Dynare Reference Manual

Release 4.6.1

Dynare team

Mar 12, 2020

Contents

Currently the development team of Dynare is composed of:

- Stéphane Adjemian (Université du Maine, Gains)
- Houtan Bastani (CEPREMAP)
- Michel Juillard (Banque de France)
- Sumudu Kankanamge (Toulouse School of Economics)
- Frédéric Karamé (Université du Maine, Gains and CEPREMAP)
- Dóra Kocsis (CEPREMAP)
- Junior Maih (Norges Bank)
- Ferhat Mihoubi (Université Paris-Est Créteil, Érudite and CEPREMAP)
- Willi Mutschler (University of Münster)
- Johannes Pfeifer (University of Cologne)
- Marco Ratto (European Commission, Joint Research Centre - JRC)
- Sébastien Villemot (CEPREMAP)

The following people used to be members of the team:

- Abdeljabar Benzougar
- Alejandro Buesa
- Fabrice Collard
- Assia Ezzeroug
- Stéphane Lhuissier
- George Perendia

CHAPTER 1

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. 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 of 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 todo this is to cite the present manual, as:

> Stéphane Adjemian, Houtan Bastani, Michel Juillard, Frédéric Karamé, Junior Maih, Ferhat Mihoubi, George Perendia, Johannes Pfeifer, Marco Ratto and Sébastien Villemot (2011), "Dynare: Reference Manual, Version 4," *Dynare Working Papers*, 1, CEPREMAP

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

```
@TechReport{Adjemianetal2011,
  author      = {Adjemian, St\'ephane and Bastani, Houtan and
                 Juillard, Michel and Karam\'e, Fr\'ederic and
                 Maih, Junior and Mihoubi, Ferhat and
                 Perendia, George and Pfeifer, Johannes and
                 Ratto, Marco and Villemot, S\'ebastien},
  title       = {Dynare: Reference Manual Version 4},
  year        = {2011},
  institution = {CEPREMAP},
  type        = {Dynare Working Papers},
  number      = {1},
}
```

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

CHAPTER 2

Installation and configuration

2.1 Software requirements

Packaged versions of Dynare are available for Windows (7, 8.1, 10), several GNU/Linux distributions (Debian, Ubuntu, Linux Mint, Arch Linux) and macOS 10.11 or later. 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 version 7.9 (R2009b) or above;
- GNU Octave version 4.2.1 or above, with the statistics package from Octave-Forge. Note however that the Dynare installers for Windows and macOS require 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, the Statistics Toolbox, the Control System Toolbox;
- If under Octave, the following Octave-Forge packages: `optim`, `io`, `control`.

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-4.x.y-win.exe` (where `4.x.y` is the version number), and follow the instructions. The default installation directory is `c:\dynare\4.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 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.

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

2.2.3 On macOS

To install Dynare for use with MATLAB, execute the automated installer called `dynare-4.x.y.pkg` (where *4.x.y* is the version number), and follow the instructions. The default installation directory is `/Applications/Dynare/4.x.y`. After installation, this directory will contain several sub-directories, among which are `matlab`, `mex`, and `doc`.

Note that 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*).

By default, the installer installs a version of GCC (for use with *use_dll*) in the installation directory, under the `.brew` folder. To do so, it also installs a version of Homebrew in the same folder and Xcode Command Line Tools (this is an Apple product) in a system folder.

All of this requires a bit of time and hard disk space. The amount of time it takes will depend on your computing power and internet connection. To reduce the time the Dynare installer takes, you can install Xcode Command Line Tools yourself (see *Prerequisites on macOS*). Dynare, Homebrew, and GCC use about 600 MB of disk space while the Xcode Command Line Tools require about 400 MB.

If you do not use the *use_dll* option, you have the choice to forgo the installation of GCC and hence Dynare will only take about 50 MB of disk space.

Dynare for Octave works with Octave installed via the package located here: https://octave-app.org.

2.2.4 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

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.

2.4 Configuration

2.4.1 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 `4.x.y` with the correct version number, type:

  ```
  >> addpath c:/dynare/4.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 `4.x.y` with the correct version number, type:

  ```
  >> addpath /Applications/Dynare/4.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.

2.4.2 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 "*4.x.y*" with the correct version number, type:

```
octave:1> addpath c:/dynare/4.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 Octave via https://octave-app.org, type:

```
octave:1> addpath /Applications/Dynare/4.x.y/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.

2.4.3 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 un-usable installation of Dynare.

CHAPTER 3

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 ith static model equation with respect to the jth 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 have 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 contains 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` will issue a `clear all` command to MATLAB (<R2015b) or Octave, thereby deleting all workspace variables and functions; this option instