver file contains tokens recognized by the Dynare preprocessor.

*Example*

```
verbatim;
% Anything contained in this block will be passed
% directly to the driver file, including comments
var = 1;
end;
```

## 4.31 Misc commands

**Command:** `set_dynare_seed(INTEGER)`

**Command:** `set_dynare_seed('default')`

**Command:** `set_dynare_seed('clock')`

**Command:** `set_dynare_seed('reset')`

**Command:** `set_dynare_seed('ALGORITHM', INTEGER)`

Sets the seed used for random number generation. It is possible to set a given integer value, to use a default value, or to use the clock (by using the latter, one will therefore get different results across different Dynare runs). The `reset` option serves to reset the seed to the value set by the last `set_dynare_seed` command. On MATLAB 7.8 or above, it is also possible to choose a specific algorithm for random number generation; accepted values are `mcg16807`, `mlfg6331_64`, `mrg32k3a`, `mt19937ar` (the default), `shr3cong` and `swb2712`.

**Command:** `save_params_and_steady_state(FILENAME);`

For all parameters, endogenous and exogenous variables, stores their value in a text file, using a simple name/value associative table.

- for parameters, the value is taken from the last parameter initialization.
- for exogenous, the value is taken from the last `initval` block.
- for endogenous, the value is taken from the last steady-state computation (or, if no steady state has been computed, from the last `initval` block).

Note that no variable type is stored in the file, so that the values can be reloaded with `load_params_and_steady_state` in a setup where the variable types are different.

The typical usage of this function is to compute the steady state of a model by calibrating the steady-state value of some endogenous variables (which implies that some parameters must be endogeneized during the steady-state computation).

You would then write a first `.mod` file which computes the steady state and saves the result of the computation at the end of the file, using `save_params_and_steady_state`.

In a second file designed to perform the actual simulations, you would use `load_params_and_steady_state` just after your variable declarations, in order to load the steady state previously computed (including the parameters which had been endogeneized during the steady-state computation).

The need for two separate `.mod` files arises from the fact that the variable declarations differ between the files for steady-state calibration and for simulation (the set of endogenous and parameters differ between the two); this leads to different `var` and `parameters` statements.

Also note that you can take advantage of the `@#include` directive to share the model equations between the two files (see *Macro processing language*).

**Command:** `load_params_and_steady_state(FILENAME);`

For all parameters, endogenous and exogenous variables, loads their value from a file created with `save_params_and_steady_state`.

- for parameters, their value will be initialized as if they had been calibrated in the `.mod` file.
- for endogenous and exogenous variables, their value will be initialized as they would have been from an `initval` block .

This function is used in conjunction with `save_params_and_steady_state`; see the documentation of that function for more information.

**Command:** `compilation_setup(OPTIONS);`

When the *use_dll* option is present, Dynare uses the GCC compiler that was distributed with it to compile the static and dynamic C files produced by the preprocessor. You can use this option to change the compiler, flags, and libraries used.

*Options*

**compiler = FILENAME**

The path to the compiler.

**substitute_flags = QUOTED_STRING**

The flags to use instead of the default flags.

**add_flags = QUOTED_STRING**

The flags to use in addition to the default flags. If `substitute_flags` is passed, these flags are added to the flags specified there.

**substitute_libs = QUOTED_STRING**

The libraries to link against instead of the default libraries.

**add_libs = QUOTED_STRING**

The libraries to link against in addition to the default libraries. If `substitute_libs` is passed, these libraries are added to the libraries specified there.

**MATLAB/Octave command:** `dynare_version ;`

Output the version of Dynare that is currently being used (i.e. the one that is highest on the MATLAB/Octave path).

**MATLAB/Octave command:** `write_latex_definitions ;`

Writes the names, LaTeX names and long names of model variables to tables in a file named `<<M_.fname>>_latex_definitions.tex`. Requires the following LaTeX packages: `longtable`.

**MATLAB/Octave command: write_latex_parameter_table ;**

Writes the LaTeX names, parameter names, and long names of model parameters to a table in a file named <<M_.fname>>_latex_parameters.tex. The command writes the values of the parameters currently stored. Thus, if parameters are set or changed in the steady-state computation, the command should be called after a *steady* command to make sure the parameters were correctly updated. The long names can be used to add parameter descriptions. Requires the following LaTeX packages: longtable, booktabs.

**MATLAB/Octave command: write_latex_prior_table ;**

Writes descriptive statistics about the prior distribution to a LaTeX table in a file named <<M_.fname>>_latex_priors_table.tex. The command writes the prior definitions currently stored. Thus, this command must be invoked after the estimated_params block. If priors are defined over the measurement errors, the command must also be preceeded by the declaration of the observed variables (with varobs). The command displays a warning if no prior densities are defined (ML estimation) or if the declaration of the observed variables is missing. Requires the following LaTeX packages: longtable, booktabs.

**MATLAB/Octave command: collect_latex_files ;**

Writes a LaTeX file named <<M_.fname>>_TeX_binder.tex that collects all TeX output generated by Dynare into a file. This file can be compiled using pdflatex and automatically tries to load all required packages. Requires the following LaTeX packages: breqn, psfrag, graphicx, epstopdf, longtable, booktabs, caption, float, amsmath, amsfonts, amssymb, and morefloats.

CHAPTER

FIVE

# PARALLEL EXECUTION AND CONFIGURATION FILE

Dynare offers two approaches for parallelizing computations:

1. **Parallel Computing Toolbox (PCT)**: A simple, built-in approach that uses MATLAB's Parallel Computing Toolbox to distribute work across cores on a single machine. No configuration file is needed. See *Parallel Computing Toolbox (PCT)* below.
2. **Cluster-based parallelization**: A more flexible approach that spawns separate MATLAB/Octave processes on local or remote machines, communicating via SMB (Windows) or SSH (Unix). This requires a configuration file and the `parallel` command-line option. See *Cluster-based Parallel Configuration* below.

## 5.1 Parallel Computing Toolbox (PCT)

Starting with Dynare 7, certain estimation tasks can be parallelized using MATLAB's Parallel Computing Toolbox (PCT). This is the recommended approach for users running Dynare on a single multi-core machine, as it requires no configuration file, no SSH setup, and no PsTools installation.

**Requirements:**

- MATLAB R2024b or later
- Parallel Computing Toolbox installed and licensed

**Currently supported tasks:**

- The MCMC posterior sampler (each chain is dispatched to a separate worker via `parfeval`)
- The initialization of SMC samplers by drawing initial particles in the prior distribution using `parfor`

**Usage:**

Parallelization via PCT is controlled by the *use_pct* option of the `estimation` command. By default, `use_pct` is `true`, meaning that if PCT is available, the supported tasks will automatically run in parallel using the current parallel pool. If no pool is open, one will be created for the duration of the computation and closed afterwards.

**Tips for tuning the parallel pool:**

- You can control the pool configuration before calling `estimation`:

```
c = parcluster('local');
c.NumThreads = 2;
parpool(c, 4);
```

- Thread pools are not supported; use a process pool.

## 5.2 Cluster-based Parallel Configuration

This section explains the alternative way to configure Dynare for parallelizing some tasks that require very little inter-process communication.

The parallelization is done by running several MATLAB or Octave processes, either on local or on remote machines. Communication between leader and follower processes are done through SMB on Windows and SSH on

UNIX. Input and output data, and also some short status messages, are exchanged through network file systems. Currently, the system works only with homogenous grids: only Windows or only Unix machines.

The following routines are currently parallelized:

- the posterior sampling algorithms when using multiple chains;
- the Metropolis-Hastings diagnostics;
- the posterior IRFs;
- the prior and posterior statistics;
- some plotting routines.

Note that creating the configuration file is not enough in order to trigger parallelization of the computations: you also need to specify the `parallel` option to the `dynare` command. For more details, and for other options related to the parallelization engine, see *Dynare invocation*.

You also need to verify that the following requirements are met by your cluster (which is composed of a leader and of one or more followers):

For a Windows grid:

- a standard Windows network (SMB) must be in place;
- the PsTools suite must be installed in the path of the leader Windows machine;
- the Windows user on the leader machine has to be user of any other follower machine in the cluster, and that user will be used for the remote computations.
- detailed step-by-step setup instructions can be found in *Windows Step-by-Step Guide*.

For a UNIX grid:

- SSH must be installed on the leader and on the follower machines;
- SSH keys must be installed so that the SSH connection from the leader to the follower can be done without passwords, or using an SSH agent.

**Warning**

Compatibility considerations between leader and follower

It is highly recommended to use the same version of Dynare on both the leader and all followers. Different versions regularly cause problems like zero acceptance rates during estimation. When upgrading to a newer Dynare version do not forget to adjust the `DynarePath`.

### 5.2.1 The Configuration File

The configuration file is used to provide Dynare with information not related to the model (and hence not placed in the model file). At the moment, it is only used when using Dynare to run parallel computations.

On Linux and macOS, the configuration file is searched by default under `dynare/dynare.ini` in the configuration directories defined by the XDG specification (typically `$HOME/.config/dynare/dynare.ini` for the user-specific configuration and `/etc/xdg/dynare/dynare.ini` for the system-wide configuration, the former having precedence over the latter). Under Windows, the configuration file is searched by default in `%APPDATA%\dynare\dynare.ini` (typically `c:\Users\USERNAME\AppData\Roaming\dynare\dynare.ini`). You can specify a non-standard location using the `conffile` option of the `dynare` command (see *Dynare invocation*).

The parsing of the configuration file is case-sensitive, and it should take the following form, with each option/choice pair placed on a newline:

```
[command0]
option0 = choice0
option1 = choice1
```

(continues on next page)

(continued from previous page)

```
[command1]
option0 = choice0
option1 = choice1
```

The configuration file follows a few conventions (self-explanatory conventions such as USER_NAME have been excluded for concision):

COMPUTER_NAME

Indicates the valid name of a server (e.g. `localhost`, `server.cepremap.org`) or an IP address.

DRIVE_NAME

Indicates a valid drive name on Windows, without the trailing colon (e.g. `C`).

PATH

Indicates a valid path in the underlying operating system (e.g. `/home/user/dynare/matlab/`).

PATH_AND_FILE

Indicates a valid path to a file in the underlying operating system (e.g. `/usr/local/MATLAB/R2023b/bin/matlab`).

BOOLEAN

Is `true` or `false`.

We now turn to the description of the configuration directives. Note that comments in the configuration file can be provided by separate lines starting with a hashtag (#).

**Configuration block: [cluster]**

When working in parallel, `[cluster]` is required to specify the group of computers that will be used. It is required even if you are only invoking multiple processes on one computer.

*Options*

**Name = CLUSTER_NAME**

The reference name of this cluster.

**Members = NODE_NAME[(WEIGHT)] NODE_NAME[(WEIGHT)] ...**

A list of nodes that comprise the cluster with an optional computing weight specified for that node. The computing weight indicates how much more powerful one node is with respect to the others (e.g. `n1(2) n2(1) n3(3)` means that `n1` is two times more powerful than `n2` whereas `n3` is three times more powerful than `n2`). Each node is separated by at least one space and the weights are in parentheses with no spaces separating them from their node.

*Example*

```
[cluster]
Name = c1
Members = n1 n2 n3

[cluster]
Name = c2
Members = n1(4) n2 n3
```

**Configuration block: [node]**

When working in parallel, `[node]` is required for every computer that will be used. The options that are required differ, depending on the underlying operating system and whether you are working locally or remotely.

*Options*

**Name = NODE_NAME**

The reference name of this node.

**CPUnbr = INTEGER | [INTEGER:INTEGER]**

If just one integer is passed, the number of processors to use. If a range of integers is passed, the specific processors to use (processor counting is defined to begin at one as opposed to zero). Note that using specific processors is only possible under Windows; under Linux and macOS, if a range is passed the same number of processors will be used, but the range will be adjusted to begin at one.

**ComputerName = COMPUTER_NAME**

The name or IP address of the node. If you want to run locally, use `localhost` (case-sensitive).

**Port = INTEGER**

The port number to connect to on the node. The default is empty, meaning that the connection will be made to the default SSH port (22).

**UserName = USER_NAME**

The username used to log into a remote system. Required for remote runs on all platforms.

**Password = PASSWORD**

The password used to log into the remote system. Required for remote runs originating from Windows.

**RemoteDrive = DRIVE_NAME**

The drive to be used for remote computation. Required for remote runs originating from Windows.

**RemoteDirectory = PATH**

The directory to be used for remote computation. Required for remote runs on all platforms.

**DynarePath = PATH**

The path to the `matlab` subdirectory within the Dynare installation directory. The default is the empty string.

**MatlabOctavePath = PATH_AND_FILE**

The path to the MATLAB or Octave executable. The default value is `matlab` as MATLAB's executable is typically in the %PATH% environment variable. When using full paths on Windows, you may need to enclose the path in quoted strings, e.g. `MatlabOctavePath="C:\Program Files\MATLAB\R2023b\bin\matlab.exe"`

**NumberOfThreadsPerJob = INTEGER**

This option controls the distribution of jobs (e.g. MCMC chains) across additional MATLAB instances that are run in parallel. Needs to be an exact divisor of the number of cores. The formula *CPUnbr* divided by *NumberOfThreadsPerJob* calculates the number of MATLAB/Octave instances that will be launched in parallel, where each instance will then execute a certain number of jobs sequentially. For example, if you run a MCMC estimation with 24 chains on a 12 core machine, setting `CPUnbr = 12` and `NumberOfThreadsPerJob = 4` will launch 3 MATLAB instances in parallel, each of which will compute 8 chains sequentially. Note that this option does not dictate the number of maximum threads utilized by each MATLAB/Octave instance, see related option *SingleCompThread* for this. Particularly for very large models, setting this option to 2 might distribute the workload in a more efficient manner, depending on your hardware and task specifics. It's advisable to experiment with different values to achieve optimal performance. The default value is 1.

**SingleCompThread = BOOLEAN**

This option allows you to enable or disable MATLAB's native multithreading capability. When set to `true`, the additional MATLAB instances are initiated in single thread mode utilizing the `-singleCompThread` startup option, thereby disabling MATLAB's native multithreading. When set to `false`, MATLAB's native multithreading is enabled, e.g. the actual number of threads utilized by each MATLAB instance is usually determined by the number of CPU cores (you can check this by running `maxNumCompThreads` in MATLAB's command window). Note: While MATLAB aims to accelerate calculations by distributing them across your computer's threads, certain tasks, like MCMC estimations, may exhibit slowdowns with MATLAB's multitasking especially when Dynare's parallel computing is turned on as we do not use MATLAB's parallel toolbox. So in many cases it is advisable

to set this setting to `true`. If you want to have more control, you can manually add the MATLAB command *maxNumCompThreads(N)* at the beginning of *fParallel.m*. The default value is `false`. This option is ineffective under Octave.

**`OperatingSystem = OPERATING_SYSTEM`**

The operating system associated with a node. Only necessary when creating a cluster with nodes from different operating systems. Possible values are `unix` or `windows`. There is no default value.

*Example*

```
[node]
Name = n1
ComputerName = localhost
CPUnbr = 1

[node]
Name = n2
ComputerName = dynserv.cepremap.org
CPUnbr = 5
UserName = usern
RemoteDirectory = /home/usern/Remote
DynarePath = /home/usern/dynare/matlab
MatlabOctavePath = matlab

[node]
Name = n3
ComputerName = dynserv.dynare.org
Port = 3333
CPUnbr = [2:4]
UserName = usern
RemoteDirectory = /home/usern/Remote
DynarePath = /home/usern/dynare/matlab
MatlabOctavePath = matlab
```

### 5.2.2 Windows Step-by-Step Guide

This section outlines the steps necessary on most Windows systems to set up Dynare for parallel execution. Note that the steps 3 to 6 are required unless parallel execution is confined to a local pool with the `parallel_use_psexec=false` option.

1. Write a configuration file containing the options you want. A minimum working example setting up a cluster consisting of two local CPU cores that allows for e.g. running two Monte Carlo Markov Chains in parallel is shown below.
2. Save the configuration file somewhere. The name and file ending do not matter if you are providing it with the `conffile` command line option. The only restrictions are that the path must be a valid filename, not contain non-alpha-numeric characters, and not contain any whitespace. For the configuration file to be accessible without providing an explicit path at the command line, you must save it under the name `dynare.ini` into your user account's `Application Data` folder.
3. Install PSTools to your system, e.g. into `C:\PSTools`.
4. Set the Windows System Path to the `PSTools` folder (e.g. using something along the line of pressing Windows Key+Pause to open the System Configuration, then go to Advanced -> Environment Variables -> Path).
5. Restart your computer to make the path change effective.
6. Open MATLAB and type into the command window:

```
!psexec
```

This executes the `psexec.exe` from PSTools on your system and shows whether Dynare will be able to locate it. If MATLAB complains at this stage, you did not correctly set your Windows system path for the `PSTools` folder.

7. If `psexec.exe` was located in the previous step, a popup will show up, asking for confirmation of the license agreement. Confirm this copyright notice of `psexec` (this needs to be done only once). After this, Dynare should be ready for parallel execution.

8. Call Dynare on your mod-file invoking the `parallel` option and providing the path to your configuration file with the `conffile` option (if you did not save it as `%APPDATA%\dynare.ini` in step 2 where it should be detected automatically):

```
dynare ls2003 parallel conffile='C:\Users\Dynare~1\parallel\conf_file.
↪ini'
```

Please keep in mind that no white spaces or names longer than 8 characters are allowed in the `conffile` path. The 8-character restriction can be circumvented by using the tilde Windows path notation as in the above example.

*Example*:

```
#cluster needs to always be defined first
[cluster]
#Provide a name for the cluster
Name=Local
#declare the nodes being member of the cluster
Members=n1

#declare nodes (they need not all be part of a cluster)
[node]
#name of the node
Name=n1
#name of the computer (localhost for the current machine)
ComputerName=localhost
#cores to be included from this node
CPUnbr=[1:2]
#path to matlab.exe; on Windows, the MATLAB bin folder is in the system path
#so we only need to provide the name of the exe file
MatlabOctavePath=matlab
#Dynare path you are using
DynarePath=C:/dynare/4.7.0/matlab
```

## 5.3 Dynare Configuration

This section explains how to configure Dynare for general processing. Currently, there is only one option available.

**`Configuration block: [hooks]`**

This block can be used to specify configuration options that will be used when running Dynare.

*Options*

**`GlobalInitFile = PATH_AND_FILE`**

The location of a global initialization file that can be used to customize some Dynare internals (typically default option values). This is a MATLAB/Octave script.

If this option is not specified, Dynare will look for a `global_init.m` file in its configuration directory (typically `$HOME/.config/dynare/global_init.m` under Linux and macOS, and `c:\Users\USERNAME\AppData\Roaming\dynare\global_init.m` under Windows).

*Example*

```
[hooks]
GlobalInitFile = /home/usern/dynare/myInitFile.m
```

**Configuration block: [paths]**

This block can be used to specify paths that will be used when running Dynare.

*Options*

**Include = PATH**

A colon-separated path to use when searching for files to include via `@#include`. Paths specified via *-I* take priority over paths specified here, while these paths take priority over those specified by `@#includepath`.

*Example*

```
[paths]
Include = /path/to/folder/containing/modfiles:/path/to/another/folder
```

CHAPTER

SIX

# TIME SERIES

Dynare provides a MATLAB/Octave class for handling time series data, which is based on a class for handling dates. Dynare also provides a new type for dates, so that the user does not have to worry about class and methods for dates. Below, you will first find the class and methods used for creating and dealing with dates and then the class used for using time series. Dynare also provides an interface to the X-13 ARIMA-SEATS seasonal adjustment program produced, distributed, and maintained by the U.S. Census Bureau (2020).

## 6.1 Dates

### 6.1.1 Dates in a mod file

Dynare understands dates in a mod file. Users can declare annual, bi-annual, quarterly, or monthly dates using the following syntax:

```
1990Y
1990A
1990S2
1990H2
1990Q4
1990M11
```

Note that there are two syntaxes for annual dates (*1990A* is equivalent to *1990Y*), and for bi-annual dates (*1990H2* is equivalent to *1990S2*).

Behind the scene, Dynare's preprocessor translates these expressions into instantiations of the MATLAB/Octave's class `dates` described below. Basic operations can be performed on dates:

**plus binary operator (+)**

An integer scalar, interpreted as a number of periods, can be added to a date. For instance, if `a = 1950Q1` then `b = 1951Q2` and `b = a + 5` are identical.

**plus unary operator (+)**

Increments a date by one period. `+1950Q1` is identical to `1950Q2`, `++++1950Q1` is identical to `1951Q1`.

**minus binary operator (-)**

Has two functions: difference and subtraction. If the second argument is a date, calculates the difference between the first date and the second date (e.g. `1951Q2-1950Q1` is equal to `5`). If the second argument is an integer `X`, subtracts `X` periods from the date (e.g. `1951Q2-2` is equal to `1950Q4`).

**minus unary operator (-)**

Subtracts one period to a date. `-1950Q1` is identical to `1949Q4`. The unary minus operator is the reciprocal of the unary plus operator, `+-1950Q1` is identical to `1950Q1`.

**colon operator (:)**

Can be used to create a range of dates. For instance, `r = 1950Q1:1951Q1` creates a `dates` object with five elements: `1950Q1`, `1950Q2`, `1950Q3`, `1950Q4` and `1951Q1`. By default, the increment between

each element is one period. This default can be changed using, for instance, the following instruction: `1950Q1:2:1951Q1` which will instantiate a `dates` object with three elements: `1950Q1`, `1950Q3` and `1951Q1`.

**horzcat operator ([,])**

Concatenates dates objects without removing repetitions. For instance `[1950Q1, 1950Q2]` is a `dates` object with two elements (`1950Q1` and `1950Q2`).

**vertcat operator ([;])**

Same as `horzcat` operator.

**eq operator (equal, ==)**

Tests if two `dates` objects are equal. `+1950Q1==1950Q2` returns `true`, `1950Q1==1950Q2` returns `false`. If the compared objects have both `n>1` elements, the `eq` operator returns a column vector, `n` by 1, of Booleans.

**ne operator (not equal, ~=)**

Tests if two `dates` objects are not equal. `+1950Q1~=` returns `false` while `1950Q1~=1950Q2` returns `true`. If the compared objects both have `n>1` elements, the `ne` operator returns an `n` by 1 column vector of Booleans.

**lt operator (less than, <)**

Tests if a `dates` object precedes another `dates` object. For instance, `1950Q1<1950Q3` returns `true`. If the compared objects have both `n>1` elements, the `lt` operator returns a column vector, `n` by 1, of Booleans.

**gt operator (greater than, >)**

Tests if a `dates` object follows another `dates` object. For instance, `1950Q1>1950Q3` returns `false`. If the compared objects have both `n>1` elements, the `gt` operator returns a column vector, `n` by 1, of Booleans.

**le operator (less or equal, <=)**

Tests if a `dates` object precedes another `dates` object or is equal to this object. For instance, `1950Q1<=1950Q3` returns `true`. If the compared objects have both `n>1` elements, the `le` operator returns a column vector, `n` by 1, of Booleans.

**ge operator (greater or equal, >=)**

Tests if a `dates` object follows another `dates` object or is equal to this object. For instance, `1950Q1>=1950Q3` returns `false`. If the compared objects have both `n>1` elements, the `ge` operator returns a column vector, `n` by 1, of Booleans.

One can select an element, or some elements, in a `dates` object as he would extract some elements from a vector in MATLAB/Octave. Let `a = 1950Q1:1951Q1` be a `dates` object, then `a(1)==1950Q1` returns `true`, `a(end)==1951Q1` returns `true` and `a(end-1:end)` selects the two last elements of `a` (by instantiating the `dates` object `[1950Q4, 1951Q1]`).

Remark: Dynare substitutes any occurrence of dates in the `.mod` file into an instantiation of the `dates` class regardless of the context. For instance, `d = 1950Q1` will be translated as `d = dates('1950Q1');`. This automatic substitution can lead to a crash if a date is defined in a string. Typically, if the user wants to display a date:

```
disp('Initial period is 1950Q1');
```

Dynare will translate this as:

```
disp('Initial period is dates('1950Q1')');
```

which will lead to a crash because this expression is illegal in MATLAB/Octave. For this situation, Dynare provides the `$` escape parameter. The following expression:

```
disp('Initial period is $1950Q1');
```

will be translated as:

```
disp('Initial period is 1950Q1');
```

in the generated MATLAB/Octave script.

### 6.1.2 The dates class

**Dynare class: dates**

**Members**

- **freq** – equal to 1, 2, 4, 12 or 365 (resp. for annual, bi-annual, quarterly, monthly, or daily dates).
- **time** – a n*1 array of integers, the number of periods since year 0.

Each member is private, one can display the content of a member but cannot change its value directly. Note also that it is not possible to mix frequencies in a `dates` object: all the elements must have common frequency.

The `dates` class has the following constructors:

**Constructor: dates()**

**Constructor: dates(FREQ)**

Returns an empty `dates` object with a given frequency (if the constructor is called with one input argument). `FREQ` is a character equal to 'Y' or 'A' for annual dates, 'S' or 'H' for bi-annual dates, 'Q' for quarterly dates, 'M' for monthly dates, or 'D' for daily dates. Note that `FREQ` is not case-sensitive, so that, for instance, 'q' is also allowed for quarterly dates. The frequency can also be set with an integer scalar equal to 1 (annual), 2 (bi-annual), 4 (quarterly), 12 (monthly), or 365 (daily). The instantiation of empty objects can be used to rename the `dates` class. For instance, if one only works with quarterly dates, object `qq` can be created as:

```
qq = dates('Q')
```

and a `dates` object holding the date `2009Q2`:

```
d0 = qq(2009,2);
```

which is much simpler if `dates` objects have to be defined programmatically. For daily dates, we would instantiate an empty daily dates object as:

```
dd = dates('D')
```

and a `dates` object holding the date `2020-12-31`:

```
d1 = dd(2020,12,31);
```

**Constructor: dates(STRING)**

**Constructor: dates(STRING, STRING, ...)**

Returns a `dates` object that represents a date as given by the string `STRING`. This string has to be interpretable as a date (only strings of the following forms are admitted: `'1990Y'`, `'1990A'`, `1990S1`, `1990H1`, `'1990Q1'`, `'1990M2'`, or `'2020-12-31'`), the routine `isdate` can be used to test if a string is interpretable as a date. If more than one argument is provided, they should all be dates represented as strings, the resulting `dates` object contains as many elements as arguments to the constructor. For the daily dates, the string must be of the form yyyy-mm-dd with two digits for the months (mm) and days (dd), even if the number of days or months is smaller than ten (in this case a leading 0 is required).

**Constructor: dates(DATES)**

**Constructor: dates(DATES, DATES, ...)**

Returns a copy of the `dates` object `DATES` passed as input arguments. If more than one argument is provided, they should all be `dates` objects. The number of elements in the instantiated `dates` object is equal to the sum of the elements in the `dates` passed as arguments to the constructor.

**Constructor: dates(FREQ, YEAR, SUBPERIOD[, S])**

where `FREQ` is a single character ('Y', 'A', 'S', 'H', 'Q', 'M', 'D') or integer (1, 2, 4, 12, or 365) specifying the frequency, `YEAR` and `SUBPERIOD` and `S` are `n*1` vectors of integers. Returns a `dates` object with `n` elements. The last argument, `S`, is only to be used for daily frequency. If `FREQ` is equal to `'Y'`, `'A'` or `1`, the third argument is not needed (because `SUBPERIOD` is necessarily a vector of ones in this case).

*Example*

```
do1 = dates('1950Q1');
do2 = dates('1950Q2','1950Q3');
do3 = dates(do1,do2);
do4 = dates('Q',1950, 1);
do5 = dates('D',1973, 1, 25);
```

A `dates` object with multiple elements can be considered a one-dimensional array of dates. Standard array operations can be applied to a `dates` object:

- square brackets can be used to concatenate dates objects:

```
>> A = dates('1938Q4');
>> B = dates('1945Q3');
>> C = [A, B];
```

- semicolons can be used to create ranges of dates:

```
>> A = dates('2009Q2');
>> B = A:A+2;
>> B

B = <dates: 2009Q2, 2009Q3, 2009Q4>
```

- objects can be indexed by an integer or a vector of integer:

```
>> B(1)

ans = <dates: 2009Q2>

>> B(end)

ans = <dates: 2009Q4>

>> B(1:2)

ans = <dates: 2009Q2, 2009Q3>
```

A list of the available methods, by alphabetical order, is given below. Note that by default the methods do not allow in place modifications: when a method is applied to an object a new object is instantiated. For instance, to apply the method `multiplybytwo` to an object `X` we write:

```
>> X = 2;
>> Y = X.multiplybytwo();
>> X

2

>> Y

4
```

or equivalently:

```
>> Y = multiplybytwo(X);
```

the object `X` is left unchanged, and the object `Y` is a modified copy of `X` (multiplied by two). This behaviour is altered if the name of the method is postfixed with an underscore. In this case the creation of a copy is avoided. For instance, following the previous example, we would have:

```
>> X = 2;
>> X.multiplybytwo_();
>> X

4
```

Modifying the objects in place, with underscore methods, is particularly useful if the methods are called in loops, since this saves the object instantiation overhead.

**Method:** **C = append**(A, B)

**Method:** **append_**(A, B)

Appends `dates` object `B`, or a string that can be interpreted as a date, to the `dates` object `A`. If `B` is a `dates` object it is assumed that it has no more than one element.

*Example*

```
>> D = dates('1950Q1','1950Q2');
>> d = dates('1950Q3');
>> E = D.append(d);
>> F = D.append('1950Q3');
>> isequal(E,F)

ans =

     1
>> F

F = <dates: 1950Q1, 1950Q2, 1950Q3>

>> D

D = <dates: 1950Q1, 1950Q2>

>> D.append_('1950Q3')

ans = <dates: 1950Q1, 1950Q2, 1950Q3>
```

**Method:** **B = char**(A)

Overloads the MATLAB/Octave `char` function. Converts a `dates` object into a character array.

*Example*

```
>> A = dates('1950Q1');
> A.char()

ans =

'1950Q1'
```

**Method:** `C = colon(A, B)`

**Method:** `C = colon(A, i, B)`

Overloads the MATLAB/Octave colon (`:`) operator. A and B are `dates` objects. The optional increment `i` is a scalar integer (default value is `i=1`). This method returns a `dates` object and can be used to create ranges of dates.

*Example*

```
>> A = dates('1950Q1');
>> B = dates('1951Q2');
>> C = A:B

C = <dates: 1950Q1, 1950Q2, 1950Q3, 1950Q4, 1951Q1>

>> D = A:2:B

D = <dates: 1950Q1, 1950Q3, 1951Q1>
```

**Method:** `B = copy(A)`

Returns a copy of a `dates` object.

**Method:** `disp(A)`

Overloads the MATLAB/Octave `disp` function for `dates` object.

**Method:** `display(A)`

Overloads the MATLAB/Octave `display` function for `dates` object.

*Example*

```
>> disp(B)

B = <dates: 1950Q1, 1950Q2, 1950Q3, 1950Q4, 1951Q1, 1951Q2, 1951Q3,␣
↪1951Q4, 1952Q1, 1952Q2, 1952Q3>


>> display(B)

B = <dates: 1950Q1, 1950Q2, ..., 1952Q2, 1952Q3>
```

**Method:** `B = double(A)`

Overloads the MATLAB/Octave `double` function. `A` is a `dates` object. The method returns a floating point representation of a `dates` object, the integer and fractional parts respectively corresponding to the year and the subperiod. The fractional part is the subperiod number minus one divided by the frequency (1, 4, or 12).

*Example*:

```
>> a = dates('1950Q1'):dates('1950Q4');
>> a.double()
```

(continues on next page)

(continued from previous page)

```
ans =

     1950.00
     1950.25
     1950.50
     1950.75
```

**Method:** `C = eq(A, B)`

Overloads the MATLAB/Octave `eq` (equal, ==) operator. `dates` objects `A` and `B` must have the same number of elements (say, `n`). The returned argument is a `n` by 1 vector of Booleans. The *i* -th element of `C` is equal to `true` if and only if the dates `A(i)` and `B(i)` are the same.

*Example*

```
>> A = dates('1950Q1','1951Q2');
>> B = dates('1950Q1','1950Q2');
>> A==B

ans =

  2x1 logical array

   1
   0
```

**Method:** `C = ge(A, B)`

Overloads the MATLAB/Octave `ge` (greater or equal, >=) operator. `dates` objects `A` and `B` must have the same number of elements (say, `n`). The returned argument is a `n` by 1 vector of Booleans. The *i* -th element of `C` is equal to `true` if and only if the date `A(i)` is posterior or equal to the date `B(i)`.

*Example*

```
>> A = dates('1950Q1','1951Q2');
>> B = dates('1950Q1','1950Q2');
>> A>=B

ans =

  2x1 logical array

   1
   1
```

**Method:** `C = gt(A, B)`

Overloads the MATLAB/Octave `gt` (greater than, >) operator. `dates` objects `A` and `B` must have the same number of elements (say, `n`). The returned argument is a `n` by 1 vector of Booleans. The *i* -th element of `C` is equal to 1 if and only if the date `A(i)` is posterior to the date `B(i)`.

*Example*

```
>> A = dates('1950Q1','1951Q2');
>> B = dates('1950Q1','1950Q2');
>> A>B

ans =

  2x1 logical array
```

(continues on next page)

(continued from previous page)

```
  0
  1
```

**Method:** `D = horzcat(A, B, C, ...)`

Overloads the MATLAB/Octave `horzcat` operator. All the input arguments must be `dates` objects. The returned argument is a `dates` object gathering all the dates given in the input arguments (repetitions are not removed).

*Example*

```
>> A = dates('1950Q1');
>> B = dates('1950Q2');
>> C = [A, B];
>> C

C = <dates: 1950Q1, 1950Q2>
```

**Method:** `C = intersect(A, B)`

Overloads the MATLAB/Octave `intersect` function. All the input arguments must be `dates` objects. The returned argument is a `dates` object gathering all the common dates given in the input arguments. If `A` and `B` are disjoint `dates` objects, the function returns an empty `dates` object. Returned dates in `dates` object `C` are sorted by increasing order.

*Example*

```
>> A = dates('1950Q1'):dates('1951Q4');
>> B = dates('1951Q1'):dates('1951Q4');
>> C = intersect(A, B);
>> C

C = <dates: 1951Q1, 1951Q2, 1951Q3, 1951Q4>
```

**Method:** `B = isempty(A)`

Overloads the MATLAB/Octave `isempty` function.

*Example*

```
>> A = dates('1950Q1');
>> A.isempty()

ans =

  logical

  0

>> B = dates();
>> B.isempty()

ans =

  logical

  1
```

**Method:** `C = isequal(A, B)`

Overloads the MATLAB/Octave `isequal` function.

*Example*

```
>> A = dates('1950Q1');
>> B = dates('1950Q2');
>> isequal(A, B)

ans =

  logical

  0
```

**Method:** `C = le(A, B)`

Overloads the MATLAB/Octave `le` (less or equal, <=) operator. `dates` objects `A` and `B` must have the same number of elements (say, `n`). The returned argument is a `n` by `1` vector of Booleans. The *i* -th element of `C` is equal to `true` if and only if the date `A(i)` is anterior or equal to the date `B(i)`.

*Example*

```
>> A = dates('1950Q1','1951Q2');
>> B = dates('1950Q1','1950Q2');
>> A<=B

ans =

  2x1 logical array

   1
   0
```

**Method:** `B = length(A)`

Overloads the MATLAB/Octave `length` function. Returns the number of elements in a `dates` object.

*Example*

```
>> A = dates('1950Q1'):dates(2000Q3);
>> A.length()

ans =

   203
```

**Method:** `C = lt(A, B)`

Overloads the MATLAB/Octave `lt` (less than, <) operator. `dates` objects `A` and `B` must have the same number of elements (say, `n`). The returned argument is a `n` by `1` vector of Booleans. The *i* -th element of `C` is equal to `true` if and only if the date `A(i)` is anterior or equal to the date `B(i)`.

*Example*

```
>> A = dates('1950Q1','1951Q2');
>> B = dates('1950Q1','1950Q2');
>> A<B

ans =

  2x1 logical array

   0
   0
```

**Method:** `D = max(A, B, C, ...)`

Overloads the MATLAB/Octave `max` function. All input arguments must be `dates` objects. The function returns a single element `dates` object containing the greatest date.

*Example*

```
>> A = {dates('1950Q2'), dates('1953Q4','1876Q2'), dates('1794Q3')};
>> max(A{:})

ans = <dates: 1953Q4>
```

**Method:** `D = min(A, B, C, ...)`

Overloads the MATLAB/Octave `min` function. All input arguments must be `dates` objects. The function returns a single element `dates` object containing the smallest date.

*Example*

```
>> A = {dates('1950Q2'), dates('1953Q4','1876Q2'), dates('1794Q3')};
>> min(A{:})

ans = <dates: 1794Q3>
```

**Method:** `C = minus(A, B)`

Overloads the MATLAB/Octave `minus` operator (`-`). If both input arguments are `dates` objects, then number of periods between `A` and `B` is returned (so that `A+C=B`). If `B` is a vector of integers, the minus operator shifts the `dates` object by `B` periods backward.

*Example*

```
>> d1 = dates('1950Q1','1950Q2','1960Q1');
>> d2 = dates('1950Q3','1950Q4','1960Q1');
>> ee = d2-d1

ee =

     2
     2
     0

>> d1-(-ee)

ans = <dates: 1950Q3, 1950Q4, 1960Q1>
```

**Method:** `C = mtimes(A, B)`

Overloads the MATLAB/Octave `mtimes` operator (`*`). `A` and `B` are respectively expected to be a `dates` object and a scalar integer. Returns `dates` object `A` replicated `B` times.

*Example*

```
>> d = dates('1950Q1');
>> d*2

ans = <dates: 1950Q1, 1950Q1>
```

**Method:** `s = ndat(A)`

Returns the number of elements in the `dates` object `A`. This method is deprecated, use `length` instead.

**Method:** `C = ne(A, B)`

Overloads the MATLAB/Octave `ne` (not equal, `~=`) operator. `dates` objects `A` and `B` must have the same number of elements (say, `n`) or one of the inputs must be a single element `dates` object. The

returned argument is a `n` by 1 vector of Booleans. The *i* -th element of `C` is equal to `true` if and only if the dates `A(i)` and `B(i)` are different.

*Example*

```
>> A = dates('1950Q1','1951Q2');
>> B = dates('1950Q1','1950Q2');
>> A~=B

ans =

  2x1 logical array

   0
   1
```

**Method:** **C = plus(A, B)**

Overloads the MATLAB/Octave `plus` operator (+). If both input arguments are `dates` objects, then the method combines `A` and `B` without removing repetitions. If `B` is a vector of integers, the `plus` operator shifts the `dates` object by `B` periods forward.

*Example*

```
>> d1 = dates('1950Q1','1950Q2')+dates('1960Q1');
>> d2 = (dates('1950Q1','1950Q2')+2)+dates('1960Q1');
>> ee = d2-d1;

ee =

     2
     2
     0

>> d1+ee
ans = <dates: 1950Q3, 1950Q4, 1960Q1>
```

**Method:** **C = pop(A)**

**Method:** **C = pop(A, B)**

**Method:** **pop_(A)**

**Method:** **pop_(A, B)**

Pop method for `dates` class. If only one input is provided, the method removes the last element of a `dates` object. If a second input argument is provided, a scalar integer between 1 and `A.length()`, the method removes element number `B` from `dates` object `A`.

*Example*

```
>> d = dates('1950Q1','1950Q2');
>> d.pop()

ans = <dates: 1950Q1>

>> d.pop_(1)

ans = <dates: 1950Q2>
```

**Method:** **C = remove(A, B)**

**Method:** **remove_(A, B)**

Remove method for `dates` class. Both inputs have to be `dates` objects, removes dates in `B` from `A`.

*Example*

```
>> d = dates('1950Q1','1950Q2');
>> d.remove(dates('1950Q2'))

ans = <dates: 1950Q1>
```

**Method:** `C = setdiff(A, B)`

Overloads the MATLAB/Octave `setdiff` function. All the input arguments must be `dates` objects. The returned argument is a `dates` object all dates present in `A` but not in `B`. If `A` and `B` are disjoint `dates` objects, the function returns `A`. Returned dates in `dates` object `C` are sorted by increasing order.

*Example*

```
>> A = dates('1950Q1'):dates('1969Q4');
>> B = dates('1960Q1'):dates('1969Q4');
>> C = dates('1970Q1'):dates('1979Q4');
>> setdiff(A, B)

ans = <dates: 1950Q1, 1950Q2,  ..., 1959Q3, 1959Q4>

>> setdiff(A, C)

ans = <dates: 1950Q1, 1950Q2,  ..., 1969Q3, 1969Q4>
```

**Method:** `B = sort(A)`

**Method:** `sort_(A)`

Sort method for `dates` objects. Returns a `dates` object with elements sorted by increasing order.

*Example*

```
>> dd = dates('1945Q3','1938Q4','1789Q3');
>> dd.sort()

ans = <dates: 1789Q3, 1938Q4, 1945Q3>
```

**Method:** `B = strings(A)`

Converts a `dates` object into a cell of char arrays.

*Example*

```
>> A = dates('1950Q1');
>> A = A:A+1;
>> A.strings()

  ans =

    1x2 cell array

    {'1950Q1'}    {'1950Q2'}
```

**Method:** `B = subperiod(A)`

Returns the subperiod of a date (an integer scalar between 1 and `A.freq`). This method is not implemented for daily dates.

*Example*

```
>> A = dates('1950Q2');
>> A.subperiod()
```

(continues on next page)

(continued from previous page)

```
ans =

     2
```

**Method:** `B = uminus(A)`

Overloads the MATLAB/Octave unary minus operator. Returns a `dates` object with elements shifted one period backward.

*Example*

```
>> dd = dates('1945Q3','1938Q4','1973Q1');
>> -dd

ans = <dates: 1945Q2, 1938Q3, 1972Q4>
```

**Method:** `D = union(A, B, C, ...)`

Overloads the MATLAB/Octave `union` function. Returns a `dates` object with elements sorted by increasing order (repetitions are removed, to keep the repetitions use the `horzcat` or `plus` operators).

*Example*

```
>> d1 = dates('1945Q3','1973Q1','1938Q4');
>> d2 = dates('1973Q1','1976Q1');
>> union(d1,d2)

ans = <dates: 1938Q4, 1945Q3, 1973Q1, 1976Q1>
```

**Method:** `B = unique(A)`

**Method:** `unique_(A)`

Overloads the MATLAB/Octave `unique` function. Returns a `dates` object with repetitions removed (only the last occurrence of a date is kept).

*Example*

```
>> d1 = dates('1945Q3','1973Q1','1945Q3');
>> d1.unique()

ans = <dates: 1973Q1, 1945Q3>
```

**Method:** `B = uplus(A)`

Overloads the MATLAB/Octave unary plus operator. Returns a `dates` object with elements shifted one period ahead.

*Example*

```
>> dd = dates('1945Q3','1938Q4','1973Q1');
>> +dd

ans = <dates: 1945Q4, 1939Q1, 1973Q2>
```

**Method:** `D = vertcat(A, B, C, ...)`

Overloads the MATLAB/Octave `horzcat` operator. All the input arguments must be `dates` objects. The returned argument is a `dates` object gathering all the dates given in the input arguments (repetitions are not removed).

**Method:** `B = year(A)`

Returns the year of a date (an integer scalar between 1 and `A.freq`).

*Example*

```
>> A = dates('1950Q2');
>> A.subperiod()

ans =

       1950
```

## 6.2 The dseries class

**Dynare class: dseries**

The MATLAB/Octave `dseries` class handles time series data. As any MATLAB/Octave statements, this class can be used in a Dynare's mod file. A `dseries` object has six members:

**Members**

- **name** – A `vobs*1` cell of strings or a `vobs*p` character array, the names of the variables.
- **tex** – A `vobs*1` cell of strings or a `vobs*p` character array, the tex names of the variables.
- **dates** (`dates`) – An object with `nobs` elements, the dates of the sample.
- **data** (`double`) – A `nobs` by `vobs` array, the data.
- **ops** – The history of operations on the variables.
- **tags** – The user-defined tags on the variables.

`data`, `name`, `tex`, and `ops` are private members. The following constructors are available:

**Constructor: dseries()**

**Constructor: dseries(INITIAL_DATE)**

Instantiates an empty `dseries` object with, if defined, an initial date given by the single element `dates` object *INITIAL_DATE*.

**Constructor: dseries(FILENAME[, INITIAL_DATE][, OPTIONS])**

Instantiates and populates a `dseries` object with a data file specified by *FILENAME*, a string passed as input. Valid file types are `.m`, `.mat`, `.csv`, `.xls` and `.xlsx` (Octave only supports `.xlsx` files and the io package must be installed). The extension of the file should be explicitly provided.

A typical `.m` file will have the following form:

```
FREQ__ = 4;
INIT__ = '1994Q3';
NAMES__ = {'azert';'yuiop'};
TEX__ = {'azert';'yuiop'};
DATA__ = randn(100,2);
```

If a `.mat` file is used instead, it should provide the same information. The `DATA__` matrix of doubles should have as many columns as elements in `NAMES__`. It is possible to omit `DATA__`, in which case there must be variables with the same names as those listed in `NAMES__`, each containing a numeric vector (and those vectors must be of the same length). It is also possible to omit both `NAMES__` and `DATA__`, in which case all existing variables (different from the standardized ones) will be used to construct the dseries object (and those variables must therefore be numeric vectors and all have the same length). The `INIT__` variable can be either a `dates` object or a string which could be used to instantiate the same `dates` object. If `INIT__` is not provided, the initial date is by default set equal to `dates('1Y')`. If the `FREQ__` variable is absent, it is deduced from the value of `INIT__` (or is set to 1 if `INIT__` is absent).

If a `.csv`, `.xls` or `.xlsx` file is used, the first row should be a header containing the variable names. The first column may contain date information that must correspond to a valid date format recognized

by Dynare. If such date information is specified in the first column, its header name must be left empty. Also note that dates should all be of the same frequency (years, months…) and form a consecutive sequence without holes (*e.g.* if the second row of the file corresponds to `2025Q1`, then the third row has to be `2025Q2`, the fourth `2025Q3`, and so on).

Note that the *save* method produces the expected input format for `.m`, and `.mat` and `.csv` files.

If a second input argument is passed to the constructor, `dates` object *INITIAL_DATE*, the initial date defined in *FILENAME* is reset to *INITIAL_DATE*. This is typically useful if `INIT__` is not provided in the data file.

By default, the whole first sheet of a `.xls` or `.xlsx` file is read. Another sheet and a range of cells can be selected with the following name/value pairs, passed after *FILENAME* or after *INITIAL_DATE* (option names are case insensitive):

- `'sheet'`: the name (string) or the index (integer) of the sheet to be read. Default: the first sheet.
- `'range'`: the range of cells to be read, of the form `'B2:D200'` (top-left and bottom-right cells). The first row of the range must contain the names of the variables, and its first column may contain the dates, as described above. Default: the whole sheet.

*Example*

```
>> ts = dseries('data.xlsx', 'sheet', 'Quarterly', 'range', 'B3:D50');
>> ts = dseries('data.xlsx', '2000Q1', 'sheet', 2);
```

**Constructor:** **dseries**(DATA_MATRIX[,INITIAL_DATE[,LIST_OF_NAMES[,TEX_NAMES]]])

**Constructor:** **dseries**(DATA_MATRIX[,RANGE_OF_DATES[,LIST_OF_NAMES[,TEX_NAMES]]])

If the data is not read from a file, it can be provided via a $T \times N$ matrix as the first argument to `dseries` constructor, with $T$ representing the number of observations on $N$ variables. The optional second argument, *INITIAL_DATE*, can be either a `dates` object representing the period of the first observation or a string which would be used to instantiate a `dates` object. Its default value is `dates('1Y')`. The optional third argument, *LIST_OF_NAMES*, is a $N \times 1$ cell of strings with one entry for each variable name. The default name associated with column `i` of *DATA_MATRIX* is `Variable_i`. The final argument, *TEX_NAMES*, is a $N \times 1$ cell of strings composed of the LaTeX names associated with the variables. The default LaTeX name associated with column `i` of *DATA_MATRIX* is `Variable_i`. If the optional second input argument is a range of dates, `dates` object *RANGE_OF_DATES*, the number of rows in the first argument must match the number of elements *RANGE_OF_DATES* or be equal to one (in which case the single observation is replicated).

**Constructor:** **dseries**(TABLE)

Creates a `dseries` object given a MATLAB/Octave `table` object provided as the sole argument. It is assumed that the first column of the table contains the dates of the `dseries` and the first row contains the names. Under Octave, only available if the datatypes packages is installed.

*Example*

Various ways to create a `dseries` object inside a `.mod` file:

```
do1 = dseries(1999Q3);
do2 = dseries('filename.csv');
do3 = dseries([1; 2; 3], 1999Q3, {'var123'}, {'var_{123}'});
```

Equivalent commands at the MATLAB/Octave prompt:

```
>> do1 = dseries(dates('1999Q3'));
>> do2 = dseries('filename.csv');
>> do3 = dseries([1; 2; 3], dates('1999Q3'), {'var123'}, {'var_{123}'});
```

**Constructor:** **dseries**(STRUCTURE[, INITIAL_DATE])

Creates a `dseries` object given a scalar structure *STRUCTURE* holding one vector of observations per variable, the field names giving the names of the variables. All the fields must hold vectors of doubles

of the same length. The optional second argument, `dates` object *INITIAL_DATE* (or a string that could be used to instantiate one), sets the period of the first observation. Its default value is `dates('1Y')`. A structure with no fields returns an empty `dseries` object.

A structure with the six fields returned by `dseries2struct` (`data`, `dates`, `name`, `tex`, `ops` and `tags`) is also accepted, and reinstantiates the `dseries` object it was built from; in that case the second argument is not allowed.

*Example*

```
>> s.Output = transpose(1:5);
>> s.Consumption = transpose(6:10);
>> ts = dseries(s, '1950Q1')

ts is a dseries object:

       | Output | Consumption
1950Q1 | 1      | 6
1950Q2 | 2      | 7
1950Q3 | 3      | 8
1950Q4 | 4      | 9
1951Q1 | 5      | 10
```

One can easily create subsamples from a `dseries` object using the overloaded parenthesis operator. If `ds` is a `dseries` object with $T$ observations and `d` is a `dates` object with $S < T$ elements, such that $\min(d)$ is not smaller than the date associated to the first observation in `ds` and $\max(d)$ is not greater than the date associated to the last observation, then `ds(d)` instantiates a new `dseries` object containing the subsample defined by `d`.

The dot and curly braces selectors described for `extract` below can be combined with this parenthesis operator on the left hand side of an assignment, to overwrite one or several variables over a range of dates:

```
>> ds.GDP(dates('1996Q1'):dates('1998Q4')) = 0;                % a single␣
↪variable
>> ds{'GDP','CPI'}(dates('1996Q1'):dates('1998Q4')) = 0;       % several␣
↪variables at once
```

The value on the right hand side can be a scalar, broadcast over all the selected periods and variables, a vector (a row broadcast over the periods of each selected variable, a column broadcast over the variables), or another `dseries` object, in which case only the periods common to both objects are overwritten.

A list of the available methods, by alphabetical order, is given below. As in the previous section the in place modifications versions of the methods are postfixed with an underscore.

**Method:** **A = abs**(B)

**Method:** **abs_**(B)

Overloads the `abs()` function for `dseries` objects. Returns the absolute value of the variables in `dseries` object `B`.

*Example*

```
>> ts0 = dseries(randn(3,2),'1973Q1',{'A1'; 'A2'},{'A_1'; 'A_2'});
>> ts1 = ts0.abs();
>> ts0

ts0 is a dseries object:

       | A1       | A2
1973Q1 | -0.67284 | 1.4367
1973Q2 | -0.51222 | -0.4948
```

(continues on next page)

(continued from previous page)

```
1973Q3 | 0.99791  | 0.22677

>> ts1

ts1 is a dseries object:

       | abs(A1) | abs(A2)
1973Q1 | 0.67284 | 1.4367
1973Q2 | 0.51222 | 0.4948
1973Q3 | 0.99791 | 0.22677
```

*Example (in-place modification version)*

```
>> ts0 = dseries(randn(3,2),'1973Q1',{'A1'; 'A2'},{'A_1'; 'A_2'});
>> ts0

ts0 is a dseries object:

       | A1       | A2
1973Q1 | -0.67284 | 1.4367
1973Q2 | -0.51222 | -0.4948
1973Q3 | 0.99791  | 0.22677

>> ts0.abs_();
>> ts0

ts0 is a dseries object:

       | abs(A1) | abs(A2)
1973Q1 | 0.67284 | 1.4367
1973Q2 | 0.51222 | 0.4948
1973Q3 | 0.99791 | 0.22677
```

**Method:** **[A, B] = align**(A, B)

**Method:** **align_**(A, B)

If `dseries` objects `A` and `B` are defined on different time ranges, this function extends `A` and/or `B` with NaNs so that they are defined on the same time range. Note that both `dseries` objects must have the same frequency.

*Example*

```
>> ts0 = dseries(rand(5,1),dates('2000Q1')); % 2000Q1 -> 2001Q1
>> ts1 = dseries(rand(3,1),dates('2000Q4')); % 2000Q4 -> 2001Q2
>> [ts0, ts1] = align(ts0, ts1);             % 2000Q1 -> 2001Q2
>> ts0

ts0 is a dseries object:

       | Variable_1
2000Q1 | 0.81472
2000Q2 | 0.90579
2000Q3 | 0.12699
2000Q4 | 0.91338
2001Q1 | 0.63236
2001Q2 | NaN

>> ts1
```

(continues on next page)

(continued from previous page)

```
ts1 is a dseries object:

       | Variable_1
2000Q1 | NaN
2000Q2 | NaN
2000Q3 | NaN
2000Q4 | 0.66653
2001Q1 | 0.17813
2001Q2 | 0.12801

>> ts0 = dseries(rand(5,1),dates('2000Q1')); % 2000Q1 -> 2001Q1
>> ts1 = dseries(rand(3,1),dates('2000Q4')); % 2000Q4 -> 2001Q2
>> align_(ts0, ts1);                         % 2000Q1 -> 2001Q2
>> ts1

ts1 is a dseries object:

       | Variable_1
2000Q1 | NaN
2000Q2 | NaN
2000Q3 | NaN
2000Q4 | 0.66653
2001Q1 | 0.17813
2001Q2 | 0.12801
```

*Example (in-place modification version)*

```
>> ts0 = dseries(rand(5,1),dates('2000Q1')); % 2000Q1 -> 2001Q1
>> ts1 = dseries(rand(3,1),dates('2000Q4')); % 2000Q4 -> 2001Q2
>> ts0

ts0 is a dseries object:

       | Variable_1
2000Q1 | 0.80028
2000Q2 | 0.14189
2000Q3 | 0.42176
2000Q4 | 0.91574
2001Q1 | 0.79221

>> ts1

ts1 is a dseries object:

       | Variable_1
2000Q4 | 0.95949
2001Q1 | 0.65574
2001Q2 | 0.035712

>> align_(ts0, ts1);                         % 2000Q1 -> 2001Q2
>> ts0

ts0 is a dseries object:

       | Variable_1
```

(continues on next page)

(continued from previous page)

```
2000Q1 | 0.80028
2000Q2 | 0.14189
2000Q3 | 0.42176
2000Q4 | 0.91574
2001Q1 | 0.79221
2001Q2 | NaN

>> ts1

ts1 is a dseries object:

       | Variable_1
2000Q1 | NaN
2000Q2 | NaN
2000Q3 | NaN
2000Q4 | 0.95949
2001Q1 | 0.65574
2001Q2 | 0.035712
```

**Method:** **C = backcast**(A, B[, diff])

**Method:** **backcast_**(A, B[, diff])

Backcasts `dseries` object `A` with `dseries` object B's growth rates (except if the last optional argument, `diff`, is true in which case first differences are used). Both `dseries` objects must have the same frequency.

**Method:** **B = baxter_king_filter**(A[, hf[, lf[, K]]])

**Method:** **baxter_king_filter_**(A[, hf[, lf[, K]]])

Implementation of the Baxter and King (1999) band pass filter for `dseries` objects. This filter isolates business cycle fluctuations with a period of length ranging between `hf` (high frequency) to `lf` (low frequency) using a symmetric moving average smoother with $2K + 1$ points, so that $K$ observations at the beginning and at the end of the sample are lost in the computation of the filter. The default value for `hf` is `6`, for `lf` is `32`, and for `K` is `12`.

*Example*

```
% Simulate a component model (stochastic trend, deterministic
% trend, and a stationary autoregressive process).
e = 0.2*randn(200,1);
u = randn(200,1);
stochastic_trend = cumsum(e);
deterministic_trend = .1*transpose(1:200);
x = zeros(200,1);
for i=2:200
    x(i) = .75*x(i-1) + u(i);
end
y = x + stochastic_trend + deterministic_trend;

% Instantiates time series objects.
ts0 = dseries(y,'1950Q1');
ts1 = dseries(x,'1950Q1'); % stationary component.

% Apply the Baxter-King filter.
ts2 = ts0.baxter_king_filter();

% Plot the filtered time series.
plot(ts1(ts2.dates).data,'-k'); % Plot of the stationary component.
hold on
```

(continues on next page)

(continued from previous page)

```
plot(ts2.data,'--r');           % Plot of the filtered y.
hold off
axis tight
id = get(gca,'XTick');
set(gca,'XTickLabel',strings(ts1.dates(id)));
```

**Method:** **B = center**(A[, geometric])

**Method:** **center_**(A[, geometric])

Centers variables in `dseries` object `A` around their arithmetic means, except if the optional argument `geometric` is set equal to `true` in which case all the variables are divided by their geometric means.

**Method:** **C = chain**(A, B)

**Method:** **chain_**(A, B)

Merge two `dseries` objects along the time dimension. The two objects must have the same number of observed variables, and the initial date in `B` must not be posterior to the last date in `A`. The returned `dseries` object, `C`, is built by extending `A` with the cumulated growth factors of `B`.

*Example*

```
>> ts = dseries([1; 2; 3; 4],dates('1950Q1'))

ts is a dseries object:

       | Variable_1
1950Q1 | 1
1950Q2 | 2
1950Q3 | 3
1950Q4 | 4

>> us = dseries([3; 4; 5; 6],dates('1950Q3'))

us is a dseries object:

       | Variable_1
1950Q3 | 3
1950Q4 | 4
1951Q1 | 5
1951Q2 | 6

>> chain(ts, us)

ans is a dseries object:

       | Variable_1
1950Q1 | 1
1950Q2 | 2
1950Q3 | 3
1950Q4 | 4
1951Q1 | 5
1951Q2 | 6
```

*Example (in-place modification version)*

```
>> ts = dseries([1; 2; 3; 4],dates('1950Q1'))
>> us = dseries([3; 4; 5; 6],dates('1950Q3'))
>> ts.chain_(us);
>> ts
```

(continues on next page)

(continued from previous page)

```
ts is a dseries object:

       | Variable_1
1950Q1 | 1
1950Q2 | 2
1950Q3 | 3
1950Q4 | 4
1951Q1 | 5
1951Q2 | 6
```

**Method:** **[error_flag, message] = check**(A)

Sanity check of `dseries` object `A`. Returns `1` if there is an error, `0` otherwise. The second output argument is a string giving brief information about the error.

**Method:** **C = commonop**(A, op, B)

Applies an element-wise binary operation `op` on the variables shared by `dseries` objects `A` and `B`. The returned object `C` holds the variables of `A`, in the same order: a variable shared with `B` is combined with its counterpart in `B`, a variable that is not in `B` is left unchanged, and a variable that is only in `B` is ignored. `A` and `B` are aligned first (see `align`), so that the operation is performed on their common sample, with `NaN`'s where an observation is missing in either object. Both objects must have the same frequency, and share at least one variable.

`op` is either one of the character arrays `'+'`, `'-'` and `'*'` (the latter being the element-wise product, not a matrix product), or a function handle taking two arguments.

*Example*

```
>> ts0 = dseries([1, 2; 3, 4; 5, 6], '1950Q1', {'A1', 'A2'});
>> ts1 = dseries([1; 1; 1], '1950Q1', {'A2'});
>> commonop(ts0, '-', ts1)

ans is a dseries object:

       | A1 | A2
1950Q1 | 1  | 1
1950Q2 | 3  | 3
1950Q3 | 5  | 5

>> commonop(ts0, @rdivide, ts1)

ans is a dseries object:

       | A1 | A2
1950Q1 | 1  | 2
1950Q2 | 3  | 4
1950Q3 | 5  | 6
```

**Method:** **B = copy**(A)

Returns a copy of `A`. If an inplace modification method is applied to `A`, object `B` will not be affected. Note that if `A` is assigned to `C`, `C = A`, then any in place modification method applied to `A` will change `C`.

*Example*

```
>> a = dseries(randn(5,1))

a is a dseries object:
```

(continues on next page)

(continued from previous page)

```
   | Variable_1
1Y | -0.16936
2Y | -1.1451
3Y | -0.034331
4Y | -0.089042
5Y | -0.66997

>> b = copy(a);
>> c = a;
>> a.abs();
>> a.abs_();
>> a

a is a dseries object:

   | Variable_1
1Y | 0.16936
2Y | 1.1451
3Y | 0.034331
4Y | 0.089042
5Y | 0.66997

>> b

b is a dseries object:

   | Variable_1
1Y | -0.16936
2Y | -1.1451
3Y | -0.034331
4Y | -0.089042
5Y | -0.66997

>> c

c is a dseries object:

   | Variable_1
1Y | 0.16936
2Y | 1.1451
3Y | 0.034331
4Y | 0.089042
5Y | 0.66997
```

**Method:** **B = cumprod**(A[, d[, v]])

**Method:** **cumprod_**(A[, d[, v]])

Overloads the MATLAB/Octave `cumprod` function for `dseries` objects. The cumulated product cannot be computed if the variables in `dseries` object `A` have NaNs. If a `dates` object `d` is provided as a second argument, then the method computes the cumulated product with the additional constraint that the variables in the `dseries` object `B` are equal to one in period `d`. If a single-observation `dseries` object `v` is provided as a third argument, the cumulated product in `B` is normalized such that `B(d)` matches `v` (`dseries` objects `A` and `v` must have the same number of variables).

*Example*

```
>> ts1 = dseries(2*ones(7,1));
>> ts2 = ts1.cumprod();
>> ts2

ts2 is a dseries object:

   | cumprod(Variable_1)
1Y | 2
2Y | 4
3Y | 8
4Y | 16
5Y | 32
6Y | 64
7Y | 128

>> ts3 = ts1.cumprod(dates('3Y'));
>> ts3

ts3 is a dseries object:

   | cumprod(Variable_1)
1Y | 0.25
2Y | 0.5
3Y | 1
4Y | 2
5Y | 4
6Y | 8
7Y | 16

>> ts4 = ts1.cumprod(dates('3Y'),dseries(pi));
>> ts4

ts4 is a dseries object:

   | cumprod(Variable_1)
1Y | 0.7854
2Y | 1.5708
3Y | 3.1416
4Y | 6.2832
5Y | 12.5664
6Y | 25.1327
7Y | 50.2655
```

**Method:** **B = cumsum**(A[, d[, v]])

**Method:** **cumsum_**(A[, d[, v]])

Overloads the MATLAB/Octave `cumsum` function for `dseries` objects. The cumulated sum cannot be computed if the variables in `dseries` object `A` have NaNs. If a `dates` object `d` is provided as a second argument, then the method computes the cumulated sum with the additional constraint that the variables in the `dseries` object `B` are zero in period `d`. If a single observation `dseries` object `v` is provided as a third argument, the cumulated sum in `B` is such that `B(d)` matches `v` (`dseries` objects `A` and `v` must have the same number of variables).

*Example*

```
>> ts1 = dseries(ones(10,1));
>> ts2 = ts1.cumsum();
>> ts2
```

(continues on next page)

(continued from previous page)

```
ts2 is a dseries object:

    | cumsum(Variable_1)
1Y  | 1
2Y  | 2
3Y  | 3
4Y  | 4
5Y  | 5
6Y  | 6
7Y  | 7
8Y  | 8
9Y  | 9
10Y | 10

>> ts3 = ts1.cumsum(dates('3Y'));
>> ts3

ts3 is a dseries object:

    | cumsum(Variable_1)
1Y  | -2
2Y  | -1
3Y  | 0
4Y  | 1
5Y  | 2
6Y  | 3
7Y  | 4
8Y  | 5
9Y  | 6
10Y | 7

>> ts4 = ts1.cumsum(dates('3Y'),dseries(pi));
>> ts4

ts4 is a dseries object:

    | cumsum(Variable_1)
1Y  | 1.1416
2Y  | 2.1416
3Y  | 3.1416
4Y  | 4.1416
5Y  | 5.1416
6Y  | 6.1416
7Y  | 7.1416
8Y  | 8.1416
9Y  | 9.1416
10Y | 10.1416
```

**Method:** `B = detrend(A[, m])`

**Method:** `detrend_(A[, m])`

Detrends `dseries` object `A` with a fitted polynomial of order `m`. Default value fir `m` is `0` (time series are detrended by removing the average). Note that each variable is detrended with a different polynomial.

**Method:** `B = dgrowth(A)`

**Method:** `dgrowth_(A)`

Computes daily growth rates.

**Method:** `B = diff(A)`

**Method:** `diff_(A)`

Returns the first difference of `dseries` object `A`.

**Method:** `disp(A[, noname])`

Overloads the MATLAB/Octave `disp` function for `dseries` object. By default the printed table is preceded by a `A is a dseries object:` header, using the name under which `A` was called; passing `true` as the second argument, `noname`, suppresses it. This is useful when displaying an expression rather than a variable, and combines naturally with `subsample` and the curly braces selector of `extract` to print a slice of a `dseries` object over a range of variables and dates:

```
>> ds = dseries([transpose(1:8), transpose(11:18)],'1950Q1',{'Y','C'}
↪);
>> disp(ds{'Y'}(dates('1951Q1'):dates('1951Q4')), true)

       | Y
1951Q1 | 5
1951Q2 | 6
1951Q3 | 7
1951Q4 | 8
```

**Method:** `display(A)`

Overloads the MATLAB/Octave `display` function for `dseries` object. `display` is the function called by MATLAB/Octave to print the content of an object if a semicolon is missing at the end of a MATLAB/Octave statement. If the `dseries` object is defined over a too large time span, only the first and last periods will be printed. If the `dseries` object contains too many variables, only the first and last variables will be printed. If all the periods and variables are required, the `disp` method should be used instead.

**Method:** `C = eq(A, B)`

Overloads the MATLAB/Octave `eq` (equal, ==) operator. The returned argument is a $T \times N$ logical array, where $T$ is the number of observations and $N$ the number of variables. Element $(i, j)$ of `C` is equal to `true` if and only if observation $i$ for variable $j$ in `A` and `B` are equal.

If `A` and `B` are both `dseries` objects, they must have the same frequency, be defined on the same dates and have the same number of variables, or one of them must have only one variable (which is then compared with each variable of the other object); otherwise an error is raised. The names of the variables are not compared (use `isequal` to test whether two `dseries` objects are identical). One of the arguments can also be a scalar, a $1 \times N$ row vector, a $T \times 1$ column vector or a $T \times N$ matrix (of doubles or Booleans), which is compared with the data of the `dseries` object.

*Example*

```
>> ts0 = dseries(2*ones(3,1));
>> ts1 = dseries([2; 0; 2]);
>> ts0==ts1

ans =

   3x1 logical array

    1
    0
    1

>> ts1==2

ans =
```

(continues on next page)

(continued from previous page)

```
   3x1 logical array

    1
    0
    1
```

**Method:** **l = exist**(A, varname)

Tests if variable `varname` exists in `dseries` object `A`. Returns `true` iff variable exists in `A`.

*Example*

```
>> ts = dseries(randn(100,1));
>> ts.exist('Variable_1')

ans =

   logical

    1

>> ts.exist('Variable_2')

ans =

   logical

    0
```

**Method:** **B = exp**(A)

**Method:** **exp_**(A)

Overloads the MATLAB/Octave `exp` function for `dseries` objects.

*Example*

```
>> ts0 = dseries(rand(10,1));
>> ts1 = ts0.exp();
```

*Exemple (in-place modification version)*

```
>> ts0 = dseries(rand(3,1))

ts0 is a dseries object:

   | Variable_1
1Y | 0.82953
2Y | 0.84909
3Y | 0.37253

>> ts0.exp_();
>> ts0

ts0 is a dseries object:

   | Variable_1
1Y | 2.2922
2Y | 2.3375
3Y | 1.4514
```

**Method:** `C = extract(A, B[, ...])`

Extracts some variables from a `dseries` object `A` and returns a `dseries` object `C`. The input arguments following `A` are strings representing the variables to be selected in the new `dseries` object `C`. To simplify the creation of sub-objects, the `dseries` class overloads the curly braces (`D = extract (A, B, C)` is equivalent to `D = A{B,C}`) and allows implicit loops (defined between a pair of `@` symbol, see examples below) or MATLAB/Octave's regular expressions (introduced by square brackets).

*Example*

The following selections are equivalent:

```
>> ts0 = dseries(ones(100,10));
>> ts1 = ts0{'Variable_1','Variable_2','Variable_3'};
>> ts2 = ts0{'Variable_@1,2,3@'};
>> ts3 = ts0{'Variable_[1-3]$'};
>> isequal(ts1,ts2) && isequal(ts1,ts3)

ans =

   logical

    1
```

It is possible to use up to two implicit loops to select variables:

```
names = {'GDP_1';'GDP_2';'GDP_3'; 'GDP_4'; 'GDP_5'; 'GDP_6'; 'GDP_7';
↪'GDP_8'; ...
    'GDP_9'; 'GDP_10'; 'GDP_11'; 'GDP_12'; ...
    'HICP_1';'HICP_2';'HICP_3'; 'HICP_4'; 'HICP_5'; 'HICP_6'; 'HICP_7
↪'; 'HICP_8'; ...
    'HICP_9'; 'HICP_10'; 'HICP_11'; 'HICP_12'};

ts0 = dseries(randn(4,24),dates('1973Q1'),names);
ts0{'@GDP,HICP@_@1,3,5@'}

ans is a dseries object:

       | GDP_1    | GDP_3     | GDP_5     | HICP_1   | HICP_3   |␣
↪HICP_5
1973Q1 | 1.7906   | -1.6606   | -0.57716  | 0.60963  | -0.52335 | 0.
↪26172
1973Q2 | 2.1624   | 3.0125    | 0.52563   | 0.70912  | -1.7158  | 1.
↪7792
1973Q3 | -0.81928 | 1.5008    | 1.152     | 0.2798   | 0.88568  | 1.
↪8927
1973Q4 | -0.03705 | -0.35899  | 0.85838   | -1.4675  | -2.1666  | -0.
↪62032
```

A tilde (~) inside a selector stands for the regular expression \w*, which offers a shorthand way to select every variable sharing a common prefix or suffix:

```
>> ts0 = dseries(ones(10,4),'1950Q1',{'OBS_A','OBS_B','X_OBS_C','ZOBS
↪'});
>> ts0{'OBS_~'}.name

ans =

  2×1 cell array
```

(continues on next page)

(continued from previous page)

```
    {'OBS_A'}
    {'OBS_B'}
```

Note that a selector returns the matching names sorted in alphabetical order, not in the order in which they appear in `A`.

**Method:** **`fill_`**`(name, v)`

Assign the value `v` to the variable `name` in a `dseries` object. If `name` is a character row array, it should correspond to an existing variable within the `dseries` object. When `v` is a scalar, its value will be applied to all periods uniformly. If `v` is a vector, its length must match the number of observations in the `dseries` object. You can invoke this method for a batch of variables by providing a $1 \times n$ cell array of character row arrays as the first argument. When `v` is a row vector with n elements, the method will be applied uniformly across all periods. If `v` is a matrix, it must have $n$ columns, and the number of rows should correspond to the number of periods.

*Example*

```
>> ts = dseries(rand(3,3));
>> ts.fill_({'Variable_1', 'Variable_3'}, [1 3]);
>> ts

ts is a dseries object:

   | Variable_1 | Variable_2 | Variable_3
1Y | 1          | 0.91338    | 3
2Y | 1          | 0.63236    | 3
3Y | 1          | 0.09754    | 3
```

**Method:** **`f = firstdate`**`(A)`

Returns the initial period in the `dseries` object `A`.

**Method:** **`f = firstobservedperiod`**`(A)`

Returns the first period where all the variables in `dseries` object `A` are observed (non NaN).

**Method:** **`B = flip`**`(A)`

**Method:** **`flip_`**`(A)`

Flips the rows in the data member (without changing the periods order).

**Method:** **`f = frequency`**`(B)`

Returns the frequency of the variables in `dseries` object `B`.

*Example*

```
>> ts = dseries(randn(3,2),'1973Q1');
>> ts.frequency

ans =

     4
```

**Method:** **`l = ge`**`(A, B)`

**Method:** **`l = gt`**`(A, B)`

Overloads the `gt` (>) and `ge` (>=) binary operators. Returns a logical array.

See `eq` for the conditions on the arguments.

*Example*

```
>> ts = dseries(randn(3,1))

ts is a dseries object:

   | Variable_1
1Y | -1.2075
2Y | 0.71724
3Y | 1.6302

>> ts>1

ans =

  3x1 logical array

   0
   0
   1

>> ds = dseries(randn(3,1))

ds is a dseries object:

   | Variable_1
1Y | 0.48889
2Y | 1.0347
3Y | 0.72689

>> ds>ts

ans =

  3x1 logical array

   1
   1
   0
```

**Method:** **B = hdiff(A)**

**Method:** **hdiff_(A)**

Computes bi-annual differences.

**Method:** **B = hgrowth(A)**

**Method:** **hgrowth_(A)**

Computes bi-annual growth rates.

**Method:** **D = horzcat(A, B[, ...])**

Overloads the `horzcat` MATLAB/Octave's method for `dseries` objects. Returns a `dseries` object `D` containing the variables in `dseries` objects passed as inputs: `A`, `B`, ... If the inputs are not defined on the same time ranges, the method adds NaNs to the variables so that the variables are redefined on the smallest common time range. Note that the names in the `dseries` objects passed as inputs must be different and these objects must have common frequency.

*Example*

```
>> ts0 = dseries(rand(5,2),'1950Q1',{'nifnif';'noufnouf'});
>> ts1 = dseries(rand(7,1),'1950Q3',{'nafnaf'});
```

(continues on next page)

(continued from previous page)

```
>> ts2 = [ts0, ts1];
>> ts2

ts2 is a dseries object:

       | nifnif  | noufnouf | nafnaf
1950Q1 | 0.17404 | 0.71431  | NaN
1950Q2 | 0.62741 | 0.90704  | NaN
1950Q3 | 0.84189 | 0.21854  | 0.83666
1950Q4 | 0.51008 | 0.87096  | 0.8593
1951Q1 | 0.16576 | 0.21184  | 0.52338
1951Q2 | NaN     | NaN      | 0.47736
1951Q3 | NaN     | NaN      | 0.88988
1951Q4 | NaN     | NaN      | 0.065076
1952Q1 | NaN     | NaN      | 0.50946
```

**Method:** `B = hpcycle(A[, lambda])`

**Method:** `hpcycle_(A[, lambda])`

Extracts the cycle component from a `dseries` `A` object using the Hodrick and Prescott (1997) filter and returns a `dseries` object, `B`. The default value for `lambda`, the smoothing parameter, is `1600`.

*Example*

```
% Simulate a component model (stochastic trend, deterministic
% trend, and a stationary autoregressive process).
e = 0.2*randn(200,1);
u = randn(200,1);
stochastic_trend = cumsum(e);
deterministic_trend = .1*transpose(1:200);
x = zeros(200,1);
for i=2:200
    x(i) = .75*x(i-1) + u(i);
end
y = x + stochastic_trend + deterministic_trend;

% Instantiates time series objects.
ts0 = dseries(y,'1950Q1');
ts1 = dseries(x,'1950Q1'); % stationary component.

% Apply the HP filter.
ts2 = ts0.hpcycle();

% Plot the filtered time series.
plot(ts1(ts2.dates).data,'-k'); % Plot of the stationary component.
hold on
plot(ts2.data,'--r');          % Plot of the filtered y.
hold off
axis tight
id = get(gca,'XTick');
set(gca,'XTickLabel',strings(ts.dates(id)));
```

**Method:** `[trend, cycle] = hpfilter(A, lambda[, ...])`

Extracts the trend and cyclical components of `dseries` object `A`, with one single variable, using the Hodrick and Prescott (1997) filter, with optional linear constraints on the trend or on the cycle. Without constraints this method returns the same components as `hptrend` and `hpcycle`, but bundled in a single call. Unlike `hptrend` and `hpcycle`, it dates its results from the first period of the sample, even if the sample starts with missing observations. Since two outputs are returned, this method must be called

with the functional syntax, `hpfilter(A, ...)`, and not with the dot syntax `A.hpfilter(...)`.

The constraints are passed as name/value pairs, following `lambda`:

- `'C', C, 'd', d`: general linear constraints $C \times trend = d$, with `C` a $m\times$ `nobs(A)` matrix and `d` a $m \times 1$ vector.
- `'trend_idx', idx, 'trend_val', v`: trend anchors, `trend(idx)=v`, where `idx` holds 1 based indices in the sample.
- `'cycle_idx', idx, 'cycle_val', x`: cycle anchors, `cycle(idx)=x`.
- `'trend_dates', D, 'trend_val', v`: trend anchors, where the anchored periods are given by a `dates` object, a date, or a two element cell array `{first, last}` defining a range.
- `'cycle_dates', D, 'cycle_val', x`: cycle anchors, same conventions as `trend_dates`. The default value for `x` is zero.
- `'trend_ds', p`: trend anchors given by a `dseries` object `p` with one single variable, `NaN`'s marking the unconstrained periods.
- `'cycle_ds', p`: cycle anchors, same conventions as `trend_ds`.

The values passed with `trend_val` or `cycle_val` are either scalars, broadcast over all the anchored periods, or vectors with as many elements as anchored periods. All the anchored periods must belong to the sample.

*Example*

Force the cycle to be zero over the first year of a quarterly sample, or to be equal to one in `1990Q1`:

```
>> ts = dseries(cumsum(randn(80,1)), '1985Q1');
>> [trend, cycle] = hpfilter(ts, 1600, 'cycle_dates', {'1985Q1', '1985Q4'},
↪'cycle_val', 0);
>> [trend, cycle] = hpfilter(ts, 1600, 'cycle_dates', dates('1990Q1'),
↪'cycle_val', 1);
```

**Method:** `B = hptrend(A[, lambda])`

**Method:** `hptrend_(A[, lambda])`

Extracts the trend component from a `dseries` A object using the Hodrick and Prescott (1997) filter and returns a `dseries` object, `B`. Default value for `lambda`, the smoothing parameter, is `1600`.

*Example*

```
% Using the same generating data process
% as in the previous example:

ts1 = dseries(stochastic_trend + deterministic_trend,'1950Q1');
% Apply the HP filter.
ts2 = ts0.hptrend();

% Plot the filtered time series.
plot(ts1.data,'-k'); % Plot of the nonstationary components.
hold on
plot(ts2.data,'--r');  % Plot of the estimated trend.
hold off
axis tight
id = get(gca,'XTick');
set(gca,'XTickLabel',strings(ts0.dates(id)));
```

**Method:** `C = insert(A, B, I)`

Inserts variables contained in `dseries` object `B` in `dseries` object `A` at positions specified by integer scalars in vector `I`, returns augmented `dseries` object `C`. The integer scalars in `I` must take values

between 1 and `A.length()+1` and refers to `A` 's column numbers. The `dseries` objects `A` and `B` need not be defined over the same time ranges, but it is assumed that they have common frequency.

*Example*

```
>> ts0 = dseries(ones(2,4),'1950Q1',{'Sly'; 'Gobbo'; 'Sneaky';
↪'Stealthy'});
>> ts1 = dseries(pi*ones(2,1),'1950Q1',{'Noddy'});
>> ts2 = ts0.insert(ts1,3)

ts2 is a dseries object:

       | Sly | Gobbo | Noddy   | Sneaky | Stealthy
1950Q1 | 1   | 1     | 3.1416 | 1      | 1
1950Q2 | 1   | 1     | 3.1416 | 1      | 1

>> ts3 = dseries([pi*ones(2,1) sqrt(pi)*ones(2,1)],'1950Q1',{'Noddy';
↪'Tessie Bear'});
>> ts4 = ts0.insert(ts1,[3, 4])

ts4 is a dseries object:

       | Sly | Gobbo | Noddy   | Sneaky | Tessie Bear | Stealthy
1950Q1 | 1   | 1     | 3.1416 | 1      | 1.7725      | 1
1950Q2 | 1   | 1     | 3.1416 | 1      | 1.7725      | 1
```

**Method:** `B = isempty(A)`

Overloads the MATLAB/Octave's `isempty` function. Returns `true` if `dseries` object `A` is empty.

**Method:** `C = isequal(A, B)`

Overloads the MATLAB/Octave's `isequal` function. Returns `true` if `dseries` objects `A` and `B` are identical.

**Method:** `C = isinf(A)`

Overloads the MATLAB/Octave's `isinf` function. Returns a logical array, with element `(i,j)` equal to `true` if and only if variable `j` is finite in period `A.dates(i)`.

**Method:** `C = isnan(A)`

Overloads the MATLAB/Octave's `isnan` function. Returns a logical array, with element `(i,j)` equal to `true` if and only if variable `j` isn't NaN in period `A.dates(i)`.

**Method:** `C = isreal(A)`

Overloads the MATLAB/Octave's `isreal` function. Returns a logical array, with element `(i,j)` equal to `true` if and only if variable `j` is real in period `A.dates(i)`.

**Method:** `B = lag(A[, p])`

**Method:** `lag_(A[, p])`

Returns lagged time series. Default value of integer scalar `p`, the number of lags, is 1. The `dseries` class overloads the parentheses operator, so that `ts.lag(p)` is equivalent to `ts(-p)`.

*Example*

```
>> ts0 = dseries(transpose(1:4), '1950Q1')

ts0 is a dseries object:

       | Variable_1
1950Q1 | 1
1950Q2 | 2
1950Q3 | 3
```

(continues on next page)

(continued from previous page)

```
1950Q4 | 4

>> ts1 = ts0.lag()

ts1 is a dseries object:

           | Variable_1
    1950Q1 | NaN
    1950Q2 | 1
    1950Q3 | 2
    1950Q4 | 3

>> ts2 = ts0.lag(2)

ts2 is a dseries object:

       | Variable_1
1950Q1 | NaN
1950Q2 | NaN
1950Q3 | 1
1950Q4 | 2

% dseries class overloads the parenthesis
% so that ts.lag(p) can be written more
% compactly as ts(-p). For instance:

>> ts0.lag(1)

ans is a dseries object:

       | Variable_1
1950Q1 | NaN
1950Q2 | 1
1950Q3 | 2
1950Q4 | 3
```

or alternatively:

```
>> ts0(-1)

ans is a dseries object:

       | Variable_1
1950Q1 | NaN
1950Q2 | 1
1950Q3 | 2
1950Q4 | 3
```

**Method:** `l = lastdate(B)`

Retrieves the final period from the `dseries` object `B`.

*Example*

```
>> ts = dseries(randn(3,2),'1973Q1');
>> ts.lastdate()

ans = <dates: 1973Q3>
```

**Method:** `f = lastobservedperiod(A)`

Returns the last period in which all variables of the `dseries` object `A` are fully observed (*i.e.*, contain no NaN values).

**Method:** `f = lastobservedperiods(A)`

Returns the last period without missing observations for each variable in the `dseries` object `A`. The output argument `f` is a structure where each field name corresponds to a variable in `A`, and the content of each field is a singleton `date` object.

**Method:** `l = le(A, B)`

**Method:** `l = lt(A, B)`

Overloads the `lt` (<) and `le` (<=) binary operators. Returns a logical array.

See `eq` for the conditions on the arguments.

*Example*

```
>> ts = dseries(randn(3,1))

ts is a dseries object:

   | Variable_1
1Y | -1.2075
2Y | 0.71724
3Y | 1.6302

>> ts<1

ans =

  3x1 logical array

   1
   1
   0

>> ds = dseries(randn(3,1))

ds is a dseries object:

   | Variable_1
1Y | 0.48889
2Y | 1.0347
3Y | 0.72689

>> ds<ts

ans =

  3x1 logical array

   0
   0
   1
```

**Method:** `B = lead(A[, p])`

**Method:** `lead_(A[, p])`

Returns a lead time series. The default value for the integer scalar `p`, which represents the number of

leads, is 1. Similar to the `lag` method, the `dseries` class overloads the parentheses operator, making `ts.lead(p)` equivalent to `ts(p)`.

*Example*

```
>> ts0 = dseries(transpose(1:4),'1950Q1');
>> ts1 = ts0.lead()

ts1 is a dseries object:

       | Variable_1
1950Q1 | 2
1950Q2 | 3
1950Q3 | 4
1950Q4 | NaN

>> ts2 = ts0(2)

ts2 is a dseries object:

       | Variable_1
1950Q1 | 3
1950Q2 | 4
1950Q3 | NaN
1950Q4 | NaN
```

*Remark*

The overload of parentheses for `dseries` objects simplifies the creation of new `dseries` instances by enabling the direct copying and pasting of equations defined within the `model` block. For example, if an Euler equation is specified in the `model` block,:

```
model;
...
1/C - beta/C(1)*(exp(A(1))*K^(alpha-1)+1-delta) ;
...
end;
```

and if variables `C`, `A` and `K` are defined as `dseries` objects, then by writing:

```
Residuals = 1/C - beta/C(1)*(exp(A(1))*K^(alpha-1)+1-delta) ;
```

outside of the `model` block, we create a new `dseries` object, called `Residuals`, for the residuals of the Euler equation (the conditional expectation of the equation defined in the `model` block is zero, but the residuals are non zero).

**Method: `B = lineartrend(A)`**

Returns a linear trend centered on 0, the length of the trend is given by the size of `dseries` object `A` (the number of periods).

*Example*

```
>> ts = dseries(ones(3,1));
>> ts.lineartrend()

ans =

     -1
      0
      1
```

**Method:** `B = log(A)`

**Method:** `log_(A)`

Overloads the MATLAB/Octave `log` function for `dseries` objects.

*Example*

```
>> ts0 = dseries(rand(10,1));
>> ts1 = ts0.log();
```

**Method:** `B = mdiff(A)`

**Method:** `mdiff_(A)`

Calculates the monthly differences of variables in the `dseries` object `A`.

**Method:** `B = mgrowth(A)`

**Method:** `mgrowth_(A)`

Calculates the monthly growth rates of variables in the `dseries` object `A`.

**Method:** `B = mean(A[, geometric])`

This function overloads the MATLAB/Octave `mean` function specifically for `dseries` objects. It calculates the mean for each variable within the `dseries` object `A`. If the second argument is set to `true`, the geometric mean is calculated; otherwise, the arithmetic mean is computed by default.

**Method:** `C = merge(A, B[, legacy])`

Merges two `dseries` objects, `A` and `B`, into a new `dseries` object `C`. The objects `A` and `B` must share a common frequency, although they can cover different time ranges. If a variable, such as `x`, exists in both `dseries` objects, the `merge` function will prioritize the definition from the second input, `B`, while retaining the values from `A` for any corresponding periods where `B` has NaN values. This behavior can be altered by setting the optional argument `legacy` to true, in which case the second variable will replace the first, even if it contains NaN values.

*Example*

```
>> ts0 = dseries(rand(3,2),'1950Q1',{'A1';'A2'})

ts0 is a dseries object:

       | A1      | A2
1950Q1 | 0.96284 | 0.5363
1950Q2 | 0.25145 | 0.31866
1950Q3 | 0.34447 | 0.4355

>> ts1 = dseries(rand(3,1),'1950Q2',{'A1'})

ts1 is a dseries object:

       | A1
1950Q2 | 0.40161
1950Q3 | 0.81763
1950Q4 | 0.97769

>> merge(ts0,ts1)

ans is a dseries object:

       | A1      | A2
1950Q1 | 0.96284 | 0.5363
1950Q2 | 0.40161 | 0.31866
1950Q3 | 0.81763 | 0.4355
```

(continues on next page)

(continued from previous page)

```
1950Q4 | 0.97769 | NaN

 >> merge(ts1,ts0)

 ans is a dseries object:

       | A1      | A2
1950Q1 | 0.96284 | 0.5363
1950Q2 | 0.25145 | 0.31866
1950Q3 | 0.34447 | 0.4355
1950Q4 | 0.97769 | NaN
```

**Method:** `C = minus(A, B)`

Overloads the MATLAB/Octave `minus` (-) operator for `dseries` objects, allowing for element-by-element subtraction. When both `A` and `B` are `dseries` objects, they do not need to be defined over the same time ranges. If `A` and `B` have $T_A$ and $T_B$ observations and $N_A$ and $N_B$ variables, then $N_A$ must equal $N_B$ or 1, and $N_B$ must equal $N_A$ or 1. If $T_A = T_B$, `isequal(A.init,B.init)` returns `true`, and $N_A = N_B$, then the `minus` operator will compute for each pair $(t, n)$, where $1 \leq t \leq T_A$ and $1 \leq n \leq N_A$, the operation `C.data(t,n)=A.data(t,n)-B.data(t,n)`. If $N_B$ equals 1 and $N_A > 1$, the smaller `dseries` object (`B`) is "broadcasted" across the larger `dseries` (`A`), ensuring compatible shapes for the subtraction of the variable defined in `B` from each variable in `A`. If `B` is a double scalar, the `minus` method will subtract `B` from all observations and variables in `A`. If `B` is a row vector of length $N_A$, the `minus` method will subtract `B(i)` from all observations of variable `i`, for $i = 1, ..., N_A$. If `B` is a column vector of length $T_A$, the `minus` method will subtract `B` from all the variables.

*Example*

```
>> ts0 = dseries(rand(3,2));
>> ts1 = ts0{'Variable_2'};
>> ts0-ts1

ans is a dseries object:

   | Variable_1 | Variable_2
1Y | -0.48853   | 0
2Y | -0.50535   | 0
3Y | -0.32063   | 0

>> ts1

ts1 is a dseries object:

   | Variable_2
1Y | 0.703
2Y | 0.75415
3Y | 0.54729

>> ts1-ts1.data(1)

ans is a dseries object:

   | Variable_2
1Y | 0
2Y | 0.051148
3Y | -0.15572
```

(continues on next page)

(continued from previous page)

```
>> ts1.data(1)-ts1

ans is a dseries object:

   | Variable_2
1Y | 0
2Y | -0.051148
3Y | 0.15572
```

**Method:** `C = mpower(A, B)`

Overloads the MATLAB/Octave `mpower` (`^`) operator for `dseries` objects, performing element-wise exponentiation. Given a `dseries` object `A` with `N` variables and `T` observations, if `B` is a real scalar, then `mpower(A,B)` yields a `dseries` object `C` where `C.data(t,n) = A.data(t,n)^B`. If `B` is also a `dseries` object with `N` variables and `T` observations, then `mpower(A,B)` produces a `dseries` object `C` such that `C.data(t,n) = A.data(t,n)^{C.data(t,n)}`.

*Example*

```
>> ts0 = dseries(transpose(1:3));
>> ts1 = ts0^2

ts1 is a dseries object:

   | Variable_1
1Y | 1
2Y | 4
3Y | 9

>> ts2 = ts0^ts0

ts2 is a dseries object:

   | Variable_1
1Y | 1
2Y | 4
3Y | 27
```

**Method:** `C = mrdivide(A, B)`

Overloads the MATLAB/Octave `mrdivide` (`/`) operator for `dseries` objects, enabling element-wise division similar to the `./` operator in MATLAB/Octave. When both `A` and `B` are `dseries` objects, they can have different time ranges. If `A` contains $T_A$ observations and $N_A$ variables, and `B` has $T_B$ observations and $N_B$ variables, then $N_A$ must equal $N_B$ or 1, and vice versa. If $T_A = T_B$ and `isequal(A.init,B.init)` returns `true`, along with $N_A = N_B$, the `mrdivide` operator calculates for each pair $(t, n)$, where $1 \leq t \leq T_A$ and $1 \leq n \leq N_A$, the value of `C.data(t,n)=A.data(t,n)/B.data(t,n)`. If $N_B$ equals 1 and $N_A > 1$, the smaller `dseries` object (`B`) is "broadcast" across the larger one (`A`) to ensure compatible shapes. In this case, the `mrdivide` operator divides each variable in `A` by the variable in `B`, observation by observation. If `B` is a double scalar, then `mrdivide` will divide all observations and variables in `A` by `B`. If `B` is a row vector of length $N_A$, then `mrdivide` will divide each observation of variable `i` by `B(i)`, for $i = 1, ..., N_A$. If `B` is a column vector of length $T_A$, then `mrdivide` will perform an element-wise division of all variables by `B`.

*Example*

```
>> ts0 = dseries(rand(3,2))

ts0 is a dseries object:
```

(continues on next page)

(continued from previous page)

```
   | Variable_1 | Variable_2
1Y | 0.72918    | 0.90307
2Y | 0.93756    | 0.21819
3Y | 0.51725    | 0.87322

>> ts1 = ts0{'Variable_2'};
>> ts0/ts1

ans is a dseries object:

   | Variable_1 | Variable_2
1Y | 0.80745    | 1
2Y | 4.2969     | 1
3Y | 0.59235    | 1
```

**Method:** `C = mtimes(A, B)`

Overloads the MATLAB/Octave `mtimes` (*) operator for `dseries` objects, enabling element-wise multiplication similar to the `.*` operator in MATLAB/Octave. When both `A` and `B` are `dseries` objects, they can have different time ranges. If `A` contains $T_A$ observations and $N_A$ variables, and `B` has $T_B$ observations and $N_B$ variables, then $N_A$ must equal $N_B$ or 1, and vice versa. If $T_A = T_B$ and `isequal(A.init,B.init)` returns `true`, along with $N_A = N_B$, the `mtimes` operator calculates for each pair $(t, n)$, where $1 \leq t \leq T_A$ and $1 \leq n \leq N_A$, the value of `C.data(t,n)=A.data(t,n)*B.data(t,n)`. If $N_B$ equals 1 and $N_A > 1$, the smaller `dseries` object (`B`) is "broadcasted" across the larger one (`A`) to ensure compatible shapes. In this case, the `mtimes` operator multiply each variable in `A` by the variable in `B`, observation by observation. If `B` is a double scalar, then `mtimes` will multiply all observations and variables in `A` by `B`. If `B` is a row vector of length $N_A$, then `mtimes` will multiply each observation of variable `i` by `B(i)`, for $i = 1, ..., N_A$. If `B` is a column vector of length $T_A$, then `mtimes` will perform an element-wise multiplication of all variables by `B`.

**Method:** `B = nanmean(A[, geometric])`

Overloads the MATLAB/Octave `nanmean` function for `dseries` objects. Computes the mean of each variable in the `dseries` object `A`, excluding NaN values. If the second argument is `true`, the geometric mean is calculated; otherwise, the default is to report the arithmetic mean.

**Method:** `B = nanstd(A[, geometric])`

Overloads the MATLAB/Octave `nanstd` function for `dseries` objects. This function calculates the standard deviation for each variable within the `dseries` object `A`, while disregarding any NaN values. If the second argument is set to `true`, the geometric standard deviation will be computed; the default value for the second argument is `false`.

**Method:** `C = ne(A, B)`

Overloads the MATLAB/Octave `ne` (not equal, ~=) operator. The returned argument is a $T \times N$ logical array, where $T$ is the number of observations and $N$ the number of variables. Element $(i, j)$ of `C` is equal to `true` if and only if observation $i$ for variable $j$ in `A` and `B` are not equal.

If `A` and `B` are both `dseries` objects, they must have the same frequency, be defined on the same dates and have the same number of variables, or one of them must have only one variable (which is then compared with each variable of the other object); otherwise an error is raised. The names of the variables are not compared (use `isequal` to test whether two `dseries` objects are identical). One of the arguments can also be a scalar, a $1 \times N$ row vector, a $T \times 1$ column vector or a $T \times N$ matrix (of doubles or Booleans), which is compared with the data of the `dseries` object.

*Example*

```
>> ts0 = dseries(2*ones(3,1));
>> ts1 = dseries([2; 0; 2]);
>> ts0~=ts1
```

(continues on next page)

(continued from previous page)

```
ans =

  3x1 logical array

   0
   1
   0
```

**Method:** **B = nobs**(A)

Returns the number of observations in `dseries` object `A`.

*Example*

```
>> ts0 = dseries(randn(10));
>> ts0.nobs

ans =

    10
```

**Method:** **B = onesidedhpcycle**(A[, lambda[, init]])

**Method:** **onesidedhpcycle_**(A[, lambda[, init]])

Extracts the cycle component from a `dseries` object `A` using a one-sided HP filter (implemented with a Kalman filter) and returns a `dseries` object, `B`. The default value for `lambda`, the smoothing parameter, is set to `1600`. By default, if `init` is not provided, the initial value is determined from the first two observations.

**Method:** **B = onesidedhptrend**(A[, lambda[, init]])

**Method:** **onesidedhptrend_**(A[, lambda[, init]])

Extracts the trend component from a `dseries` object `A` using a one-sided HP filter (implemented with a Kalman filter) and returns a `dseries` object, `B`. The default value for `lambda`, the smoothing parameter, is set to `1600`. By default, if `init` is not provided, the initial value is derived from the first two observations.

**Method:** **h = plot**(A)

**Method:** **h = plot**(A, B)

**Method:** **h = plot**(A[, ...])

**Method:** **h = plot**(A, B[, ...])

Overloads the MATLAB/Octave `plot` function for `dseries` objects. This function returns a MATLAB/Octave plot handle, which can be utilized to modify the properties of the plotted time series. If a single `dseries` object, `A`, is provided as an argument, the plot function will place the corresponding dates on the x-axis. If this `dseries` object contains only one variable, additional arguments can be included to adjust the plot properties, similar to how one would with MATLAB/Octave's original plot function. However, if the `dseries` object `A` has more than one variable, additional arguments cannot be passed, and modifications to the plotted time series properties must be done using the returned plot handle alongside the MATLAB/Octave `set` function (refer to the example below). When two `dseries` objects, `A` and `B`, are passed as input arguments, the plot function will display the variables in `A` against those in `B` (it is essential that both objects contain the same number of variables; otherwise, an error will occur). Once more, if each object includes only one variable, additional arguments can be utilized to alter the plotted time series properties; otherwise, the MATLAB/Octave `set` command must be employed.

*Example*

Define a `dseries` object with two variables (named by default `Variable_1` and `Variable_2`):

```
>> ts = dseries(randn(100,2),'1950Q1');
```

The following command will plot the first variable in `ts`:

```
>> plot(ts{'Variable_1'},'-k','linewidth',2);
```

The next command will draw all the variables in `ts` on the same figure:

```
>> h = plot(ts);
```

If one wants to modify the properties of the plotted time series (line style, colours, ...), the set function can be used (see MATLAB/Octave's documentation):

```
>> set(h(1),'-k','linewidth',2);
>> set(h(2),'--r');
```

The following command will plot `Variable_1` against `exp(Variable_1)`:

```
>> plot(ts{'Variable_1'},ts{'Variable_1'}.exp(),'ok');
```

Again, the properties can also be modified using the returned plot handle and the `set` function:

```
>> h = plot(ts, ts.exp());
>> set(h(1),'ok');
>> set(h(2),'+r');
```

**Method:** `C = plus(A, B)`

Overloads the MATLAB/Octave `plus` (+) operator for `dseries` objects, allowing for element-wise addition. When both `A` and `B` are `dseries` objects, they do not need to be defined over the same time ranges. If `A` and `B` are `dseries` objects with $T_A$ and $T_B$ observations and $N_A$ and $N_B$ variables, then $N_A$ must be equal to $N_B$ or 1 and $N_B$ must be equal to $N_A$ or 1. If $T_A = T_B$, `isequal(A.init, B.init)` returns `true` and $N_A = N_B$, then the `plus` operator will compute for each pair $(t, n)$, with $1 \leq t \leq T_A$ and $1 \leq n \leq N_A$, `C.data(t,n)=A.data(t,n)+B.data(t,n)`. If $N_B$ is equal to 1 and $N_A > 1$, the smaller `dseries` object (`B`) is "broadcasted" across the larger `dseries` (`A`) to ensure compatible shapes, the plus operator will add the variable defined in `B` to each variable in `A`. If `B` is a double scalar, then the method `plus` will add `B` to all the observations/variables in `A`. If `B` is a row vector of length $N_A$, then the `plus` method will add `B(i)` to all the observations of variable `i`, for $i = 1, \ldots, N_A$. If `B` is a column vector of length $T_A$, then the `plus` method will add `B` to all the variables.

**Method:** `C = pop(A[, B])`

**Method:** `pop_(A[, B])`

Removes the variable `B` from the `dseries` object `A`. By default, if the second argument is not specified, the last variable is removed.

*Example*

```
>> ts0 = dseries(ones(3,3));
>> ts1 = ts0.pop('Variable_2');

ts1 is a dseries object:

   | Variable_1 | Variable_3
1Y | 1          | 1
2Y | 1          | 1
3Y | 1          | 1
```

**Method:** `B = prefix(A, p)`

**Method:** **prefix_**(A, p)

Prepends a prefix p, a non empty row character array, to the name of each variable of the dseries object A. The TeX names are given the same prefix, escaped with name2tex, unless a TeX name was explicitly set by the user, in which case the prefix is simply prepended to it. The ops member is left unchanged, as it is by rename_.

*Example*

```
>> ts0 = dseries(randn(10,2), '1950Q1', {'GDP', 'CPI'});
>> ts1 = ts0.prefix('OBS_');
>> ts1.name

ans =

  2×1 cell array

    {'OBS_GDP'}
    {'OBS_CPI'}
```

**Method:** **A = projection**(A, info, periods)

Projects variables in the dseries object A. The info variable is a $n \times 3$ cell array, where each row contains essential information for projecting a variable. The first column holds the variable name (as a character array), while the second column indicates the projection method used (also a character array). The possible values for this column are 'Trend', 'Constant', and 'AR'. The third column provides quantitative details related to the projection: if the second column is 'Trend', the third column specifies the growth factor of the (exponential) trend; if 'Constant', it indicates the variable's level; and if 'AR', it denotes the autoregressive parameter. Variables can be projected using an AR($p$) model if the third column contains a $1 \times p$ vector of doubles. Note that the stationarity of the AR($p$) model is not tested. For constant projections, one can use either 'Trend' with a growth factor of 1 or 'AR' with an autoregressive parameter of one (indicating a random walk). This projection routine solely addresses exponential trends.

*Example*

```
>> data = ones(10,4);
>> ts = dseries(data, '1990Q1', {'A1', 'A2', 'A3', 'A4'});
>> info = {'A1', 'Trend', 1.2; 'A2', 'Constant', 0.0; 'A3', 'AR', .5;
↪'A4', 'AR', [.4, -.2]};
>> ts.projection(info, 10);
```

**Method:** **B = qdiff**(A)

**Method:** **qdiff_**(A)

Computes quarterly differences.

*Example*

```
>> ts0 = dseries(transpose(1:4),'1950Q1');
>> ts1 = ts0.qdiff()

ts1 is a dseries object:

       | Variable_1
1950Q1 | NaN
1950Q2 | 1
1950Q3 | 1
1950Q4 | 1

>> ts0 = dseries(transpose(1:6),'1950M1');
>> ts1 = ts0.qdiff()
```

(continues on next page)

(continued from previous page)

```
ts1 is a dseries object:

        | Variable_1
1950M1  | NaN
1950M2  | NaN
1950M3  | NaN
1950M4  | 3
1950M5  | 3
1950M6  | 3
```

**Method:** `B = qgrowth(A)`

**Method:** `qgrowth_(A)`

Computes quarterly growth rates.

**Method:** `C = remove(A, B)`

**Method:** `remove_(A, B)`

If `B` is a row character array representing the name of a variable, these methods serve as aliases for the `pop` and `pop_` methods that accept two arguments. They remove the variable `B` from the `dseries` object `A`. To remove multiple variables, you can pass a cell array of row character arrays for `B`.

*Example*

```
>> ts0 = dseries(ones(3,3));
>> ts1 = ts0.remove('Variable_2');

ts1 is a dseries object:

   | Variable_1 | Variable_3
1Y | 1          | 1
2Y | 1          | 1
3Y | 1          | 1
```

A more concise syntax is available: `remove(ts, 'Variable_2')`, which is equivalent to `ts{'Variable_2'} = []` (where `[]` can be substituted with any empty object). This alternative syntax proves useful when removing multiple variables. For example:

```
ts{'Variable_@2,3,4@'} = [];
```

will remove `Variable_2`, `Variable_3` and `Variable_4` from `dseries` object `ts` (if these variables exist). Regular expressions cannot be used but implicit loops can.

**Method:** `B = rename(A, oldname, newname)`

**Method:** `rename_(A, oldname, newname)`

Renames the variable `oldname` to `newname` in the `dseries` object `A`. This function returns a `dseries` object. If multiple variables need to be renamed, you can provide cell arrays of row character arrays as the second and third arguments.

*Example*

```
>> ts0 = dseries(ones(2,2));
>> ts1 = ts0.rename('Variable_1','Stinkly')

ts1 is a dseries object:

   | Stinkly | Variable_2
1Y | 1       | 1
2Y | 1       | 1
```

**Method:** `C = rename(A, newname)`

**Method:** `rename_(A, newname)`

Replace the names in `A` with those specified in the cell of row character arrays `newname`. The cell `newname` must contain the same number of elements as there are variables in the `dseries` object `A`.

*Example*

```
>> ts0 = dseries(ones(2,3));
>> ts1 = ts0.rename({'TinkyWinky','Dipsy','LaaLaa'})

ts1 is a dseries object:

   | TinkyWinky | Dipsy | LaaLaa
1Y | 1          | 1     | 1
2Y | 1          | 1     | 1
```

**Method:** `A = resetops(A, ops)`

Redefine `ops` member.

**Method:** `A = resettags(A, tags)`

Redefine `tags` member.

**Method:** `B = round(A[, n])`

**Method:** `round_(A[, n])`

Rounds each value to the nearest decimal or integer. The parameter `n` specifies the precision (number of decimal places), with a default value of `0`, indicating that the method will round to the nearest integer by default.

*Example*

```
>> ts = dseries(pi)

ts is a dseries object:

   | Variable_1
1Y | 3.1416

>> ts.round_();
>> ts

ts is a dseries object:

   | Variable_1
1Y | 3
```

**Method:** `save(A, basename[, format])`

Overloads the MATLAB/Octave `save` function to save the `dseries` object `A` to disk. The available formats include `mat` (default, MATLAB/Octave binary data file), `m` (MATLAB/Octave script), and `csv` (comma-separated values file). The base name of the file, excluding the extension, is specified by `basename`.

*Example*

```
>> ts0 = dseries(ones(2,2));
>> ts0.save('ts0', 'csv');
```

The last command will create a file `ts0.csv` with the following content:

```
,Variable_1,Variable_2
1Y,                1,                1
2Y,                1,                1
```

To create a MATLAB/Octave script, the following command:

```
>> ts0.save('ts0','m');
```

will produce a file `ts0.m` with the following content:

```
% File created on 14-Nov-2013 12:08:52.

FREQ__ = 1;
INIT__ = ' 1Y';

NAMES__ = {'Variable_1'; 'Variable_2'};
TEX__ = {'Variable_{1}'; 'Variable_{2}'};
OPS__ = {};
TAGS__ = struct();

Variable_1 = [
               1
               1];

Variable_2 = [
               1
               1];
```

The generated (`csv`, `m`, or `mat`) files can be loaded when instantiating a `dseries` object as explained above.

**Method:** **`B = set_names(A, s1, s2, ...)`**

Renames the variables in the `dseries` object `A` and returns a new `dseries` object `B` with the updated names `s1`, `s2`, and so forth. The number of input arguments following the first one (the `dseries` object `A`) must be equal to `A.vobs` (the total number of variables in `A`). The name `s1` will correspond to the first variable in `B`, `s2` to the second variable in `B`, and this pattern continues for the remaining variables.

*Example*

```
>> ts0 = dseries(ones(1,3));
>> ts1 = ts0.set_names('Barbibul',[],'Barbouille')

ts1 is a dseries object:

   | Barbibul | Variable_2 | Barbouille
1Y | 1        | 1          | 1
```

**Method:** **`[T, N] = size(A[, dim])`**

Overloads the MATLAB/Octave `size` function to return the number of observations in the `dseries` object `A` (*i.e.*, `A.nobs`) as well as the number of variables (*i.e.*, `A.vobs`). If a second input argument is provided, the `size` function will return the number of observations when `dim` equals `1` or the number of variables when `dim` equals `2`. An error will be issued for any other values of `dim`.

*Example*

```
>> ts0 = dseries(ones(1,3));
>> ts0.size()

ans =
```

(continues on next page)

(continued from previous page)

```
    1     3
```

**Method:** **B = std(A[, geometric])**

Overloads the MATLAB/Octave `std` function for `dseries` objects. This function returns the standard deviation of each variable within the `dseries` object `A`. If the second argument is set to `true`, the geometric standard deviation is calculated (the default value for the second argument is `false`).

**Method:** **B = subsample(A, d1, d2)**

Returns a subsample for the period between `d1` and `d2`. While you can achieve the same result by indexing a `dseries` object with a `dates` object, the `subsample` method offers a more straightforward approach for programmatic use.

*Example*

```
>> o = dseries(transpose(1:5));
>> o.subsample(dates('2y'),dates('4y'))

ans is a dseries object:

   | Variable_1
2Y | 2
3Y | 3
4Y | 4
```

**Method:** **B = suffix(A, s)**

**Method:** **suffix_(A, s)**

Appends a suffix `s`, a non empty row character array, to the name of each variable of the `dseries` object `A`. The TeX names are given the same suffix, escaped with `name2tex`, unless a TeX name was explicitly set by the user, in which case the suffix is simply appended to it. The `ops` member is left unchanged, as it is by `rename_`. A prefix and a suffix can be combined.

*Example*

```
>> ts0 = dseries(randn(10,2), '1950Q1', {'GDP', 'CPI'});
>> ts1 = ts0.suffix('_sa');
>> ts1.name

ans =

  2×1 cell array

    {'GDP_sa'}
    {'CPI_sa'}
```

**Method:** **A = tag(A, a[, b, c])**

Adds a tag to a variable in `dseries` object `A`.

*Example*

```
>> ts = dseries(randn(10, 3));
>> tag(ts, 'type');              % Define a tag name.
>> tag(ts, 'type', 'Variable_1', 'Stock');
>> tag(ts, 'type', 'Variable_2', 'Flow');
>> tag(ts, 'type', 'Variable_3', 'Stock');
```

**Method:** **B = tex_rename(A, name, newtexname)**

**Method:** **B = tex_rename(A, newtexname)**

**Method:** **tex_rename_**(A, name, newtexname)

**Method:** **tex_rename_**(A, newtexname)

Updates the TeX name of the variable `name` to `newtexname` in the `dseries` object `A`. Returns an updated `dseries` object.

With just two arguments, `A` and `newtexname`, this function redefines the TeX names of the entries in `A` to those specified in `newtexname`. The `newtexname` argument must be a cell row character arrays containing the same number of entries as there are variables in `A`.

**Method:** **B = transform**(A, varlist, fun, pfx)

**Method:** **transform_**(A, varlist, fun, pfx)

Applies function `fun` to a subset `varlist` of the variables of the `dseries` object `A`, and appends the results, under prefixed names, to the object. `varlist` is a row character array or a cell array of row character arrays holding the names of the variables to be transformed. `fun` is a function handle applied to each variable of `varlist` separately, taking a `dseries` object with one single variable and returning a `dseries` object. If `fun` returns an object with one single variable, that variable is named `pfx` followed by the name of the transformed variable. If it returns more than one variable, each of them is renamed by prepending `pfx` to its own name. An error is raised if the resulting names are not distinct or clash with the names already present in `A`.

*Example*

```
>> ts0 = dseries(rand(10,2)+1, '1950Q1', {'CPI', 'CPIX'});
>> ts1 = ts0.transform({'CPI', 'CPIX'}, @(x) 100*log(x), 'L_');
>> ts1.name

ans =

  4×1 cell array

    {'CPI'   }
    {'CPIX'  }
    {'L_CPI' }
    {'L_CPIX'}
```

**Method:** **B = uminus**(A)

Overloads the `uminus` operator (-, unary minus) for the `dseries` object.

*Example*

```
>> ts0 = dseries(1)

ts0 is a dseries object:

   | Variable_1
1Y | 1

>> ts1 = -ts0

ts1 is a dseries object:

   | Variable_1
1Y | -1
```

**Method:** **D = vertcat**(A, B[, ...])

Overloads the `vertcat` method in MATLAB/Octave for `dseries` objects. This method facilitates the appending of additional observations to a `dseries` object. It returns a new `dseries` object, `D`, which contains the variables from the input `dseries` objects. All input arguments must be `dseries` objects that share the same variables but are defined over different time ranges.

*Example*

```
>> ts0 = dseries(rand(2,2),'1950Q1',{'nifnif';'noufnouf'});
>> ts1 = dseries(rand(2,2),'1950Q3',{'nifnif';'noufnouf'});
>> ts2 = [ts0; ts1]

ts2 is a dseries object:

       | nifnif   | noufnouf
1950Q1 | 0.82558  | 0.31852
1950Q2 | 0.78996  | 0.53406
1950Q3 | 0.089951 | 0.13629
1950Q4 | 0.11171  | 0.67865
```

**Method:** `B = vobs(A)`

Returns the count of variables in the `dseries` object `A`.

*Example*

```
>> ts0 = dseries(randn(10,2));
>> ts0.vobs

ans =

    2
```

**Method:** `B = ydiff(A)`

**Method:** `ydiff_(A)`

Calculates annual differences.

**Method:** `B = ygrowth(A)`

**Method:** `ygrowth_(A)`

Calculates annual growth rates.

## 6.3 X-13 ARIMA-SEATS interface

**Dynare class:** `x13`

The x13 class provides a method for each X-13 command as documented in the X-13 ARIMA-SEATS reference manual (`x11`, `automdl`, `estimate`, ...). The respective options (see Chapter 7 of U.S. Census Bureau (2020)) can then be passed by key/value pairs. The `x13` class has 22 members:

**Members**

- **`y`** – `dseries` object with a single variable.
- **`x`** – `dseries` object with an arbitrary number of variables (to be used in the REGRESSION block).
- **`arima`** – structure containing the options of the ARIMA model command.
- **`automdl`** – structure containing the options of the ARIMA model selection command.
- **`regression`** – structure containing the options of the Regression command.
- **`estimate`** – structure containing the options of the estimation command.
- **`transform`** – structure containing the options of the transform command.
- **`outlier`** – structure containing the options of the outlier command.
- **`forecast`** – structure containing the options of the forecast command.
- **`check`** – structure containing the options of the check command.

- **x11** – structure containing the options of the X11 command.
- **force** – structure containing the options of the force command.
- **history** – structure containing the options of the history command.
- **metadata** – structure containing the options of the metadata command.
- **identify** – structure containing the options of the identify command.
- **pickmdl** – structure containing the options of the pickmdl command.
- **seats** – structure containing the options of the seats command.
- **slidingspans** – structure containing the options of the slidingspans command.
- **spectrum** – structure containing the options of the spectrum command.
- **x11regression** – structure containing the options of the x11Regression command.
- **results** – structure containing the results returned by x13.
- **commands** – cell array containing the list of commands.

All these members are private. The following constructors are available:

**Constructor:** **x13**(y)

Instantiates an x13 object with dseries object y. The dseries object passed as an argument must contain only one variable, the one we need to pass to X-13.

**Constructor:** **x13**(y, x)

Instantiates an x13 object with dseries objects y and x. The first dseries object passed as an argument must contain only one variable, the second dseries object contains the exogenous variables used by some of the X-13 commands. Both objects must be defined on the same time span.

The following methods allow to set sequence of X-13 commands, write an .spc file, and run the X-13 binary:

**Method:** **A = arima**(A, key, value[, key, value[, [...]]])

Interface to the arima command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** **A = automdl**(A, key, value[, key, value[, [...]]])

Interface to the automdl command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** **A = regression**(A, key, value[, key, value[, [...]]])

Interface to the regression command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** **A = estimate**(A, key, value[, key, value[, [...]]])

Interface to the estimate command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** **A = transform**(A, key, value[, key, value[, [...]]])

Interface to the transform command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs. For example, the key/value pair function,log instructs the use of a multiplicative instead of an additive seasonal pattern, while function,auto triggers an automatic selection between the two based on their fit.

**Method:** **A = outlier**(A, key, value[, key, value[, [...]]])

Interface to the outlier command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** **A = forecast**(A, key, value[, key, value[, [...]]])

Interface to the forecast command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = check(A, key, value[, key, value[, [...]]])`

Interface to the `check` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = x11(A, key, value[, key, value[, [...]]])`

Interface to the `x11` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = force(A, key, value[, key, value[, [...]]])`

Interface to the `force` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = history(A, key, value[, key, value[, [...]]])`

Interface to the `history` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = metadata(A, key, value[, key, value[, [...]]])`

Interface to the `metadata` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = identify(A, key, value[, key, value[, [...]]])`

Interface to the `identify` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = pickmdl(A, key, value[, key, value[, [...]]])`

Interface to the `pickmdl` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = seats(A, key, value[, key, value[, [...]]])`

Interface to the `seats` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = slidingspans(A, key, value[, key, value[, [...]]])`

Interface to the `slidingspans` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = spectrum(A, key, value[, key, value[, [...]]])`

Interface to the `spectrum` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `A = x11regression(A, key, value[, key, value[, [...]]])`

Interface to the `x11regression` command, see the X-13 ARIMA-SEATS reference manual. All the options must be passed by key/value pairs.

**Method:** `print(A[, basefilename])`

Prints an `.spc` file with all the X-13 commands. The optional second argument is a row char array specifying the name (without extension) of the file.

**Method:** `run(A)`

Calls the X-13 binary and run the previously defined commands. All the results are stored in the structure `A.results`. When it makes sense these results are saved in `dseries` objects (*e.g.* for forecasts or filtered variables).

**Method:** `clean(A)`

Removes the temporary files created by an x13 run that store the intermediate results. This method allows keeping the main folder clean but will also delete potentially important debugging information.

*Example*

```
>> ts = dseries(rand(100,1),'1999M1');
>> o = x13(ts);

>> o.x11('save','(d11)');
>> o.automdl('savelog','amd','mixed','no');
>> o.outlier('types','all','save','(fts)');
>> o.check('maxlag',24,'save','(acf pcf)');
>> o.estimate('save','(mdl est)');
>> o.forecast('maxlead',18,'probability',0.95,'save','(fct fvr)');

>> o.run();
```

The above example shows a run of X13 with various commands an options specified.

*Example*

```
%   1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960
y = [112  115  145  171  196  204  242  284  315  340  360  417 ..
↪.   % Jan
     118  126  150  180  196  188  233  277  301  318  342  391 ..
↪.   % Feb
     132  141  178  193  236  235  267  317  356  362  406  419 ..
↪.   % Mar
     129  135  163  181  235  227  269  313  348  348  396  461 ..
↪.   % Apr
     121  125  172  183  229  234  270  318  355  363  420  472 ..
↪.   % May
     135  149  178  218  243  264  315  374  422  435  472  535 ..
↪.   % Jun
     148  170  199  230  264  302  364  413  465  491  548  622 ..
↪.   % Jul
     148  170  199  242  272  293  347  405  467  505  559  606 ..
↪.   % Aug
     136  158  184  209  237  259  312  355  404  404  463  508 ..
↪.   % Sep
     119  133  162  191  211  229  274  306  347  359  407  461 ..
↪.   % Oct
     104  114  146  172  180  203  237  271  305  310  362  390 ..
↪.   % Nov
     118  140  166  194  201  229  278  306  336  337  405  432 ]
↪'; % Dec

ts = dseries(y,'1949M1');
o = x13(ts);
o.transform('function','auto','savelog','atr');
o.automdl('savelog','all');
o.x11('save','(d11 d10)');
o.run();
o.clean();

y_SA=o.results.d11;
y_seasonal_pattern=o.results.d10;

figure('Name','Comparison raw data and SAed data');
plot(ts.dates,log(o.y.data),ts.dates,log(y_SA.data),ts.dates,
↪log(y_seasonal_pattern.data))
```

The above example shows how to remove a seasonal pattern from a time series. `o.`

`transform('function','auto','savelog','atr')` instructs the subsequent `o.automdl()` command to check whether an additional or a multiplicative pattern fits the data better and to save the result. The result is saved in `o.results.autotransform`, which in the present example indicates that a log transformation, *i.e.* a multiplicative model was preferred. The `o.automdl('savelog','all')` automatically selects a fitting ARIMA model and saves all relevant output to the `.log` file. The `o.x11('save','(d11, d10)')` instructs `x11` to save both the final seasonally adjusted series `d11` and the final seasonal factor `d10` into `dseries` with the respective names in the output structure `o.results`. `o.clean()` removes the temporary files created by `o.run()`. Among these are the `.log` file storing summary information, the `.err` file storing information on problems encountered, the `.out` file storing the raw output, and the `.spc` file storing the specification for the `x11` run. There may be further files depending on the output requested. The last part of the example reads out the results and plots a comparison of the logged raw data and its log-additive decomposition into a seasonal pattern and the seasonally adjusted series.

## 6.4 Miscellaneous

### 6.4.1 Time aggregation

A set of functions allows to convert time series to lower frequencies:

- `dseries2M` converts daily time series object to monthly time series object.
- `dseries2Q` converts daily or monthly time series object to quarterly time series object.
- `dseries2S` converts daily, monthly, or quarterly time series object to bi-annual time series object.
- `dseries2Y` converts daily, monthly, quarterly, or bi-annual time series object to annual time series object.

All these routines have two mandatory input arguments: the first one is a `dseries` object, the second one the name (row char array) of the aggregation method. Possible values for the second argument are:

- `arithmetic-average` (for growth rates),
- `geometric-average` (for growth factors),
- `sum` (for flow variables), and
- `end-of-period` (for stock variables).

*Example*

```
>> ts = dseries(rand(12,1),'2000M1')

ts is a dseries object:

        | Variable_1
2000M1  | 0.55293
2000M2  | 0.14228
2000M3  | 0.38036
2000M4  | 0.39657
2000M5  | 0.57674
2000M6  | 0.019402
2000M7  | 0.57758
2000M8  | 0.9322
2000M9  | 0.10687
2000M10 | 0.73215
2000M11 | 0.97052
2000M12 | 0.60889
```

(continues on next page)

(continued from previous page)

```
>> ds = dseries2Y(ts, 'end-of-period')

ds is a dseries object:

      | Variable_1
2000Y | 0.60889
```

### 6.4.2 Create time series with a univariate model

It is possible to expand a `dseries` object recursively with the `from` command. For instance to create a `dseries` object containing the simulation of an ARMA(1,1) model:

```
>> e = dseries(randn(100, 1), '2000Q1', 'e', '\varepsilon');
>> y = dseries(zeros(100, 1), '2000Q1', 'y');
>> from 2000Q2 to 2024Q4 do y(t)=.9*y(t-1)+e(t)-.4*e(t-1);
>> y

y is a dseries object:

       | y
2000Q1 | 0
2000Q2 | -0.95221
2000Q3 | -0.6294
2000Q4 | -1.8935
2001Q1 | -1.1536
2001Q2 | -1.5905
2001Q3 | 0.97056
2001Q4 | 1.1409
2002Q1 | -1.9255
2002Q2 | -0.29287
       |
2022Q2 | -1.4683
2022Q3 | -1.3758
2022Q4 | -1.2218
2023Q1 | -0.98145
2023Q2 | -0.96542
2023Q3 | -0.23203
2023Q4 | -0.34404
2024Q1 | 1.4606
2024Q2 | 0.901
2024Q3 | 2.4906
2024Q4 | 0.79661
```

The expression following the `do` keyword can be any univariate equation, the only constraint is that the model cannot have leads. It can be a static equation, or a very nonlinear backward equation with an arbitrary number of lags. The `from` command must be followed by a range, which is separated from the (recursive) expression to be evaluated by the `do` command. The assigned variable can also be a variable of a `dseries` object holding several variables (`ds.y(t)=...`); the other variables of the object are left unchanged. The object holding the assigned variable is modified in place (`dseries` objects are handle objects), unless it has to be extended up to the last date of the range, in which case a new object is created. The right hand side can involve the variables of the workspace (parameters) and any function.

CHAPTER

SEVEN

# REPORTING

Dynare provides a simple interface for creating LaTeX reports, comprised of LaTeX tables and PGFPLOTS/TikZ graphs. You can use the report as created through Dynare or pick out the pieces (tables and graphs) you want for inclusion in your own paper. Though Dynare provides a subset of options available through PGFPLOTS/TikZ, you can easily modify the graphs created by Dynare using the options available in the PGFPLOTS/TikZ manual. You can either do this manually or by passing the options to *miscTikzAxisOptions* or *graphMiscTikzAddPlotOptions*.

Reports are created and modified by calling methods on class objects. The objects are hierarchical, with the following order (from highest to lowest): Report, Page, Section, Graph/Table/Vspace, Series. For simplicity of syntax, we abstract away from these classes, allowing you to operate directly on a Report object, while maintaining the names of these classes in the Report class methods you will use.

The report is created sequentially, command by command, hence the order of the commands matters. When an object of a certain hierarchy is inserted, all methods will function on that object until an object of equal or greater hierarchy is added. Hence, once you add a Page to the report, every time you add a Section object, it will be added to this Page until another Page is added to the report (via *addPage*). This will become more clear with the example at the end of the section.

Options to methods are passed differently than those to Dynare commands. They take the form of named options to MATLAB functions where the arguments come in pairs (e.g. `function_name('option_1_name', 'option_1_value', 'option_2_name', 'option_2_value', ...)`, where `option_X_name` is the name of the option while `option_X_value` is the value assigned to that option). The ordering of the option pairs matters only in the unusual case when an option is provided twice (probably erroneously). In this case, the last value passed is the one that is used.

Below, you will see a list of methods available for the Report class and a clarifying example.

**Constructor: report**

Instantiates a Report object.

*Options*

**compiler, FILENAME**

The full path to the LaTeX compiler on your system. If this option is not provided, Dynare will try to find the appropriate program to compile LaTeX on your system. Default is system dependent:

- Windows: the result of findtexmf `--file-type=exe pdflatex`.
- macOS and Linux: the result of `which pdflatex`.

**directory, FILENAME**

The path to the directory you want the report created in. Default: current directory.

**showDate, BOOLEAN**

Display the date and time when the report was compiled. Default: `true`.

**fileName, FILENAME**

The file name to use when saving this report. Default: `report.tex`.

**header, STRING**

The valid LaTeX code to be included in the report before `\begin{document}`. Default: `empty`.

**maketoc, BOOLEAN**

Whether to make the table of contents. One entry is made per page containing a title. Default: `false`.

**margin, DOUBLE**

The margin size. Default: `2.5`.

**marginUnit, 'cm' | 'in'**

Units associated with the margin. Default: `'cm'`.

**orientation, 'landscape' | 'portrait'**

Paper orientation: Default: `'portrait'`.

**paper, 'a4' | 'letter'**

Paper size. Default: `'a4'`.

**reportDirName, FILENAME**

The name of the folder in which to store the component parts of the report (preamble, document, end). Default: `tmpRepDir`.

**showDate, BOOLEAN**

Display the date and time when the report was compiled. Default: true.

**showOutput, BOOLEAN**

Print report creation progress to screen. Shows you the page number as it is created and as it is written. This is useful to see where a potential error occurs in report creation. Default: `true`.

**title, STRING**

Report Title. Default: `none`.

**Method: addPage**

Adds a Page to the Report.

*Options*

**footnote, STRING**

A footnote to be included at the bottom of this page. Default: `none`.

**latex, STRING**

The valid LaTeX code to be used for this page. Allows the user to create a page to be included in the report by passing LaTeX code directly. If this option is passed, the page itself will be saved in the *pageDirName* directory in the form `page_X.tex` where X refers to the page number. Default: `empty`.

**orientation, 'landscape' | 'portrait'**

See *orientation*.

**pageDirName, FILENAME**

The name of the folder in which to store this page. Directory given is relative to the *directory* option of the report class. Only used when the *latex* command is passed. Default: `tmpRepDir`.

**paper, 'a4' | 'letter'**

See *paper*.

**title, STRING | CELL_ARRAY_STRINGS**

With one entry (a STRING), the title of the page. With more than one entry (a CELL_ARRAY_STRINGS), the title and subtitle(s) of the page. Values passed must be valid LaTeX code (e.g., % must be \%). Default: `none`.

**titleFormat, STRING | CELL_ARRAY_STRINGS**

A string representing the valid LaTeX markup to use on `title`. The number of cell array entries must be equal to that of the `title` option if you do not want to use the default value for the title (and subtitles). Default: `\large\bfseries`.

**titleTruncate, INTEGER**

Useful when automatically generating page titles that may become too long, `titleTruncate` can be used to truncate a title (and subsequent subtitles) when they pass the specified number of characters. Default: `.off`.

**Method: addSection**

Adds a `Section` to a `Page`.

*Options*

**cols, INTEGER**

The number of columns in the section. Default: `1`.

**height, STRING**

A string to be used with the `\sectionheight` LaTeX command. Default: `'!'`

**Method: addGraph**

Adds a `Graph` to a `Section`.

*Options*

**data, dseries**

The `dseries` that provides the data for the graph. Default: `none`.

**axisShape, 'box' | 'L'**

The shape the axis should have. `'box'` means that there is an axis line to the left, right, bottom, and top of the graphed line(s). `'L'` means that there is an axis to the left and bottom of the graphed line(s). Default: `'box'`.

**graphDirName, FILENAME**

The name of the folder in which to store this figure. Directory given is relative to the *directory* option of the report class. Default: `tmpRepDir`.

**graphName, STRING**

The name to use when saving this figure. Default: something of the form `graph_pg1_sec2_row1_col3.tex`.

**height, DOUBLE**

The height of the graph, in inches. Default: `4.5`.

**showGrid, BOOLEAN**

Whether to display the major grid on the graph. Default: `true`.

**showLegend, BOOLEAN**

Whether to display the legend.

Unless you use the *graphLegendName* option, the name displayed in the legend is the TeX name associated with the `dseries`. You can modify this TeX name by using *tex_rename*. Default: `false`.

**legendAt, NUMERICAL_VECTOR**

The coordinates for the legend location. If this option is passed, it overrides the *legendLocation* option. Must be of size 2. Default: `empty`.

**showLegendBox, BOOLEAN**

Whether or not to display a box around the legend. Default: `false`.

**legendLocation, OPTION**

Where to place the legend in the graph. Possible values for OPTION are:

```
'south west' | 'south east' | 'north west' | 'north east' | 'outer north east
↪'
```

Default: `'south east'`.

**legendOrientation, 'vertical' | 'horizontal'**

Orientation of the legend. Default: 'horizontal'.

**legendFontSize, OPTION**

The font size for legend entries. Possible values for OPTION are:

```
'tiny' | 'scriptsize' | 'footnotesize' | 'small' | 'normalsize' |
'large' | 'Large' | 'LARGE' | 'huge' | 'Huge'
```

Default: 'tiny'.

**miscTikzAxisOptions, STRING**

If you are comfortable with PGFPLOTS/TikZ, you can use this option to pass arguments directly to the PGFPLOTS/TikZ axis environment command. Specifically to be used for desired PGFPLOTS/TikZ options that have not been incorporated into Dynare Reporting. Default: empty.

**miscTikzPictureOptions, STRING**

If you are comfortable with PGFPLOTS/TikZ, you can use this option to pass arguments directly to the PGFPLOTS/TikZ tikzpicture environment command. (e.g., to scale the graph in the x and y dimensions, you can pass following to this option: 'xscale=2.5, yscale=0.5'). Specifically to be used for desired ``PGFPLOTS/TikZ options that have not been incorporated into Dynare Reporting. Default: empty.

**seriesToUse, CELL_ARRAY_STRINGS**

The names of the series contained in the dseries provided to the *data* option. If empty, use all series provided to data option. Default: empty.

**shade, dates**

The date range showing the portion of the graph that should be shaded. Default: none.

**shadeColor, STRING**

The color to use in the shaded portion of the graph. All valid color strings defined for use by PGFPLOTS/TikZ are valid. A list of defined colors is:

```
'red', 'green', 'blue', 'cyan', 'magenta', 'yellow', 'black', 'gray',
'white','darkgray', 'lightgray', 'brown', 'lime', 'olive', 'orange',
'pink', 'purple', 'teal', 'violet'.
```

Furthermore, You can use combinations of these colors. For example, if you wanted a color that is 20% green and 80% purple, you could pass the string 'green!20!purple'. You can also use RGB colors, following the syntax: 'rgb,255:red,231;green,84;blue,121' which corresponds to the RGB color (231;84;121). More examples are available in the section 4.7.5 of the PGFPLOTS/TikZ manual, revision 1.10. Default: 'green'

**shadeOpacity, DOUBLE**

The opacity of the shaded area, must be in [0,100]. Default: 20.

**tickFontSize, OPTION**

The font size for x- and y-axis tick labels. Possible values for OPTION are:

```
'tiny' | 'scriptsize' | 'footnotesize' | 'small' | 'normalsize' |
'large' | 'Large' | 'LARGE' | 'huge' | 'Huge'
```

Default: 'normalsize'.

**title, STRING | CELL_ARRAY_STRINGS**

Same as *title*, just for graphs.

**titleFontSize, OPTION**

The font size for title. Possible values for OPTION are:

```
'tiny' | 'scriptsize' | 'footnotesize' | 'small' | 'normalsize' |
'large' | 'Large' | 'LARGE' | 'huge' | 'Huge'
```

Default: `'normalsize'`.

**titleFormat, STRING**

The format to use for the graph title. Unlike *titleFormat*, due to a constraint of TikZ, this format applies to the title and subtitles. Default: TikZ default.

**width, DOUBLE**

The width of the graph, in inches. Default: `6.0`.

**writeCSV, BOOLEAN**

Whether to write a CSV file with only the plotted data. The file will be saved in the directory specified by *graphDirName* with the same base name as specified by *graphName* with the ending `.csv`. Default: `false`.

**xlabel, STRING**

The x-axis label. Default: `none`.

**ylabel, STRING**

The y-axis label. Default: `none`.

**xAxisTight, BOOLEAN**

Use a tight x-axis. If false, uses PGFPLOTS/TikZ `enlarge x limits` to choose appropriate axis size. Default: `true`.

**xrange, dates**

The boundary on the x-axis to display in the graph. Default: `all`.

**xTicks, NUMERICAL_VECTOR**

Used only in conjunction with *xTickLabels*, this option denotes the numerical position of the label along the x-axis. The positions begin at `1`. Default: the indices associated with the first and last dates of the `dseries` and, if passed, the index associated with the first date of the *shade* option.

**xTickLabels, CELL_ARRAY_STRINGS | 'ALL'**

The labels to be mapped to the ticks provided by `xTicks`. Default: the first and last dates of the `dseries` and, if passed, the date first date of the *shade* option.

**xTickLabelAnchor, STRING**

Where to anchor the x tick label. Default: `'east'`.

**xTickLabelRotation, DOUBLE**

The amount to rotate the x tick labels by. Default: `0`.

**yAxisTight, BOOLEAN**

Use a tight y-axis. If false, uses PGFPLOTS/TikZ `enlarge y limits` to choose appropriate axis size. Default: `false`.

**yrange, NUMERICAL_VECTOR**

The boundary on the y-axis to display in the graph, represented as a `NUMERICAL_VECTOR` of size `2`, with the first entry less than the second entry. Default: `all`.

**yTickLabelFixed, BOOLEAN**

Round the y tick labels to a fixed number of decimal places, given by `yTickLabelPrecision`. Default: `true`.

**yTickLabelPrecision, INTEGER**

The precision with which to report the `yTickLabel`. Default: `0`.

**yTickLabelScaled, BOOLEAN**

Determines whether there is a common scaling factor for the y-axis. Default: `true`.

**yTickLabelZeroFill, BOOLEAN**

Whether to fill missing precision spots with zeros. Default: `true`.

**showZeroline, BOOLEAN**

Display a solid black line at $y = 0$. Default: `false`.

**zeroLineColor, STRING**

The color to use for the zero line. Only used if *showZeroLine* is true. See the explanation in *shadeColor* for how to use colors with reports. Default: `'black'`.

**Method: addTable**

Adds a `Table` to a `Section`.

*Options*

**data, dseries**

See *data*.

**highlightRows, CELL_ARRAY_STRINGS**

A cell array containing the colors to use for row highlighting. See *shadeColor* for how to use colors with reports. Highlighting for a specific row can be overridden by using the *tableRowColor* option to *addSeries*. Default: `empty`.

**showHlines, BOOLEAN**

Whether to show horizontal lines separating the rows. Default: `false`.

**precision, INTEGER**

The number of decimal places to report in the table data (rounding done via the *round half away from zero* method). Default: `1`.

**range, dates**

The date range of the data to be displayed. Default: `all`.

**seriesToUse, CELL_ARRAY_STRINGS**

See *seriesToUse*.

**tableDirName, FILENAME**

The name of the folder in which to store this table. Directory given is relative to the *directory* option of the report class. Default: `tmpRepDir`.

**tableName, STRING**

The name to use when saving this table. Default: something of the form `table_pg1_sec2_row1_col3.tex`.

**title, STRING**

Same as *title*, just for tables.

**titleFormat, STRING**

Same as *titleFormat*, just for tables. Default: `'\large'`.

**vlineAfter, dates | CELL_ARRAY_DATES**

Show a vertical line after the specified date (or dates if a cell array of dates is passed). Default: `empty`.

**vlineAfterEndOfPeriod, BOOLEAN**

Show a vertical line after the end of every period (i.e. after every year, after the fourth quarter, etc.). Default: `false`.

**showVlines, BOOLEAN**

Whether to show vertical lines separating the columns. Default: `false`.

**writeCSV, BOOLEAN**

Whether to write a CSV file containing the data displayed in the table. The file will be saved in the directory specified by *tableDirName* with the same base name as specified by *tableName* with the ending `.csv`. Default: `false`.

**Method: addSeries**

Adds a `Series` to a `Graph` or a `Table`.

Options specific to graphs begin with `'graph'` while options specific to tables begin with `'table'`.

*Options*

**data, dseries**

See *data*.

**graphBar, BOOLEAN**

Whether to display this series as a bar graph as opposed to the default of displaying it as a line graph. Default: `false`.

**graphFanShadeColor, STRING**

The shading color to use between a series and the previously-added series in a graph. Useful for making fan charts. Default: `empty`.

**graphFanShadeOpacity, INTEGER**

The opacity of the color passed in *graphFanShadeColor*. Default: `50`.

**graphBarColor, STRING**

The outline color of each bar in the bar graph. Only active if *graphBar* is passed. Default: `'black'`.

**graphBarFillColor, STRING**

The fill color of each bar in the bar graph. Only active if *graphBar* is passed. Default: `'black'`.

**graphBarWidth, DOUBLE**

The width of each bar in the bar graph. Only active if *graphBar* is passed. Default: `2`.

**graphHline, DOUBLE**

Use this option to draw a horizontal line at the given value. Default: `empty`.

**graphLegendName, STRING**

The name to display in the legend for this series, passed as valid LaTeX (e.g., `GDP_{US}`, `$\alpha$`, `\color{red}GDP\color{black}`). Will be displayed only if the `data` and *showLegend* options have been passed. Default: the tex name of the series.

**graphLineColor, STRING**

Color to use for the series in a graph. See the explanation in *shadeColor* for how to use colors with reports. Default: `'black'`

**graphLineStyle, OPTION**

Line style for this series in a graph. Possible values for OPTION are:

```
'none' | 'solid' | 'dotted' | 'densely dotted' | 'loosely dotted' | 'dashed'␣
↪|
'densely dashed' | 'loosely dashed' | 'dashdotted' | 'densely dashdotted' |
'loosely dashdotted' | 'dashdotdotted' | 'densely dashdotdotted' |
'loosely dashdotdotted'
```

Default: `'solid'`.

**graphLineWidth DOUBLE**

Line width for this series in a graph. Default: `0.5`.

**graphMarker, OPTION**

The Marker to use on this series in a graph. Possible values for OPTION are:

```
'x' | '+' | '-' | '|' | 'o' | 'asterisk' | 'star' | '10-pointed star' |
'oplus' | 'oplus*' | 'otimes' | 'otimes*' | 'square' | 'square*' |
'triangle' | 'triangle*' | 'diamond' | 'diamond*' | 'halfdiamond*' |
'halfsquare*' | 'halfsquare right*' | 'halfsquare left*' | 'Mercedes star' |
'Mercedes star flipped' | 'halfcircle' | 'halfcircle*' | 'pentagon' |
'pentagon star'
```

Default: none.

**graphMarkerEdgeColor, STRING**

The edge color of the graph marker. See the explanation in *shadeColor* for how to use colors with reports. Default: graphLineColor.

**graphMarkerFaceColor, STRING**

The face color of the graph marker. See the explanation in *shadeColor* for how to use colors with reports. Default: graphLineColor.

**graphMarkerSize, DOUBLE**

The size of the graph marker. Default: 1.

**graphMiscTikzAddPlotOptions, STRING**

If you are comfortable with PGFPLOTS/TikZ, you can use this option to pass arguments directly to the PGFPLOTS/TikZ addPlots command. (e.g., Instead of passing the marker options above, you can pass a string such as the following to this option: 'mark=halfcircle*,mark options={rotate=90, scale=3}'). Specifically to be used for desired PGFPLOTS/TikZ options that have not been incorporated into Dynare Reproting. Default: empty.

**graphShowInLegend, BOOLEAN**

Whether or not to show this series in the legend, given that the *showLegend* option was passed to *addGraph*. Default: true.

**graphVline, dates**

Use this option to draw a vertical line at a given date. Default: empty.

**tableDataRhs, dseries**

A series to be added to the right of the current series. Useful for displaying aggregate data for a series, e.g, if the series is quarterly tableDataRhs could point to the yearly averages of the quarterly series. This would cause quarterly data to be displayed followed by annual data. Default: empty.

**tableRowColor, STRING**

The color that you want the row to be. Predefined values include 'LightCyan' and 'Gray'. Default: 'white'.

**tableRowIndent, INTEGER**

The number of times to indent the name of the series in the table. Used to create subgroups of series. Default: 0.

**tableShowMarkers, BOOLEAN**

In a Table, if true, surround each cell with brackets and color it according to *tableNegColor* and *tablePosColor*. No effect for graphs. Default: false.

**tableAlignRight, BOOLEAN**

Whether to align the series name to the right of the cell. Default: false.

**tableMarkerLimit, DOUBLE**

For values less than $-1 * \texttt{tableMarkerLimit}$, mark the cell with the color denoted by tableNegColor. For those greater than tableMarkerLimit, mark the cell with the color denoted by tablePosColor. Default: 1e-4.

**tableNaNSymb, STRING**

Replace `NaN` values with the text in this option. Default: `NaN`.

**tableNegColor, LATEX_COLOR**

The color to use when marking Table data that is less than zero. Default: `'red'`

**tablePrecision, INTEGER**

The number of decimal places to report in the table data. Default: the value set by *precision*.

**tablePosColor, LATEX_COLOR**

The color to use when marking Table data that is greater than zero. Default: `'blue'`

**tableSubSectionHeader, STRING**

A header for a subsection of the table. No data will be associated with it. It is equivalent to adding an empty series with a name. Default: `''`

**zeroTol, DOUBLE**

The zero tolerance. Anything smaller than `zeroTol` and larger than `-zeroTol` will be set to zero before being graphed or written to the table. Default: `1e-6`.

**Method: addParagraph**

Adds a `Paragraph` to a `Section`.

The `Section` can only be comprised of `Paragraphs` and must only have 1 column.

*Options*

**balancedCols, BOOLEAN**

Determines whether the text is spread out evenly across the columns when the `Paragraph` has more than one column. Default: `true`.

**cols, INTEGER**

The number of columns for the `Paragraph`. Default: `1`.

**heading, STRING**

The heading for the `Paragraph` (like a section heading). The string must be valid LaTeX code. Default: `empty`.

**indent, BOOLEAN**

Whether or not to indent the paragraph. Default: `true`.

**text, STRING**

The paragraph itself. The string must be valid LaTeX code. Default: `empty`.

**Method: addVspace**

Adds a `Vspace` (vertical space) to a `Section`.

*Options*

**hline, INTEGER**

The number of horizontal lines to be inserted. Default: `0`.

**number, INTEGER**

The number of new lines to be inserted. Default: `1`.

**Method: write**

Writes the LaTeX representation of this `Report`, saving it to the file specified by *filename*.

**Method: compile**

Compiles the report written by `write` into a `pdf` file. If the report has not already been written (determined by the existence of the file specified by *filename*, `write` is called.

*Options*

**compiler, FILENAME**

Like *compiler*, except will not overwrite the value of `compiler` contained in the report object. Hence, passing the value here is useful for using different LaTeX compilers or just for passing the value at the last minute.

**showOutput, BOOLEAN**

Print the compiler output to the screen. Useful for debugging your code as the LaTeX compiler hangs if there is a problem. Default: the value of *showOutput*.

**showReport, BOOLEAN**

Open the compiled report (works on Windows and macOS on MATLAB). Default: `true`.

*Example*

The following code creates a one-page report. The first part of the page contains two graphs displayed across two columns and one row. The bottom of the page displays a centered table:

```
%% Create dseries
dsq = dseries('quarterly.csv');
dsa = dseries('annual.csv');
dsca = dseries('annual_control.csv');

%% Report
rep = report();

%% Page 1
rep.addPage('title', {'My Page Title', 'My Page Subtitle'}, ...
            'titleFormat', {'\large\bfseries', '\large'});

% Section 1
rep.addSection('cols', 2);

rep.addGraph('title', 'Graph Column 1', 'showLegend', true, ...
             'xrange', dates('2007q1'):dates('2013q4'), ...
             'shade', dates('2012q2'):dates('2013q4'));
rep.addSeries('data', dsq{'GROWTH_US'}, 'graphLineColor', 'blue', ...
             'graphLineStyle', 'loosely dashed', 'graphLineWidth', 1);
rep.addSeries('data', dsq{'GROWTH_EU'}, 'graphLineColor', 'green', ...
             'graphLineWidth', 1.5);

rep.addGraph('title', 'Graph Column 2', 'showLegend', true, ...
             'xrange', dates('2007q1'):dates('2013q4'), ...
             'shade', dates('2012q2'):dates('2013q4'));
rep.addSeries('data', dsq{'GROWTH_JA'}, 'graphLineColor', 'blue', ...
             'graphLineWidth', 1);
rep.addSeries('data', dsq{'GROWTH_RC6'}, 'graphLineColor', 'green', ...
             'graphLineStyle', 'dashdotdotted', 'graphLineWidth', 1.5);

% Section 2
rep.addVspace('number', 15);
rep.addSection();
rep.addTable('title', 'Table 1', 'range', dates('2012Y'):dates('2014Y'));
shortNames = {'US', 'EU'};
longNames  = {'United States', 'Euro Area'};
for i=1:length(shortNames)
    rep.addSeries('data', dsa{['GROWTH_' shortNames{i}]});
    delta = dsa{['GROWTH_' shortNames{i}]}-dsca{['GROWTH_' shortNames{i}]};
    delta.tex_rename_('$\Delta$');
    rep.addSeries('data', delta, ...
```

(continues on next page)

(continued from previous page)

```
                'tableShowMarkers', true, 'tableAlignRight', true);
end

%% Write & Compile Report
rep.write();
rep.compile();
```

Once compiled, the report looks like:

**My Page Title**

My Page Subtitle




Table 1

| | 2012 | 2013 | 2014 |
|---|---|---|---|
| GROWTH_US | 1.7 | 2.7 | 3.3 |
| Δ | [0.0] | [-0.1] | [0.0] |
| GROWTH_EU | -0.8 | 0.6 | 1.5 |
| Δ | [-0.2] | [-0.2] | [-0.1] |

CHAPTER

EIGHT

# EXAMPLES

Dynare comes with a database of example `.mod` files, which are designed to show a broad range of Dynare features, and are taken from academic papers for most of them. They are distributed in the `examples` subdirectory.

Here is a short list of the examples included. For a more complete description, please refer to the comments inside the files themselves.

## 8.1 Stochastic simulations

`stochastic_simulations/collard_2001_theoretical_moments.mod` `stochastic_simulations/collard_2001_simulated_moments.mod`

Two examples of a small RBC model in a stochastic setup, presented in Collard (2001) (see the file `guide.pdf` which comes with Dynare), both using *`stoch_simul`*. Both use numerical steady state computation. The first one uses theoretical moments, the second one uses simulated moments.

`stochastic_simulations/collard_2001_analytical_steady_state.mod`

A small RBC model in a stochastic setup, presented in Collard (2001) using *`stoch_simul`*. The steady state is solved analytically using the *`steady_state_model`* block and a helper function to call a solver.

`stochastic_simulations/schorfheide_2000_nonstationary.mod`

The cash-in-advance model of Schorfheide (2000) (same as `estimation/schorfheide_2000.mod`) written in non-stationary form. Detrending of the equations is done by Dynare.

`stochastic_simulations/aguiar_gopinath_2007_trend.mod`

Small open economy RBC model with shocks to the growth trend, presented in Aguiar and Gopinath (2007).

`stochastic_simulations/nk_baseline.mod`

Baseline New Keynesian Model estimated in Fernández-Villaverde (2010). It demonstrates how to use an explicit steady-state file to update parameters and call a numerical solver.

## 8.2 Estimation

`estimation/schorfheide_2000.mod`

A cash-in-advance model, estimated by Schorfheide (2000). The file shows how to use the *`estimation`* command.

`estimation/gali_2015.mod`

Basic New Keynesian model of Galí (2015), Chapter 3 showing how to i) use "system prior"-type prior restrictions as in Andrle and Plašil (2018) and ii) run prior/posterior-functions.

`estimation/rbc_irf_matching.mod`

Baseline RBC model with government spending shocks estimated via impulse response function (IRF) matching using *method_of_moments*. Both Frequentist (Maximum Likelihood) and Bayesian (Slice Sampling) approaches are presented. Additionally, it is shown how to estimate an AR(2)-process by working with the roots of the autoregressive process instead of the coefficients.

## 8.3 Perfect foresight

`perfect_foresight/perfect_foresight_rbc.mod`

An elementary real business cycle (RBC) model, simulated in a perfect foresight setup using *perfect_foresight_setup* and *perfect_foresight_solver*.

`perfect_foresight/perfect_foresight_expectation_errors.mod`

Elementary RBC model (same as `perfect_foresight/perfect_foresight_rbc.mod`), simulated in perfect foresight with expectation errors using *perfect_foresight_with_expectation_errors_setup* and *perfect_foresight_with_expectation_errors_solver*: agents behave as under perfect foresight, but they can still be surprised by unexpected shocks, and thus recompute their optimal plans when such an unexpected shock happens.

## 8.4 Optimal policy

`optimal_policy/nk_ramsey_osr.mod`

File demonstrating how to conduct optimal policy experiments in a simple New Keynesian model either under commitment (*ramsey_model*) or using optimal simple rules (*osr*). Based on Christiano *et al.* (2007).

`optimal_policy/nk_ramsey_steady_file.mod`

File demonstrating how to conduct optimal policy experiments in a simple New Keynesian model under commitment (*ramsey_model*) with a user-defined conditional steady state file. Based on Christiano *et al.* (2007).

## 8.5 Heterogeneity

`heterogeneity/krusell_smith_1998.mod`

Krusell and Smith (1998) model with heterogeneous households. Demonstrates loading a pre-computed steady state via *heterogeneity_load_steady_state*, solving the model with *heterogeneity_solve*, and computing impulse response functions.

`heterogeneity/krusell_smith_1998_steady_state.mod`

Krusell and Smith (1998) model with heterogeneous households. Demonstrates computing the steady state numerically using *heterogeneity_compute_steady_state*.

`heterogeneity/hank_one_asset.mod`

One-asset HANK model. Demonstrates loading a pre-computed steady state via *heterogeneity_load_steady_state*, solving with *heterogeneity_solve*, and running stochastic simulations with *heterogeneity_simulate*.

`heterogeneity/hank_one_asset_steady_state.mod`

One-asset HANK model. Demonstrates computing the steady state with parameter calibration using *heterogeneity_compute_steady_state*.

`heterogeneity/hank_two_assets.mod`

Two-asset HANK model with liquid and illiquid assets. Demonstrates loading a pre-computed steady state via *heterogeneity_load_steady_state* and simulating news shocks.

`heterogeneity/hank_two_assets_steady_state.mod`

Two-asset HANK model. Demonstrates computing the steady state with multi-parameter calibration using *heterogeneity_compute_steady_state*.

## 8.6 OccBin

`occbin/rbc_occbin.mod`

RBC model with two occasionally binding constraints. Demonstrates how to set up OccBin using *occbin_setup* and *occbin_solver*. Based on Guerrieri and Iacoviello (2015).

## 8.7 Macroprocessor

`macroprocessor/bkk_1992.mod`

Multi-country RBC model with time to build, presented in Backus *et al.* (1992). The file shows how to use Dynare's macro processor.

## 8.8 Reporting

`reporting/collard_2001_reporting.mod`

Example of Dynare's reporting features using the Collard (2001) RBC model.

## 8.9 Semistructural

`semistructural/pac_model.mod`

Example of a semi-structural model employing PAC (polynomial adjustment cost) specification.

CHAPTER

# NINE

# DYNARE MISC COMMANDS

**MATLAB/Octave command: send_endogenous_variables_to_workspace ;**

Puts the simulation results for the endogenous variables stored in `oo_.endo_simul` into vectors with the same name as the respective variables into the base workspace.

**MATLAB/Octave command: send_exogenous_variables_to_workspace ;**

Puts the simulation results for the exogenous variables stored in `oo_.exo_simul` into vectors with the same name as the respective variables into the base workspace.

**MATLAB/Octave command: send_irfs_to_workspace ;**

Puts the IRFs stored in `oo_.irfs` into vectors with the same name into the base workspace.

**MATLAB/Octave command: clean_current_folder ;**

Cleans the current folder from all files generated by Dynare for the mod-files currently located in the current folder. It will delete all class files (+folders), log-files, and folders generated by previous runs of these files. Note that this also applies to the `results` subfolder. Proceed with caution.

**Command: prior_function**`(OPTIONS);`

Executes a user-defined function on parameter draws from the prior distribution. Dynare returns the results of the computations for all draws in an *ndraws* by $n$ cell array named `oo_.prior_function_results`.

*Options*

**function = FUNCTION_NAME**

The function must have the following header `output_cell = FILENAME(xparam1,M_,options_, oo_,estim_params_,dataset_,dataset_info)`, providing read-only access to all Dynare structures. The only output argument allowed is a $1 \times n$ cell array, which allows for storing any type of output/computations. This option is required.

**sampling_draws = INTEGER**

Number of draws used for sampling. Default: 500.

**Command: posterior_function**`(OPTIONS);`

Same as the *prior_function* command but for the posterior distribution. Results returned in `oo_.posterior_function_results`.

*Options*

**function = FUNCTION_NAME**

See *prior_function_function*.

**sampling_draws = INTEGER**

See *prior_function_sampling_draws*.

**MATLAB/Octave command: generate_trace_plots**(CHAIN_NUMBER);

Generates trace plots of the MCMC draws for all estimated parameters and the posterior density for the specified Markov Chain(s) CHAIN_NUMBER. If CHAIN_NUMBER is a vector of integers, the trace plots will plot contains separate lines for each chain.

**MATLAB/Octave command: internals** FLAG ROUTINENAME[.m]|MODFILENAME

Depending on the value of FLAG, the internals command can be used to run unitary tests specific to a MATLAB/Octave routine (if available), to display documentation about a MATLAB/Octave routine, or to extract some information about the state of Dynare.

*Flags*

--test

Performs the unitary test associated to ROUTINENAME (if this routine exists and if the MATLAB/Octave .m file has unitary test sections).

*Example*

```
>> internals --test ROUTINENAME
```

if routine.m is not in the current directory, the full path has to be given:

```
>> internals --test ../matlab/fr/ROUTINENAME
```

--display-mh-history

Displays information about the previously saved MCMC draws generated by a .mod file named MODFILENAME. This file must be in the current directory.

*Example*

```
>> internals --display-mh-history MODFILENAME
```

--load-mh-history

Loads into the MATLAB/Octave's workspace information about the previously saved MCMC draws generated by a .mod file named MODFILENAME.

*Example*

```
>> internals --load-mh-history MODFILENAME
```

This will create a structure called mcmc_informations (in the workspace) with the following fields:

Nblck

The number of MCMC chains.

InitialParameters

A Nblck*n, where n is the number of estimated parameters, array of doubles. Initial state of the MCMC.

LastParameters

A Nblck*n, where n is the number of estimated parameters, array of doubles. Current state of the MCMC.

InitialLogPost

A Nblck*1 array of doubles. Initial value of the posterior kernel.

LastLogPost

A `Nblck*1` array of doubles. Current value of the posterior kernel.

`InitialSeeds`

A `1*Nblck` structure array. Initial state of the random number generator.

`LastSeeds`

A `1*Nblck` structure array. Current state of the random number generator.

`AcceptanceRatio`

A `1*Nblck` array of doubles. Current acceptance ratios.

**MATLAB/Octave command: prior** `[OPTIONS[ ...]];`

Prints information about the prior distribution given the provided options. If no options are provided, the command returns the list of available options.

*Options*

**table**

Prints a table describing the marginal prior distributions (mean, mode, std., lower and upper bounds, HPD interval).

**moments**

Computes and displays first and second order moments of the endogenous variables at the prior mode (considering the linearized version of the model).

**moments(distribution)**

Computes and displays the prior mean and prior standard deviation of the first and second moments of the endogenous variables (considering the linearized version of the model) by randomly sampling from the prior. The results will also be stored in the `prior` subfolder in a `_endogenous_variables_prior_draws.mat` file.

**irfs(distribution)**

Computes the prior distribution of the impulse response functions of the endogenous variables (considering the linearized version of the model) by randomly sampling from the prior. The results will be stored in the `prior` subfolder in a `_endogenous_variables_irfs_prior_draws.mat` file.

**optimize**

Optimizes the prior density (starting from a random initial guess). The parameters such that the steady state does not exist or does not satisfy the Blanchard and Kahn conditions are penalized, as they would be when maximizing the posterior density. If a significant proportion of the prior mass is defined over such regions, the optimization algorithm may fail to converge to the true solution (the prior mode).

**simulate**

Computes the effective prior mass using a Monte Carlo. Ideally the effective prior mass should be equal to 1, otherwise problems may arise when maximizing the posterior density and model comparison based on marginal densities may be unfair. When comparing models, say $A$ and $B$, the marginal densities, $m_A$ and $m_B$, should be corrected for the estimated effective prior mass $p_A \neq p_B \leq 1$ so that the prior mass of the compared models are identical.

**plot**

Plots the marginal prior density.

**MATLAB/Octave command: search** `VARIABLENAME[ OPTION]`

Searches all occurrences of a variable in a model, and prints the equations where the variable appear in the command line window. If OPTION is set to *withparamvalues*, the values of the parameters (if available) are displayed instead of the name of the parameters. Requires the *json* command line option to be set.

*Example*

Assuming that we already ran a *.mod* file and that the workspace has not been cleaned after, we can search for all the equations containing variable *X*

```
>> search X

Y = alpha*X/(1-X)+e;

diff(X) = beta*(X(-1)-mX) + gamma1*Z + gamma2*R + u;
```

To replace the parameters with estimated or calibrated parameters:

```
>> search X withparamvalues

Y = 1.254634*X/(1-X)+e;

diff(X) = -0.031459*(X(-1)-mX) + 0.1*Z - 0.2*R + u;
```

**MATLAB/Octave command: dplot** [OPTION VALUE[ ...]]

Plot expressions extracting data from different `dseries` objects.

*Options*

**--expression EXPRESSION**

`EXPRESSION` is a mathematical expression involving variables available in the `dseries` objects, `dseries` methods, numbers or parameters. All the referenced objects are supposed to be available in the calling workspace.

**--dseries NAME**

`NAME` is the name of a `dseries` object from which the variables involved in `EXPRESSION` will be extracted.

**--range DATE1:DATE2**

This option is not mandatory and allows plotting the expressions only over a sub-range. `DATE1` and `DATE2` must be dates as defined in *Dates in a mod file*.

**--style MATLAB_SCRIPT_NAME**

Name of a Matlab script (without extension) containing Matlab commands to customize the produced figure.

**--title MATLAB_STRING**

Adds a title to the figure.

**--with-legend**

Prints a legend below the produced plot.

*Remarks*

- More than one –expression argument is allowed, and they must come first.
- For each `dseries` object we plot all the expressions. We use two nested loops, the outer loop is over the `dseries` objects and the inner loop over the expressions. This determines the ordering of the plotted lines.
- All `dseries` objects must be defined in the calling workspace, if a `dseries` object is missing the routine throws a warning (we only build the plots for the available `dseries` objects), if all `dseries` objects are missing the routine throws an error.
- If the range is not provided, the expressions cannot involve leads or lags.

*Example*

```
>> toto = dseries(randn(100,3), dates('2000Q1'), {'x','y','z'});
>> noddy = dseries(randn(100,3), dates('2000Q1'), {'x','y','z'});
>> b = 3;
>> dplot --expression 2/b*cumsum(x/y(-1)-1) --dseries toto --dseries␣
↪noddy --range 2001Q1:2024Q1 --title 'This is my plot'
```

will produce plots for `2/b*cumsum(x/y(-1)-1)`, where `x` and `y` are variables in `dseries` objects `toto` and `noddy`, in the same figure.

**MATLAB/Octave command: `set_dynare_threads(NAME_OF_MEX_FILE,INTEGER);`**

A NAME_OF_MEX_FILE and INTEGER pair that can be used to set the number of parallel threads employed during the execution of MEX files. To get the number `n` of logical cores available, you can run `n=num_procs`.

Available NAME_OF_MEX_FILE options are:

`'sparse_hessian_times_B_kronecker_C'`

MEX file used during computation of second-order solutions and in identification. Default number of threads: a quarter of the number of logical cores, with a ceiling of 4.

`'local_state_space_iteration_2'`

MEX file used during nonlinear filtering at second order without `k_order_solver` option or with pruning. Default number of threads: a quarter of the number of logical cores, with a ceiling of 4.

`'local_state_space_iteration_3'`

MEX file used during nonlinear filtering at third order without `k_order_solver` option or with pruning. Default number of threads: a quarter of the number of logical cores, with a ceiling of 4.

`'local_state_space_iteration_k'`

MEX file used during nonlinear filtering at order higher than 3 (without pruning) or at third order with `k_order_solver` option and without pruning. Default number of threads: 1.

`'perfect_foresight_problem'`

MEX file used during perfect foresight simulations. Default number of threads: a quarter of the number of logical cores, with a ceiling of 4.

`'perfect_foresight_block_problem'`

MEX file used during perfect foresight simulations with block decomposition. Default number of threads: a quarter of the number of logical cores, with a ceiling of 4.

`'stochastic_perfect_foresight_problem'`

MEX file used during stochastic extended path simulations (`extended_path` with `order>0`). Default number of threads: a quarter of the number of logical cores, with a ceiling of 4.

`'k_order_perturbation'`

MEX file used for perturbation solutions with `k_order_solver` option. Default number of threads: a quarter of the number of logical cores, with a ceiling of 4.

`'pardiso'`

MEX file used during perfect foresight simulations with `stack_solve_algo=9` or `stack_solve_algo=10` option. Default number of threads: number of logical cores.

# BIBLIOGRAPHY


[1] Milton Abramowitz and Irene A. Stegun, (1965). *Handbook of mathematical functions*. Dover Publications, New York.

[2] Stéphane Adjemian, Matthieu Darracq Parriès, and Stéphane Moyen, (2008). Towards a monetary policy evaluation framework. Working Paper Series 942, European Central Bank.

[3] Stéphane Adjemian and Michel Juillard, (2025). Stochastic extended path. *Journal of Economic Dynamics and Control*, pages 105227. doi:10.1016/j.jedc.2025.105227.

[4] Stéphane Adjemian and Michel Juillard, (2025). Stochastic extended path. Dynare Working Papers 84, CEPREMAP. URL: https://ideas.repec.org/p/cpm/dynare/084.html, doi:None.

[5] Mark Aguiar and Gita Gopinath, (2007). Emerging market business cycles: the cycle is the trend. *Journal of Political Economy*, 115:69–102. doi:10.1086/511283.

[6] Gianni Amisano and Oreste Tristani, (2010). Euro area inflation persistence in an estimated nonlinear DSGE model. *Journal of Economic Dynamics and Control*, 34(10):1837–1858. doi:10.1016/j.jedc.2010.05.001.

[7] Martin M. Andreasen, Jesús Fernández-Villaverde, and Juan F. Rubio-Ramírez, (2018). The pruned state-space system for non-linear DSGE models: theory and empirical applications. *Review of Economic Studies*, 85(1):1–49. doi:10.1093/restud/rdx037.

[8] Donald W. K. Andrews, (1991). Heteroskedasticity and autocorrelation consistent covariance matrix estimation. *Econometrica*, 59(3):817–858. doi:10.2307/2938229.

[9] Michal Andrle and Miroslav Plašil, (2018). Econometrics with system priors. *Economics Letters*, 172:134–137. doi:10.1016/j.econlet.2018.08.038.

[10] Ienkaran Arasaratnam and Simon Haykin, (2009). Cubature Kalman Filters. *IEEE Transactions on Automatic Control*, 54(6):1254–1269. doi:10.1109/tac.2009.2019800.

[11] Adrien Auclert, Bence Bardóczy, Matthew Rognlie, and Ludwig Straub, (2021). Using the Sequence-Space Jacobian to solve and estimate heterogeneous-agent models. *Econometrica*, 89(5):2375–2408. doi:10.3982/ecta17434.

[12] David K. Backus, Patrick J. Kehoe, and Finn E. Kydland, (1992). International real business cycles. *Journal of Political Economy*, 100(4):745–75. doi:10.1086/261838.

[13] Marianne Baxter and Robert G. King, (1999). Measuring business cycles: approximate band-pass filters for economic time series. *Review of Economics and Statistics*, 81(4):575–593. doi:10.1162/003465399558454.

[14] Anmol Bhandari, Thomas Bourany, David Evans, and Mikhail Golosov, (2023). A perturbational approach for approximating heterogeneous agent models. NBER Working Papers 31744, National Bureau of Economic Research, Inc.

[15] Dario A. Bini, Guy Latouche, and Beatrice Meini, (2002). Solving matrix polynomial equations arising in queueing problems. *Linear Algebra and its Applications*, 340(1–3):225–244. doi:10.1016/s0024-3795(01)00426-8.

[16] Gregor Boehl, (2022). Dime mcmc: a swiss army knife for bayesian inference. *SSRN Electronic Journal*. doi:10.2139/ssrn.4250395.

[17] Benjamin Born and Johannes Pfeifer, (2014). Policy risk and the business cycle. *Journal of Monetary Economics*, 68:68–85. doi:10.1016/j.jmoneco.2014.07.012.

[18] Raouf Boucekkine, (1995). An alternative methodology for solving nonlinear forward-looking models. *Journal of Economic Dynamics and Control*, 19(4):711–734. doi:10.1016/0165-1889(94)00800-w.

[19] Flint Brayton, Morris Davis, and Peter Tulip Tulip, (2000). Polynomial adjustment costs in FRB/US. Technical Report, Board of Governors of the Federal Reserve System (U.S.).

[20] Flint Brayton and Peter Tinsley, (1996). A guide to FRB/US: a macroeconomic model of the United States. Finance and Economics Discussion Series 96-42, Board of Governors of the Federal Reserve System (U.S.). URL: https://EconPapers.repec.org/RePEc:fip:fedgfe:96-42.

[21] Stephen P. Brooks and Andrew Gelman, (1998). General methods for monitoring convergence of iterative simulations. *Journal of computational and graphical statistics*, 7(4):434–455. doi:10.1080/10618600.1998.10474787.

[22] Margarida F. Cardoso, Romualdo L. Salcedo, and S. Feyo de Azevedo, (1996). The simplex-simulated annealing approach to continuous non-linear optimization. *Computers & Chemical Engineering*, 20(9):1065–1080. doi:10.1016/0098-1354(95)00221-9.

[23] Béatrice Cherrier, Aurélien Saïdi, and Francesco Sergi, (2023). "write your model almost as you would on paper and Dynare will take care of the rest!" a history of the dynare software. *OEconomia*, 13–3:801–848. doi:10.4000/oeconomia.16123.

[24] Siddhartha Chib and Srikanth Ramamurthy, (2010). Tailored randomized block MCMC methods with application to DSGE models. *Journal of Econometrics*, 155(1):19–38. doi:10.1016/j.jeconom.2009.08.003.

[25] Lawrence J. Christiano, Martin Eichenbaum, and Charles L. Evans, (2005). Nominal rigidities and the dynamic effects of a shock to monetary policy. *Journal of Political Economy*, 113(1):1–45. doi:10.1086/426038.

[26] Lawrence J. Christiano, Roberto Motto, and Massimo Rostagno, (2007). Notes on Ramsey-optimal monetary policy. Available at http://faculty.wcas.northwestern.edu/Âălchrist/d16/d1606/ramsey.pdf.

[27] Lawrence J. Christiano, Mathias Trabandt, and Karl Walentin, (2010). DSGE models for monetary policy analysis. In Benjamin M. Friedman and Michael Woodford, editors, *Handbook of Monetary Economics*, volume 3, chapter 7, pages 285–367. Elsevier. doi:10.1016/B978-0-444-53238-1.00007-7.

[28] Lawrence J. Christiano, Mathias Trabandt, and Karl Walentin, (2011). Introducing financial frictions and unemployment into a small open economy model. *Journal of Economic Dynamics and Control*, 35(12):1999–2041. doi:10.1016/j.jedc.2011.09.005.

[29] Kai Christoffel, Günter Coenen, and Anders Warne, (2011). Forecasting with DSGE models. In Michael P. Clements and David F. Hendry, editors, *The Oxford Handbook of Economic Forecasting*, pages 89–128. Oxford University Press. doi:10.1093/oxfordhb/9780195398649.013.0005.

[30] Fabrice Collard, (2001). Stochastic simulations with DYNARE. a practical guide. Technical Report, GREMAQ, University of Toulouse.

[31] Fabrice Collard and Michel Juillard, (2001). A higher-order Taylor expansion approach to simulation of stochastic forward-looking models with an application to a nonlinear Phillips curve model. *Computational Economics*, 17(2-3):125–139. doi:10.1023/A:1011624124377.

[32] Fabrice Collard and Michel Juillard, (2001). Accuracy of stochastic perturbation methods: the case of asset pricing models. *Journal of Economic Dynamics and Control*, 25(6-7):979–999. doi:10.1016/s0165-1889(00)00064-6.

[33] Angelo Corana, Maurizio Marchesi, Claudio Martini, and Sandro Ridella, (1987). Minimizing multimodal functions of continuous variables with the \textquotedblleft simulated annealing\textquotedblright algorithm. *ACM Transactions on Mathematical Software*, 13(3):262–280. doi:10.1145/29380.29864.

[34] Pablo Cuba-Borda, Luca Guerrieri, Matteo Iacoviello, and Molin Zhong, (2019). Likelihood evaluation of models with occasionally binding constraints. *Journal of Applied Econometrics*, 34(7):1073–1085. doi:10.1002/jae.2729.

[35] Richard Dennis, (2007). Optimal policy in rational expectations models: new solution algorithms. *Macroeconomic Dynamics*, 11(1):31–55. doi:10.1017/s1365100507050341.

[36] Darrell Duffie and Kenneth J. Singleton, (1993). Simulated moments estimation of Markov models of asset prices. *Econometrica*, 61(4):929. doi:10.2307/2951768.

[37] James Durbin and Siem J. Koopman, (2012). *Time series analysis by state space methods*. Oxford University Press, Oxford, second revised edition edition.

[38] Ray C. Fair and John B. Taylor, (1983). Solution and maximum likelihood estimation of dynamic nonlinear rational expectations models. *Econometrica*, 51(4):1169. doi:10.2307/1912057.

[39] Jesús Fernández-Villaverde, (2010). The econometrics of DSGE models. *SERIEs*, 1(1-2):3–49. doi:10.1007/s13209-009-0014-7.

[40] Jesús Fernández-Villaverde and Juan F. Rubio-Ramírez, (2005). Estimating dynamic equilibrium economies: linear versus nonlinear likelihood. *Journal of Applied Econometrics*, 20(7):891–910. doi:10.1002/jae.814.

[41] Jesús Fernández-Villaverde and Juan Francisco Rubio-Ramírez, (2004). Comparing dynamic equilibrium models to data: a Bayesian approach. *Journal of Econometrics*, 123(1):153–187. doi:10.1016/j.jeconom.2003.10.031.

[42] Michael C. Ferris and Todd S. Munson, (1999). Interfaces to PATH 3.0: design, implementation and usage. *Computational Optimization and Applications*, 12(1–3):207–227. doi:10.1023/a:1008636318275.

[43] Jordi Galí, (2015). *Monetary policy, inflation and the business cycle*. Princeton University Press, Princeton, 2 edition.

[44] John Geweke, (1992). Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments. In José M. Bernardo, James O. Berger, A. Philip Dawid, and Adrian F. M. Smith, editors, *Bayesian Statistics*, volume 4, pages 169–194. Clarendon Press, Oxford. doi:10.1093/oso/9780198522669.003.0010.

[45] John Geweke, (1999). Using simulation methods for Bayesian econometric models: inference, development, and communication. *Econometric Reviews*, 18(1):1–73. doi:10.1080/07474939908800428.

[46] Paolo Giordani, Michael Pitt Pitt, and Robert Kohn, (2011). Bayesian inference for time series state space models. In John Geweke, Gary Koop, and Herman van Dijk, editors, *The Oxford Handbook of Bayesian Econometrics*, chapter 3, pages 60–124. Oxford University Press, Oxford. doi:10.1093/oxfordhb/9780199559084.013.0004.

[47] Massimo Giovannini, Philipp Pfeiffer, and Marco Ratto, (2021). Efficient and robust inference of models with occasionally binding constraints. JRC Working Papers in Economics and Finance 2021-03, Joint Research Centre, European Commission. URL: https://ideas.repec.org/p/jrs/wpaper/202103.html.

[48] William L. Goffe, Gary D. Ferrier, and John Rogers, (1994). Global optimization of statistical functions with simulated annealing. *Journal of Econometrics*, 60(1/2):65–100. doi:10.1016/0304-4076(94)90038-8.

[49] N. J. Gordon, D. J. Salmond, and A. F. M. Smith, (1993). Novel approach to nonlinear/non-Gaussian Bayesian state estimation. *IEE Proceedings F - Radar and Signal Processing*, 140(2):107–113. doi:10.1049/ip-f-2.1993.0015.

[50] Luca Guerrieri and Matteo Iacoviello, (2015). OccBin: a toolkit for solving dynamic models with occasionally binding constraints easily. *Journal of Monetary Economics*, 70(C):22–38. doi:10.1016/j.jmoneco.2014.08.005.

[51] Gaygysyz Guljanov, Willi Mutschler, and Mark Trede, (forthcoming). Pruned skewed kalman filter and smoother with application to dsge models. *Journal of Economic Dynamics and Control*.

[52] Lars Peter Hansen, (1982). Large sample properties of generalized method of moments estimators. *Econometrica*, 50(4):1029–1054. doi:10.2307/1912775.

[53] Nikolaus Hansen and Stefan Kern, (2004). *Evaluating the CMA Evolution Strategy on multimodal test functions*, pages 282–291. Springer, Berlin, 2004. doi:10.1007/978-3-540-30217-9_29.

[54] Andrew C. Harvey and Garry D. A. PHILLIPS, (1979). Maximum likelihood estimation of regression models with autoregressive-moving average disturbances. *Biometrika*, 66(1):49–58. doi:10.1093/biomet/66.1.49.

[55] Edward Herbst, (2015). Using the 'Chandrasekhar Recursions' for likelihood evaluation of DSGE models. *Computational Economics*, 45(4):693–705. doi:10.1007/s10614-014-9430-2.

[56] Edward Herbst and Frank Schorfheide, (2014). Sequential Monte Carlo sampling for DSGE models. *Journal of Applied Econometrics*, 29(7):1073–1098. doi:10.1002/jae.2397.

[57] Robert J. Hodrick and Edward C. Prescott, (1997). Post-war U.S. business cycles: an empirical investigation. *Journal of Money, Credit and Banking*, 29(1):1–16. doi:10.2307/2953682.

[58] Edward L. Ionides, (2003). Inference and filtering for partially observed diffusion processes via Sequential Monte Carlo. Statistics Department Technical Report 405, University of Michigan.

[59] Peter N. Ireland, (2004). A method for taking models to the data. *Journal of Economic Dynamics and Control*, 28(6):1205–1226. doi:10.1016/s0165-1889(03)00080-0.

[60] Nikolay Iskrev, (2010). Local identification in DSGE models. *Journal of Monetary Economics*, 57:189–202. doi:10.1016/j.jmoneco.2009.12.007.

[61] Kenneth Judd, (1996). Approximation, perturbation, and projection methods in economic analysis. In Hans Amman, David Kendrick, and John Rust, editors, *Handbook of Computational Economics*, volume 1, chapter 12, pages 511–585. North Holland Press. doi:10.1016/S1574-0021(96)01014-3.

[62] Michel Juillard, (1996). Dynare: a program for the resolution and simulation of dynamic models with forward variables through the use of a relaxation algorithm. CEPREMAP Working Papers (Couverture Orange) 9602, CEPREMAP. URL: https://ideas.repec.org/p/cpm/cepmap/9602.html.

[63] Simon J. Julier and Jeffrey K. Uhlmann, (1997). A new extension of the Kalman filter to nonlinear systems. In *The 11th International Symposium on Aerospace/Defence Sensing, Simulation and Controls*, 182–193. Orlando, 1997. doi:10.1117/12.280797.

[64] Christian Kanzow and Stefania Petra, (2004). On a semismooth least squares formulation of complementarity problems with gap reduction. *Optimization Methods and Software*, 19(5):507–525. doi:10.1080/10556780410001683096.

[65] Jinill Kim, Sunghyun Kim, Ernst Schaumburg, and Christopher A. Sims, (2008). Calculating and using second-order accurate solutions of discrete time dynamic equilibrium models. *Journal of Economic Dynamics and Control*, 32:3397–3414. 11. doi:10.1016/j.jedc.2008.02.003.

[66] Ivana Komunjer and Serena Ng, (2011). Dynamic identification of dynamic stochastic general equilibrium models. *Econometrica*, 79(6):1995–2032. doi:10.3982/ecta8916.

[67] Gary Koop, (2003). *Bayesian econometrics*. John Wiley & Sons, Chichester.

[68] Siem J. Koopman and James Durbin, (2000). Fast filtering and smoothing for multivariate state space models. *Journal of Time Series Analysis*, 21(3):281–296. doi:10.1111/1467-9892.00186.

[69] Siem J. Koopman and James Durbin, (2003). Filtering and smoothing of state vector for diffuse state-space models. *Journal of Time Series Analysis*, 24(1):85–98. doi:10.1111/1467-9892.00294.

[70] Jayesh H. Kotecha and Petar M. Djuric, (2003). Gaussian particle filtering. *IEEE Transactions on Signal Processing*, 51(10):2592–2601. doi:10.1109/tsp.2003.816758.

[71] Jayesh H. Kotecha and Petar M. Djuric, (2003). Gaussian sum particle filtering. *IEEE Transactions on Signal Processing*, 51(10):2602–2612. doi:10.1109/tsp.2003.816754.

[72] Per Krusell and Anthony A. Smith, Jr., (1998). Income and wealth heterogeneity in the macroeconomy. *Journal of Political Economy*, 106(5):867–896. doi:10.1086/250034.

[73] Alexei V. Kuntsevich and Franz Kappel, (1997). SolvOpt - the solver for local nonlinear optimization problems (version 1.1, Matlab, C, FORTRAN). Technical Report, University of Graz.

[74] Laffargue, (1990). Résolution d'un modèle macroéconomique avec anticipations rationnelles. *Annales d'Économie et de Statistique*, pages 97. doi:10.2307/20075774.

[75] Jane Liu and Mike West, (2001). *Combined parameter and state estimation in simulation-based filtering*, chapter 10, pages 197–223. Springer, New York, 2001. doi:10.1007/978-1-4757-3437-9_10.

[76] Lawrence M. Murray, Emlyn M. Jones, and John Parslow, (2013). On disturbance state-space models and the particle marginal Metropolis-Hastings sampler. *SIAM/ASA Journal on Uncertainty Quantification*, 1(1):494–521. doi:10.1137/130915376.

[77] Willi Mutschler, (2015). Identification of DSGE models—the effect of higher-order approximation and pruning. *Journal of Economic Dynamics and Control*, 56:34–54. doi:10.1016/j.jedc.2015.04.007.

[78] Willi Mutschler, (2018). Higher-order statistics for DSGE models. *Econometrics and Statistics*, 6:44–56. doi:10.1016/j.ecosta.2016.10.005.

[79] Marco Del Negro and Frank Schorfheide, (2004). Priors from general equilibrium models for VARS. *International Economic Review*, 45(2):643–673. doi:10.1111/j.1468-2354.2004.00139.x.

[80] Joseph G. Pearlman, Paul D. Currie, and Peter W. Levine, (1986). Rational expectations models with partial information. *Economic Modelling*, 3:90–105. doi:10.1016/0264-9993(86)90001-5.

[81] Johannes Pfeifer, (2013). A guide to specifying observation equations for the estimation of DSGE models. Technical Report, Universität der Bundeswehr München. URL: https://drive.google.com/file/d/1r89OU5OE3CBa6tOlj6l3hNVWEaRH5Anv/view?usp=sharing.

[82] Johannes Pfeifer, (2014). An introduction to graphs in Dynare. Technical Report, Universität der Bundeswehr München.

[83] Michael K. Pitt and Neil Shephard, (1999). Filtering via simulation: auxiliary particle filters. *Journal of the American Statistical Association*, 94(446):590–599. doi:10.1080/01621459.1999.10474153.

[84] Christophe Planas, Marco Ratto, and Alessandro Rossi, (2015). Slice sampling in Bayesian estimation of DSGE models. Technical Report, European Commission, Joint Research Centre.

[85] Zhongjun Qu and Denis Tkachenko, (2012). Identification and frequency domain quasi-maximum likelihood estimation of linearized dynamic stochastic general equilibrium models: QML estimation of linearized DSGE models. *Quantitative Economics*, 3(1):95–132. doi:10.3982/qe126.

[86] Pau Rabanal and Juan F. Rubio-Ramírez, (2005). Comparing New Keynesian models of the business cycle: a Bayesian approach. *Journal of Monetary Economics*, 52(6):1151–1166. doi:10.1016/j.jmoneco.2005.08.008.

[87] Adrian E. Raftery and Steven Lewis, (1992). How many iterations in the Gibbs sampler? In José M. Bernardo, James O. Berger, A. Philip Dawid, and Adrian F. M. Smith, editors, *Bayesian Statistics*, volume 4, pages 763–773. Clarendon Press, Oxford.

[88] Marco Ratto, (2008). Analysing DSGE models with global sensitivity analysis. *Computational Economics*, 31(2):115–139.

[89] Marco Ratto and Nikolay Iskrev, (2011). Algorithms for identification analysis under the Dynare environment: final version of the software. Technical Report, Joint Research Centre, European Commission.

[90] Normann Rion, (2026). Rich heterogeneity in Dynare 7: a practical description. Dynare Working Papers 88, CEPREMAP. URL: https://www.dynare.org/wp-repo/dynarewp088.pdf.

[91] K. Geert Rouwenhorst, (1995). Asset pricing implications of equilibrium business cycle models. In Thomas F. Cooley, editor, *Frontiers of Business Cycle Research*, pages 294–330. Princeton University Press, Princeton, NJ.

[92] Francisco J. Ruge-Murcia, (2012). Estimating nonlinear DSGE models by the simulated method of moments: with an application to business cycles. *Journal of Economic Dynamics and Control*, 36(6):914–938. doi:10.1016/j.jedc.2012.01.008.

[93] Stephanie Schmitt-Grohé and Martin Uríbe, (2004). Solving dynamic general equilibrium models using a second-order approximation to the policy function. *Journal of Economic Dynamics and Control*, 28(4):755–775. doi:10.1016/s0165-1889(03)00043-5.

[94] Robert B. Schnabel and Elizabeth Eskow, (1990). A new modified Cholesky factorization. *SIAM Journal on Scientific and Statistical Computing*, 11(6):1136–1158. doi:10.1137/0911064.

[95] Frank Schorfheide, (2000). Loss function-based evaluation of DSGE models. *Journal of Applied Econometrics*, 15(6):645–670. doi:10.1002/jae.582.

[96] Christopher A. Sims, Daniel F. Waggoner, and Tao Zha, (2008). Methods for inference in large multiple-equation Markov-switching models. *Journal of Econometrics*, 146(2):255–274. doi:10.1016/j.jeconom.2008.08.023.

[97] Martin Sköld and Gareth O. Roberts, (2003). Density estimation for the Metropolis–Hastings algorithm. *Scandinavian Journal of Statistics*, 30(4):699–718. doi:10.1111/1467-9469.00359.

[98] Frank Smets and Rafael Wouters, (2003). An estimated dynamic stochastic general equilibrium model of the euro area. *Journal of the European Economic Association*, 1(5):1123–1175. doi:10.1162/154247603770383415.

[99] James H. Stock and Mark W. Watson, (1999). Forecasting inflation. *Journal of Monetary Economics*, 44(2):293–335. doi:10.1016/S0304-3932(99)00027-6.

[100] Cajo J. F. Ter Braak, (2006). A Markov Chain Monte Carlo version of the genetic algorithm Differential Evolution: easy Bayesian computing for real parameter spaces. *Statistics and Computing*, 16(3):239–249. doi:10.1007/s11222-006-8769-1.

[101] Harald Uhlig, (2001). A toolkit for analysing nonlinear dynamic stochastic models easily. In Ramon Marimon and Andrew Scott, editors, *Computational Methods for the Study of Dynamic Economies*, pages 30–61. Oxford University Press, Oxford. doi:10.1093/0199248273.003.0003.

[102] Sébastien Villemot, (2011). Solving rational expectations models at first order: what Dynare does. Dynare Working Papers 2, CEPREMAP. URL: https://ideas.repec.org/p/cpm/dynare/002.html.

[103] Daniel F. Waggoner, Hongwei Wu, and Tao Zha, (2016). Striated Metropolis–Hastings sampler for high-dimensional models. *Journal of Econometrics*, 192(2):406–420. doi:10.1016/j.jeconom.2016.02.007.

[104] Eric A. Wan and Rudolph van der Merwe, (2001). The unscented Kalman filter. In Haykin. Simon, editor, *Kalman Filtering and Neural Networks*, chapter 7, pages 221–280. John Wiley & Sons, New York. doi:10.1002/0471221546.ch7.

[105] U.S. Census Bureau, (2020). X-13 ARIMA-SEATS Reference Manual, Version 1.1. Technical Report, Center for Statistical Research and Methodology, U.S. Census Bureau. URL: https://www.census.gov/data/software/x13as.html.

# INDEX


## A

abs (*dseries method*), 278
abs (*function*), 29
abs(B) (*dseries method*), 278
abs(x) (*function*), 29
abs_ (*dseries method*), 278
abs_(B) (*dseries method*), 278
acos (*function*), 29
acos(x) (*function*), 29
acosh (*function*), 30
acosh(x) (*function*), 30
addGraph (*reporting method*), 319
addPage (*reporting method*), 318
addParagraph (*reporting method*), 325
addSection (*reporting method*), 319
addSeries (*reporting method*), 323
addTable (*reporting method*), 322
addVspace (*reporting method*), 325
align (*dseries method*), 279
align(A, B) (*dseries method*), 279
align_ (*dseries method*), 279
align_(A, B) (*dseries method*), 279
append (*dates method*), 267
append(A, B) (*dates method*), 267
append_ (*dates method*), 267
append_(A, B) (*dates method*), 267
arima (*x13 method*), 311
arima(A, key, value[, key, value[, [...]]]) (*x13 method*), 311
asin (*function*), 29
asin(x) (*function*), 29
asinh (*function*), 30
asinh(x) (*function*), 30
atan (*function*), 29
atan(x) (*function*), 29
atanh (*function*), 30
atanh(x) (*function*), 30
automdl (*x13 method*), 311
automdl(A, key, value[, key, value[, [...]]]) (*x13 method*), 311

## B

backcast (*dseries method*), 281
backcast(A, B[, diff]) (*dseries method*), 281
backcast_ (*dseries method*), 281
backcast_(A, B[, diff]) (*dseries method*), 281
baxter_king_filter (*dseries method*), 281
baxter_king_filter(A[, hf[, lf[, K]]]) (*dseries method*), 281
baxter_king_filter_ (*dseries method*), 281
baxter_king_filter_(A[, hf[, lf[, K]]]) (*dseries method*), 281
bvar_density (*command*), 152
bvar_forecast (*command*), 186
bvar_irf (*command*), 152

## C

calib_smoother (*command*), 176
calib_smoother(OPTIONS...) (*command*), 176
cbrt (*function*), 29
cbrt(x) (*function*), 29
center (*dseries method*), 282
center(A[, geometric]) (*dseries method*), 282
center_ (*dseries method*), 282
center_(A[, geometric]) (*dseries method*), 282
chain (*dseries method*), 282
chain(A, B) (*dseries method*), 282
chain_ (*dseries method*), 282
chain_(A, B) (*dseries method*), 282
change_type (*command*), 24
change_type(var|varexo|varexo_det|parameters) (*command*), 24
char (*dates method*), 267
char(A) (*dates method*), 267
check (*command*), 64
check (*dseries method*), 283
check (*x13 method*), 311
check(A) (*dseries method*), 283
check(A, key, value[, key, value[, [...]]]) (*x13 method*), 311
check(OPTIONS...) (*command*), 64
clean (*x13 method*), 312
clean(A) (*x13 method*), 312
clean_current_folder (*MATLAB command*), 333
cluster (*config block*), 257
collect_latex_files (*MATLAB command*), 253
colon (*dates method*), 268
colon(A, B) (*dates method*), 268
colon(A, i, B) (*dates method*), 268
commonop (*dseries method*), 283
commonop(A, op, B) (*dseries method*), 283
compilation_setup (*command*), 252

compilation_setup(OPTIONS) (*command*), 252
compile (*reporting method*), 325
conditional_forecast (*command*), 180
conditional_forecast(OPTIONS...) (*command*), 180
conditional_forecast_paths (*block*), 184
conditional_forecast_paths(overwrite) (*block*), 184
copy (*dates method*), 268
copy(A) (*dates method*), 268
cos (*function*), 29
cos(x) (*function*), 29
cosh (*function*), 29
cosh(x) (*function*), 29
cumprod (*dseries method*), 284
cumprod(A[, d[, v]]) (*dseries method*), 284
cumprod_ (*dseries method*), 284
cumprod_(A[, d[, v]]) (*dseries method*), 284
cumsum (*dseries method*), 285
cumsum(A[, d[, v]]) (*dseries method*), 285
cumsum_ (*dseries method*), 285
cumsum_(A[, d[, v]]) (*dseries method*), 285

## D

data (*command*), 151
data(OPTIONS...) (*command*), 151
database (*command*), 86
dataset_ (*MATLAB variable*), 17
define (*macro directive*), 244
detrend (*dseries method*), 286
detrend(A[, m]) (*dseries method*), 286
detrend_ (*dseries method*), 286
detrend_(A[, m]) (*dseries method*), 286
dgrowth (*dseries method*), 286
dgrowth(A) (*dseries method*), 286
dgrowth_ (*dseries method*), 286
dgrowth_(A) (*dseries method*), 286
diff (*dseries method*), 287
diff(A) (*dseries method*), 287
diff_ (*dseries method*), 287
diff_(A) (*dseries method*), 287
discretionary_policy (*command*), 191
discretionary_policy(OPTIONS...) (*command*), 191
disp (*dates method*), 268
disp(A) (*dates method*), 268
disp(A[, noname]) (*dseries method*), 287
display (*dates method*), 268
display(A) (*dates method*), 268
double (*dates method*), 268
double(A) (*dates method*), 268
dplot (*MATLAB command*), 336
dsample (*command*), 57
dseries (*class*), 276
dynare (*MATLAB command*), 11
dynare_version (*MATLAB command*), 252
dynasave (*command*), 240
dynasave(FILENAME) (*command*), 240
dynatype (*command*), 239
dynatype(FILENAME) (*command*), 239

## E

echo (*macro directive*), 247
echomacrovars (*macro directive*), 247
echomacrovars(save) (*macro directive*), 247
else (*macro directive*), 245
elseif (*macro directive*), 245
endfor (*macro directive*), 246
endif (*macro directive*), 245
endval (*block*), 40
endval(learnt_in=INTEGER|DATE) (*block*), 76
endval(OPTIONS...) (*block*), 40
eq (*dates method*), 269
eq(A, B) (*dates method*), 269
erf (*function*), 30
erf(x) (*function*), 30
erfc (*function*), 30
erfc(x) (*function*), 30
error (*macro directive*), 247
estimate (*x13 method*), 311
estimate(A, key, value[, key, value[, [...]]]) (*x13 method*), 311
estimated_params (*block*), 108
estimated_params(overwrite) (*block*), 108
estimated_params_bounds (*block*), 111
estimated_params_init (*block*), 111
estimated_params_init(OPTIONS...) (*block*), 111
estimated_params_remove (*block*), 111
estimation (*command*), 112
estimation(OPTIONS...) (*command*), 112
ESTIMATOR) (*command*), 167
evaluate_epilogue (*MATLAB command*), 217
evaluate_epilogue(ds, M_, oo_) (*MATLAB command*), 217
evaluate_planner_objective (*command*), 187
evaluate_planner_objective(OPTIONS...) (*command*), 187
exist (*dseries method*), 288
exist(A, varname) (*dseries method*), 288
exp (*dseries method*), 288
exp (*function*), 29
exp(A) (*dseries method*), 288
exp(x) (*function*), 29
exp_ (*dseries method*), 288
exp_(A) (*dseries method*), 288
EXPECTATION (*operator*), 28
extended_path (*command*), 96
extended_path(OPTIONS...) (*command*), 96
external_function (*command*), 30
external_function(OPTIONS...) (*command*), 30
extract (*dseries method*), 288
extract(A, B[, ...]) (*dseries method*), 288

## F

fill_ (*dseries method*), 290

fill_(name, v) (*dseries method*), 290
filter_initial_state (*block*), 152
filter_tunes (*block*), 150
firstdate (*dseries method*), 290
firstdate(A) (*dseries method*), 290
firstobservedperiod (*dseries method*), 290
firstobservedperiod(A) (*dseries method*), 290
flip (*dseries method*), 290
flip(A) (*dseries method*), 290
flip_ (*dseries method*), 290
flip_(A) (*dseries method*), 290
for (*macro directive*), 246
force (*x13 method*), 312
force(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
forecast (*command*), 178
forecast (*x13 method*), 311
forecast(A, key, value[, key, value[, [...]]]) (*x13 method*), 311
forecast(OPTIONS...) (*command*), 178
frequency (*dseries method*), 290
frequency(B) (*dseries method*), 290

## G

ge (*dates method*), 269
ge (*dseries method*), 290
ge(A, B) (*dates method*), 269
ge(A, B) (*dseries method*), 290
generate_trace_plots (*MATLAB command*), 333
generate_trace_plots(CHAIN_NUMBER) (*MATLAB command*), 333
get_conditional_forecast (*MATLAB command*), 184
get_conditional_forecast('TYPE', 'MOMENT', 'VARIABLE_NAME' [, 'VARIABLE_NAME']...) (*MATLAB command*), 184
get_conditional_forecast_dseries (*MATLAB command*), 184
get_conditional_forecast_dseries('TYPE', 'MOMENT') (*MATLAB command*), 184
get_conditional_forecast_dseries('TYPE', 'MOMENT', 'VARIABLE_NAME' [, 'VARIABLE_NAME']...) (*MATLAB command*), 184
get_irf (*MATLAB command*), 93
get_irf('EXOGENOUS_NAME' [, 'ENDOGENOUS_NAME']... ) (*MATLAB command*), 93
get_mean (*MATLAB command*), 60
get_mean('ENDOGENOUS_NAME' [, 'ENDOGENOUS_NAME']... ) (*MATLAB command*), 60
get_param_by_name (*MATLAB command*), 32
get_param_by_name('PARAMETER_NAME') (*MATLAB command*), 32
get_shock_skew_by_name (*MATLAB command*), 57
get_shock_skew_by_name('EXOGENOUS_NAME1', 'EXOGENOUS_NAME2', 'EXOGENOUS_NAME3') (*MATLAB command*), 57
get_shock_stderr_by_name (*MATLAB command*), 56
get_shock_stderr_by_name('EXOGENOUS_NAME') (*MATLAB command*), 56
get_smooth (*MATLAB command*), 144
get_smooth('VARIABLE_NAME' [, 'VARIABLE_NAME']...) (*MATLAB command*), 144
get_smooth_dseries (*MATLAB command*), 144
get_smooth_dseries('VARIABLE_NAME' [, 'VARIABLE_NAME']...) (*MATLAB command*), 144
get_update (*MATLAB command*), 145
get_update('VARIABLE_NAME' [, 'VARIABLE_NAME']...) (*MATLAB command*), 145
gt (*dates method*), 269
gt (*dseries method*), 290
gt(A, B) (*dates method*), 269
gt(A, B) (*dseries method*), 290

## H

hdiff (*dseries method*), 291
hdiff(A) (*dseries method*), 291
hdiff_ (*dseries method*), 291
hdiff_(A) (*dseries method*), 291
heterogeneity_compute_steady_state (*command*), 222
heterogeneity_compute_steady_state(OPTIONS...) (*command*), 222
heterogeneity_dimension (*command*), 217
heterogeneity_load_steady_state (*command*), 220
heterogeneity_load_steady_state(OPTIONS...) (*command*), 220
heterogeneity_simulate (*command*), 228
heterogeneity_simulate(OPTIONS...) (*command*), 228
heterogeneity_solve (*command*), 227
heterogeneity_solve(OPTIONS...) (*command*), 227
heteroskedastic_shocks (*block*), 55
heteroskedastic_shocks(overwrite) (*block*), 55
hgrowth (*dseries method*), 291
hgrowth(A) (*dseries method*), 291
hgrowth_ (*dseries method*), 291
hgrowth_(A) (*dseries method*), 291
history (*x13 method*), 312
history(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
histval (*block*), 43
histval(OPTIONS...) (*block*), 43
histval_file (*command*), 49
histval_file(OPTIONS...) (*command*), 49

homotopy_setup (*block*), 60
homotopy_setup(from_initval_to_endval) (*block*), 60
hooks (*config block*), 260
horzcat (*dates method*), 270
horzcat(A, B, C, ...) (*dates method*), 270
horzcat(A, B[, ...]) (*dseries method*), 291
hpcycle (*dseries method*), 292
hpcycle(A[, lambda]) (*dseries method*), 292
hpcycle_ (*dseries method*), 292
hpcycle_(A[, lambda]) (*dseries method*), 292
hpfilter (*dseries method*), 292
hpfilter(A, lambda[, ...]) (*dseries method*), 292
hptrend (*dseries method*), 293
hptrend(A[, lambda]) (*dseries method*), 293
hptrend_ (*dseries method*), 293
hptrend_(A[, lambda]) (*dseries method*), 293

## I

identification (*command*), 200
identification(OPTIONS...) (*command*), 200
identify (*x13 method*), 312
identify(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
if (*macro directive*), 245
ifdef (*macro directive*), 245
ifndef (*macro directive*), 245
include (*macro directive*), 244
includepath (*macro directive*), 244
inf (*constant*), 27
init2shocks(OPTIONS...) (*block*), 174
initial_condition_decomposition (*command*), 175
initial_condition_decomposition(OPTIONS...) (*command*), 175
initval (*block*), 39
initval(OPTIONS...) (*block*), 39
initval_file (*command*), 45
initval_file(OPTIONS...) (*command*), 45
insert (*dseries method*), 293
insert(A, B, I) (*dseries method*), 293
INTEGER) (*command*), 186
internals (*MATLAB command*), 334
intersect (*dates method*), 270
intersect(A, B) (*dates method*), 270
irf_calibration (*block*), 198
irf_calibration(OPTIONS...) (*block*), 198
isempty (*dates method*), 270
isempty(A) (*dates method*), 270
isequal (*dates method*), 270
isequal(A, B) (*dates method*), 270
isinf (*dseries method*), 294
isinf(A) (*dseries method*), 294
isnan (*dseries method*), 294
isnan(A) (*dseries method*), 294
isreal (*dseries method*), 294
isreal(A) (*dseries method*), 294

## L

lag (*dseries method*), 294
lag(A[, p]) (*dseries method*), 294
lag_ (*dseries method*), 294
lag_(A[, p]) (*dseries method*), 294
lastdate (*dseries method*), 295
lastdate(B) (*dseries method*), 295
lastobservedperiod (*dseries method*), 295
lastobservedperiod(A) (*dseries method*), 295
lastobservedperiods (*dseries method*), 296
lastobservedperiods(A) (*dseries method*), 296
le (*dates method*), 271
le (*dseries method*), 296
le(A, B) (*dates method*), 271
le(A, B) (*dseries method*), 296
lead (*dseries method*), 296
lead(A[, p]) (*dseries method*), 296
lead_ (*dseries method*), 296
lead_(A[, p]) (*dseries method*), 296
length (*dates method*), 271
length(A) (*dates method*), 271
lineartrend (*dseries method*), 297
lineartrend(A) (*dseries method*), 297
ln (*function*), 29
ln(x) (*function*), 29
load_params_and_steady_state (*command*), 252
load_params_and_steady_state(FILENAME) (*command*), 252
log (*dseries method*), 297
log(A) (*dseries method*), 297
log(x) (*function*), 29
log10 (*function*), 29
log10(x) (*function*), 29
log_ (*dseries method*), 297
log_(A) (*dseries method*), 297
log_trend_var (*command*), 26
lt (*dates method*), 271
lt (*dseries method*), 296
lt(A, B) (*dates method*), 271
lt(A, B) (*dseries method*), 296

## M

M_ (*MATLAB variable*), 17
M_.maximum_lag (*MATLAB variable*), 74
M_.maximum_lead (*MATLAB variable*), 74
M_.osr.param_bounds (*MATLAB variable*), 195
M_.osr.param_indices (*MATLAB variable*), 195
M_.osr.param_names (*MATLAB variable*), 194
M_.osr.variable_indices (*MATLAB variable*), 195
M_.osr.variable_weights (*MATLAB variable*), 195
M_.param_names (*MATLAB variable*), 31
M_.params (*MATLAB variable*), 31
M_.state_var (*MATLAB variable*), 94
markov_switching (*command*), 207
markov_switching(OPTIONS...) (*command*), 207
matched_irfs (*block*), 154
matched_irfs(overwrite) (*block*), 154
matched_irfs_weights (*block*), 155

matched_irfs_weights(overwrite) (*block*), 155
matched_moments (*block*), 153
max (*dates method*), 271
max (*function*), 30
max(a, b) (*function*), 30
max(A, B, C, ...) (*dates method*), 271
mdiff (*dseries method*), 298
mdiff(A) (*dseries method*), 298
mdiff_ (*dseries method*), 298
mdiff_(A) (*dseries method*), 298
mean (*dseries method*), 298
mean(A[, geometric]) (*dseries method*), 298
merge (*dseries method*), 298
merge(A, B[, legacy]) (*dseries method*), 298
metadata (*x13 method*), 312
metadata(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
method_of_moments (*command*), 156
method_of_moments(OPTIONS...) (*command*), 156
mgrowth (*dseries method*), 298
mgrowth(A) (*dseries method*), 298
mgrowth_ (*dseries method*), 298
mgrowth_(A) (*dseries method*), 298
min (*dates method*), 272
min (*function*), 30
min(a, b) (*function*), 30
min(A, B, C, ...) (*dates method*), 272
minus (*dates method*), 272
minus(A, B) (*dates method*), 272
model (*block*), 32
model(heterogeneity=NAME) (*block*), 218
model(OPTIONS...) (*block*), 32
model_comparison (*command*), 167
model_diagnostics (*command*), 64
MODEL_EXPR) (*command*), 22, 26
model_info (*command*), 64
model_info(OPTIONS...) (*command*), 64
model_local_variable (*command*), 26
model_options (*command*), 35
model_options(OPTIONS...) (*command*), 35
model_remove (*command*), 35
model_remove(TAGS...) (*command*), 35
model_replace (*block*), 36
model_replace(TAGS...) (*block*), 36
moment_calibration (*block*), 199
moment_calibration(OPTIONS...) (*block*), 199
mpower (*dseries method*), 300
mpower(A, B) (*dseries method*), 300
mrdivide (*dseries method*), 300
mrdivide(A, B) (*dseries method*), 300
ms_compute_mdd (*command*), 212
ms_compute_mdd(OPTIONS...) (*command*), 212
ms_compute_probabilities (*command*), 213
ms_compute_probabilities(OPTIONS...) (*command*), 213
ms_estimation (*command*), 209
ms_estimation(OPTIONS...) (*command*), 209
ms_forecast (*command*), 214
ms_forecast(OPTIONS...) (*command*), 214
ms_irf (*command*), 213
ms_irf(OPTIONS...) (*command*), 213
ms_simulation (*command*), 212
ms_simulation(OPTIONS...) (*command*), 212
ms_variance_decomposition (*command*), 215
ms_variance_decomposition(OPTIONS...) (*command*), 215
mshocks (*block*), 55
mshocks(learnt_in=INTEGER|DATE) (*block*), 77
mshocks(learnt_in=INTEGER|DATE,OPTIONS...) (*block*), 77
mshocks(OPTIONS...) (*block*), 55
mtimes (*dates method*), 272
mtimes(A, B) (*dates method*), 272

## N

NAME) (*command*), 217
nan (*constant*), 27
nanmean (*dseries method*), 301
nanmean(A[, geometric]) (*dseries method*), 301
nanstd (*dseries method*), 301
nanstd(A[, geometric]) (*dseries method*), 301
ndat (*dates method*), 272
ndat(A) (*dates method*), 272
ne (*dates method*), 272
ne(A, B) (*dates method*), 272
nobs (*dseries method*), 302
nobs(A) (*dseries method*), 302
node (*config block*), 257
normcdf (*function*), 30
normcdf(x) (*function*), 30
normcdf(x, mu, sigma) (*function*), 30
normpdf (*function*), 30
normpdf(x) (*function*), 30
normpdf(x, mu, sigma) (*function*), 30

## O

observation_trends (*block*), 107
observation_trends(observation_trend_base_period = DATE) (*block*), 107
observation_trends(observation_trend_base_period = INTEGER) (*block*), 107
occbin_constraints (*block*), 98
occbin_graph (*command*), 106
occbin_graph(OPTIONS...) (*command*), 106
occbin_setup (*command*), 100
occbin_setup(OPTIONS...) (*command*), 100
occbin_solver (*command*), 105
occbin_solver(OPTIONS...) (*command*), 105
occbin_write_regimes (*command*), 106
occbin_write_regimes(OPTIONS...) (*command*), 106
onesidedhpcycle (*dseries method*), 302
onesidedhpcycle(A[, lambda[, init]]) (*dseries method*), 302
onesidedhpcycle_ (*dseries method*), 302

onesidedhpcycle_(A[, lambda[, init]]) (*dseries method*), 302
onesidedhptrend (*dseries method*), 302
onesidedhptrend(A[, lambda[, init]]) (*dseries method*), 302
onesidedhptrend_ (*dseries method*), 302
onesidedhptrend_(A[, lambda[, init]]) (*dseries method*), 302
oo.dr.state_var (*MATLAB variable*), 94
oo_ (*MATLAB variable*), 17
oo_.autocorr (*MATLAB variable*), 91
oo_.conditional_forecast.cond (*MATLAB variable*), 183
oo_.conditional_forecast.initial_date (*MATLAB variable*), 183
oo_.conditional_forecast.uncond (*MATLAB variable*), 183
oo_.conditional_variance_decomposition (*MATLAB variable*), 92
oo_.conditional_variance_decomposition_ME (*MATLAB variable*), 93
oo_.contemporaneous_correlation (*MATLAB variable*), 93
oo_.convergence.geweke (*MATLAB variable*), 149
oo_.convergence.raftery_lewis (*MATLAB variable*), 150
oo_.deterministic_simulation.homotopy_linearization (*MATLAB variable*), 74
oo_.deterministic_simulation.homotopy_marginal_linearization (*MATLAB variable*), 74
oo_.deterministic_simulation.sim1 (*MATLAB variable*), 74
oo_.deterministic_simulation.sim2 (*MATLAB variable*), 75
oo_.deterministic_simulation.status (*MATLAB variable*), 74
oo_.dr (*MATLAB variable*), 93
oo_.dr.eigval (*MATLAB variable*), 64
oo_.dr.inv_order_var (*MATLAB variable*), 94
oo_.dr.order_var (*MATLAB variable*), 94
oo_.dsge_var.posterior_mode (*MATLAB variable*), 138, 148
oo_.endo_simul (*MATLAB variable*), 73, 98
oo_.exo_simul (*MATLAB variable*), 74, 98
oo_.exo_steady_state (*MATLAB variable*), 60
oo_.extended_path.status (*MATLAB variable*), 98
oo_.FilterCovariance (*MATLAB variable*), 145
oo_.Filtered_Variables_X_step_ahead (*MATLAB variable*), 143
oo_.FilteredVariables (*MATLAB variable*), 143
oo_.FilteredVariablesKStepAhead (*MATLAB variable*), 143
oo_.FilteredVariablesKStepAheadVariances (*MATLAB variable*), 143
oo_.FilteredVariablesShockDecomposition (*MATLAB variable*), 143
oo_.forecast (*MATLAB variable*), 179
oo_.forecast_shock_decomposition (*MATLAB variable*), 170
oo_.gamma_y (*MATLAB variable*), 92
oo_.heterogeneity (*MATLAB variable*), 222
oo_.heterogeneity.dr (*MATLAB variable*), 227
oo_.heterogeneity.dr.G (*MATLAB variable*), 227
oo_.heterogeneity.steady_state (*MATLAB variable*), 222
oo_.initial_exo_steady_state (*MATLAB variable*), 74
oo_.initial_steady_state (*MATLAB variable*), 74
oo_.irfs (*MATLAB variable*), 93
oo_.kurtosis (*MATLAB variable*), 91
oo_.MarginalDensity.LaplaceApproximation (*MATLAB variable*), 142
oo_.MarginalDensity.ModifiedHarmonicMean (*MATLAB variable*), 142
oo_.mean (*MATLAB variable*), 91
oo_.MeanForecast (*MATLAB variable*), 180
oo_.Model_Comparison (*MATLAB variable*), 168
oo_.mom.data_moments (*MATLAB variable*), 163
oo_.mom.gmm_mode (*MATLAB variable*), 164
oo_.mom.gmm_std_at_mode (*MATLAB variable*), 164
oo_.mom.irf_model_varobs (*MATLAB variable*), 165
oo_.mom.J_test (*MATLAB variable*), 165
oo_.mom.m_data (*MATLAB variable*), 164
oo_.mom.MarginalDensity.LaplaceApproximation (*MATLAB variable*), 167
oo_.mom.MarginalDensity.ModifiedHarmonicMean (*MATLAB variable*), 167
oo_.mom.model_moments (*MATLAB variable*), 164
oo_.mom.model_moments_params_derivs (*MATLAB variable*), 164
oo_.mom.posterior.metropolis (*MATLAB variable*), 166
oo_.mom.posterior.optimization (*MATLAB variable*), 165
oo_.mom.posterior_deciles (*MATLAB variable*), 167
oo_.mom.posterior_density (*MATLAB variable*), 166
oo_.mom.posterior_hpdinf (*MATLAB variable*), 166
oo_.mom.posterior_hpdsup (*MATLAB variable*), 166
oo_.mom.posterior_mean (*MATLAB variable*), 166
oo_.mom.posterior_median (*MATLAB variable*), 167
oo_.mom.posterior_mode (*MATLAB variable*), 166
oo_.mom.posterior_std (*MATLAB variable*), 166
oo_.mom.posterior_std_at_mode (*MATLAB variable*), 166
oo_.mom.posterior_variance (*MATLAB variable*), 167
oo_.mom.prior (*MATLAB variable*), 165
oo_.mom.prior_density (*MATLAB variable*), 166

oo_.mom.Q (*MATLAB variable*), 164
oo_.mom.smm_mode (*MATLAB variable*), 164
oo_.mom.smm_std_at_mode (*MATLAB variable*), 164
oo_.mom.verbose (*MATLAB variable*), 164
oo_.mom.weighting_info (*MATLAB variable*), 164
oo_.occbin.simul.linear (*MATLAB variable*), 106
oo_.occbin.simul.piecewise (*MATLAB variable*), 106
oo_.occbin.simul.regime_history (*MATLAB variable*), 106
oo_.occbin.simul.shocks_sequence (*MATLAB variable*), 106
oo_.occbin.simul.ys (*MATLAB variable*), 106
oo_.osr.objective_function (*MATLAB variable*), 194
oo_.osr.optim_params (*MATLAB variable*), 194
oo_.pfwee.shocks_info (*MATLAB variable*), 80
oo_.pfwee.terminal_info (*MATLAB variable*), 80
oo_.planner_objective_value.conditional (*MATLAB variable*), 188
oo_.planner_objective_value.conditional.steady_state (*MATLAB variable*), 188
oo_.planner_objective_value.conditional.zero_initial_condition (*MATLAB variable*), 188
oo_.planner_objective_value.unconditional (*MATLAB variable*), 188
oo_.PointForecast (*MATLAB variable*), 179
oo_.posterior.metropolis (*MATLAB variable*), 143
oo_.posterior.optimization (*MATLAB variable*), 142
oo_.posterior_density (*MATLAB variable*), 147
oo_.posterior_hpdinf (*MATLAB variable*), 147
oo_.posterior_hpdsup (*MATLAB variable*), 147
oo_.posterior_mean (*MATLAB variable*), 147
oo_.posterior_median (*MATLAB variable*), 148
oo_.posterior_mode (*MATLAB variable*), 147
oo_.posterior_std (*MATLAB variable*), 148
oo_.posterior_std_at_mode (*MATLAB variable*), 147
oo_.posterior_var (*MATLAB variable*), 148
oo_.PosteriorIRF.dsge (*MATLAB variable*), 144
oo_.PosteriorTheoreticalMoments (*MATLAB variable*), 146
oo_.realtime_conditional_shock_decomposition (*MATLAB variable*), 172
oo_.realtime_forecast_shock_decomposition (*MATLAB variable*), 172
oo_.realtime_shock_decomposition (*MATLAB variable*), 172
oo_.RecursiveForecast (*MATLAB variable*), 149
oo_.shock_decomposition (*MATLAB variable*), 170
oo_.skewness (*MATLAB variable*), 91
oo_.SmoothedMeasurementErrors (*MATLAB variable*), 144
oo_.SmoothedShocks (*MATLAB variable*), 144
oo_.SmoothedShockVariance (*MATLAB variable*), 146
oo_.SmoothedShockVariances (*MATLAB variable*), 145
oo_.SmoothedVariables (*MATLAB variable*), 144
oo_.Smoother.Constant (*MATLAB variable*), 146
oo_.Smoother.initial_date (*MATLAB variable*), 145
oo_.Smoother.loglinear (*MATLAB variable*), 146
oo_.Smoother.Shock_variance (*MATLAB variable*), 145
oo_.Smoother.State_uncertainty (*MATLAB variable*), 145
oo_.Smoother.SteadyState (*MATLAB variable*), 146
oo_.Smoother.Trend (*MATLAB variable*), 146
oo_.Smoother.TrendCoeffs (*MATLAB variable*), 146
oo_.Smoother.Variance (*MATLAB variable*), 145
oo_.SpectralDensity (*MATLAB variable*), 93
oo_.steady_state (*MATLAB variable*), 60
oo_.time (*MATLAB variable*), 18
oo_.trend_component.MODEL_NAME.CompanionMatrix (*MATLAB variable*), 231
oo_.UpdatedVariables (*MATLAB variable*), 145
oo_.var (*MATLAB variable*), 91
oo_.var.MODEL_NAME.CompanionMatrix (*MATLAB variable*), 230
oo_.var_list (*MATLAB variable*), 91
oo_.variance_decomposition (*MATLAB variable*), 92
oo_.variance_decomposition_ME (*MATLAB variable*), 92
oo_recursive_ (*MATLAB variable*), 17
optim_weights (*block*), 193
options_ (*MATLAB variable*), 17
osr (*command*), 191
osr(OPTIONS...) (*command*), 191
osr_params (*command*), 193
osr_params_bounds (*block*), 194
outlier (*x13 method*), 311
outlier(A, key, value[, key, value[, [...]]]) (*x13 method*), 311

## P

pac.estimate.iterative_ols (*MATLAB command*), 239
pac.estimate.iterative_ols(EQNAME, GUESS, DATA, RANGE) (*MATLAB command*), 239
pac.estimate.nls (*MATLAB command*), 239
pac.estimate.nls(EQNAME, GUESS, DATA, RANGE[, ALGO]) (*MATLAB command*), 239
pac.initialize (*MATLAB command*), 237
pac.initialize(NAME_OF_PAC_MODEL) (*MATLAB command*), 237
pac.update (*MATLAB command*), 237
pac.update(NAME_OF_PAC_MODEL) (*MATLAB command*), 237
pac_expectation (*operator*), 236

pac_model (*command*), 235
pac_model(OPTIONS...) (*command*), 235
pac_target_info (*block*), 236
pac_target_info(NAME_OF_PAC_MODEL) (*block*), 236
pac_target_nonstationary (*operator*), 237
parameters (*command*), 24
paths (*config block*), 261
perfect_foresight_controlled_paths (*block*), 81
perfect_foresight_controlled_paths(OPTIONS...) (*block*), 81
perfect_foresight_setup (*command*), 66
perfect_foresight_setup(OPTIONS...) (*command*), 66
perfect_foresight_solver (*command*), 67
perfect_foresight_solver(OPTIONS...) (*command*), 67
perfect_foresight_with_expectation_errors_setup (*command*), 78
perfect_foresight_with_expectation_errors_setup(OPTIONS...) (*command*), 78
perfect_foresight_with_expectation_errors_solver (*command*), 80
perfect_foresight_with_expectation_errors_solver(OPTIONS...) (*command*), 80
pickmdl (*x13 method*), 312
pickmdl(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
planner_objective (*command*), 187
plot (*dseries method*), 302
plot(A) (*dseries method*), 302
plot(A, B) (*dseries method*), 302
plot(A, B[, ...]) (*dseries method*), 302
plot(A[, ...]) (*dseries method*), 302
plot_conditional_forecast (*command*), 186
plot_shock_decomposition (*command*), 173
plot_shock_decomposition(OPTIONS...) (*command*), 173
plus (*dates method*), 273
plus(A, B) (*dates method*), 273
pop (*dates method*), 273
pop(A) (*dates method*), 273
pop(A, B) (*dates method*), 273
pop(A[, B]) (*dseries method*), 303
pop_ (*dates method*), 273
pop_(A) (*dates method*), 273
pop_(A, B) (*dates method*), 273
pop_(A[, B]) (*dseries method*), 303
posterior_function (*command*), 333
posterior_function(OPTIONS) (*command*), 333
predetermined_variables (*command*), 25
prefix (*dseries method*), 303
prefix(A, p) (*dseries method*), 303
prefix_ (*dseries method*), 303
prefix_(A, p) (*dseries method*), 303
print_bytecode_dynamic_model (*command*), 65
print_bytecode_static_model (*command*), 65
prior (*MATLAB command*), 335
prior_function (*command*), 333
prior_function(OPTIONS) (*command*), 333
projection (*dseries method*), 304
projection(A, info, periods) (*dseries method*), 304

## Q

qdiff (*dseries method*), 304
qdiff(A) (*dseries method*), 304
qdiff_ (*dseries method*), 304
qdiff_(A) (*dseries method*), 304
qgrowth (*dseries method*), 305
qgrowth(A) (*dseries method*), 305
qgrowth_ (*dseries method*), 305
qgrowth_(A) (*dseries method*), 305

## R

ramsey_constraints (*block*), 190
ramsey_model (*command*), 189
ramsey_model(OPTIONS...) (*command*), 189
ramsey_policy (*command*), 190
ramsey_policy(OPTIONS...) (*command*), 190
realtime_shock_decomposition (*command*), 170
realtime_shock_decomposition(OPTIONS...) (*command*), 170
regression (*x13 method*), 311
regression(A, key, value[, key, value[, [...]]]) (*x13 method*), 311
remove (*dates method*), 273
remove(A, B) (*dates method*), 273
remove_ (*dates method*), 273
remove_(A, B) (*dates method*), 273
rename (*dseries method*), 305
rename(A, newname) (*dseries method*), 305
rename(A, oldname, newname) (*dseries method*), 305
rename_ (*dseries method*), 305
rename_(A, newname) (*dseries method*), 305
rename_(A, oldname, newname) (*dseries method*), 305
resetops (*dseries method*), 306
resetops(A, ops) (*dseries method*), 306
resettags (*dseries method*), 306
resettags(A, tags) (*dseries method*), 306
resid (*command*), 44
resid(OPTIONS...) (*command*), 44
round (*dseries method*), 306
round(A[, n]) (*dseries method*), 306
round_ (*dseries method*), 306
round_(A[, n]) (*dseries method*), 306
rouwenhorst (*MATLAB command*), 226
rouwenhorst(rho, sigma, N, tol, max_iter) (*MATLAB command*), 226
rplot (*command*), 239
run (*x13 method*), 312
run(A) (*x13 method*), 312

## S


save (*dseries method*), 306
save(A, basename[, format]) (*dseries method*), 306
save_params_and_steady_state (*command*), 251
save_params_and_steady_state(FILENAME) (*command*), 251
sbvar (*command*), 208
sbvar(OPTIONS...) (*command*), 208
search (*MATLAB command*), 335
seats (*x13 method*), 312
seats(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
send_endogenous_variables_to_workspace (*MATLAB command*), 333
send_exogenous_variables_to_workspace (*MATLAB command*), 333
send_irfs_to_workspace (*MATLAB command*), 333
sensitivity (*command*), 195
sensitivity(OPTIONS...) (*command*), 195
set_dynare_seed (*command*), 251
set_dynare_seed('ALGORITHM', INTEGER) (*command*), 251
set_dynare_seed('clock') (*command*), 251
set_dynare_seed('default') (*command*), 251
set_dynare_seed('reset') (*command*), 251
set_dynare_seed(INTEGER) (*command*), 251
set_dynare_threads (*MATLAB command*), 337
set_dynare_threads(NAME_OF_MEX_FILE,INTEGER) (*MATLAB command*), 337
set_names (*dseries method*), 307
set_names(A, s1, s2, ...) (*dseries method*), 307
set_param_value (*MATLAB command*), 32
set_param_value('PARAMETER_NAME', MATLAB_EXPRESSION) (*MATLAB command*), 32
set_shock_skew_value (*MATLAB command*), 57
set_shock_skew_value('EXOGENOUS_NAME1', 'EXOGENOUS_NAME2', 'EXOGENOUS_NAME3', MATLAB_EXPRESSION) (*MATLAB command*), 57
set_shock_stderr_value (*MATLAB command*), 56
set_shock_stderr_value('EXOGENOUS_NAME', MATLAB_EXPRESSION) (*MATLAB command*), 56
setdiff (*dates method*), 274
setdiff(A, B) (*dates method*), 274
shock_decomposition (*command*), 168
shock_decomposition(OPTIONS...) (*command*), 168
shock_groups (*block*), 170
shock_groups(OPTIONS...) (*block*), 170
shock_paths (*block*), 83
shock_paths(OPTIONS...) (*block*), 83
shocks (*block*), 52
shocks(heterogeneity=NAME) (*block*), 219
shocks(learnt_in=INTEGER|DATE) (*block*), 75
shocks(learnt_in=INTEGER|DATE, overwrite) (*block*), 75
shocks(overwrite) (*block*), 52
shocks(surprise) (*block*), 99
shocks(surprise,overwrite) (*block*), 99
sign (*function*), 29
sign(x) (*function*), 29
simul (*command*), 73
simul(OPTIONS...) (*command*), 73
Simulated_time_series (*MATLAB variable*), 74, 98
sin (*function*), 29
sin(x) (*function*), 29
sinh (*function*), 29
sinh(x) (*function*), 29
size (*dseries method*), 307
size(A[, dim]) (*dseries method*), 307
slidingspans (*x13 method*), 312
slidingspans(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
smoother2histval (*command*), 186
smoother2histval(OPTIONS...) (*command*), 186
sort (*dates method*), 274
sort(A) (*dates method*), 274
sort_ (*dates method*), 274
sort_(A) (*dates method*), 274
spectrum (*x13 method*), 312
spectrum(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
sqrt (*function*), 29
sqrt(x) (*function*), 29
squeeze_shock_decomposition (*command*), 176
std (*dseries method*), 308
std(A[, geometric]) (*dseries method*), 308
steady (*command*), 57
steady(OPTIONS...) (*command*), 57
STEADY_STATE (*operator*), 28
steady_state_model (*block*), 62
stoch_simul (*command*), 86
stoch_simul(OPTIONS...) (*command*), 86
strings (*dates method*), 274
strings(A) (*dates method*), 274
subperiod (*dates method*), 274
subperiod(A) (*dates method*), 274
subsample (*dseries method*), 308
subsample(A, d1, d2) (*dseries method*), 308
suffix (*dseries method*), 308
suffix(A, s) (*dseries method*), 308
suffix_ (*dseries method*), 308
suffix_(A, s) (*dseries method*), 308
svar (*command*), 208
svar(OPTIONS...) (*command*), 208
svar_identification (*block*), 208


## T


tag (*dseries method*), 308
tag(A, a[, b, c]) (*dseries method*), 308
tan (*function*), 29

tan(x) (*function*), 29
tanh (*function*), 29
tanh(x) (*function*), 29
tex_rename (*dseries method*), 308
tex_rename(A, name, newtexname) (*dseries method*), 308
tex_rename(A, newtexname) (*dseries method*), 308
tex_rename_ (*dseries method*), 308
tex_rename_(A, name, newtexname) (*dseries method*), 308
tex_rename_(A, newtexname) (*dseries method*), 308
transform (*dseries method*), 309
transform (*x13 method*), 311
transform(A, key, value[, key, value[, [...]]]) (*x13 method*), 311
transform(A, varlist, fun, pfx) (*dseries method*), 309
transform_ (*dseries method*), 309
transform_(A, varlist, fun, pfx) (*dseries method*), 309
trend_component_model (*command*), 231
trend_component_model(OPTIONS...) (*command*), 231
trend_var (*command*), 26

## U

uminus (*dates method*), 275
uminus(A) (*dates method*), 275
union (*dates method*), 275
union(A, B, C, ...) (*dates method*), 275
unique (*dates method*), 275
unique(A) (*dates method*), 275
unique_ (*dates method*), 275
unique_(A) (*dates method*), 275
uplus (*dates method*), 275
uplus(A) (*dates method*), 275

## V

var (*command*), 22
var(log) (*command*), 22
var_expectation (*operator*), 233
var_expectation.initialize (*MATLAB command*), 234
var_expectation.initialize(NAME_OF_VAR_EXPECTATION_MODEL) (*MATLAB command*), 234
var_expectation.update (*MATLAB command*), 234
var_expectation.update(NAME_OF_VAR_EXPECTATION_MODEL) (*MATLAB command*), 234
var_expectation_model (*command*), 233
var_expectation_model(OPTIONS...) (*command*), 233
var_model (*command*), 229
var_model(OPTIONS...) (*command*), 229
var_remove (*command*), 25
varexo (*command*), 23
varexo_det (*command*), 24
varobs (*command*), 107
vertcat (*dates method*), 275
vertcat(A, B, C, ...) (*dates method*), 275
vertcat(A, B[, ...]) (*dseries method*), 309
vobs (*dseries method*), 310
vobs(A) (*dseries method*), 310

## W

write (*reporting method*), 325
write_latex_definitions (*MATLAB command*), 252
write_latex_dynamic_model (*command*), 36
write_latex_dynamic_model(OPTIONS) (*command*), 36
write_latex_original_model (*command*), 36
write_latex_original_model(OPTIONS) (*command*), 36
write_latex_parameter_table (*MATLAB command*), 252
write_latex_prior_table (*MATLAB command*), 253
write_latex_static_model (*command*), 37
write_latex_static_model(OPTIONS) (*command*), 37
write_latex_steady_state_model (*command*), 37

## X

x11 (*x13 method*), 312
x11(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
x11regression (*x13 method*), 312
x11regression(A, key, value[, key, value[, [...]]]) (*x13 method*), 312
x13 (*class*), 310

## Y

ydiff (*dseries method*), 310
ydiff(A) (*dseries method*), 310
ydiff_ (*dseries method*), 310
ydiff_(A) (*dseries method*), 310
year (*dates method*), 275
year(A) (*dates method*), 275
ygrowth (*dseries method*), 310
ygrowth(A) (*dseries method*), 310
ygrowth_ (*dseries method*), 310
ygrowth_(A) (*dseries method*), 310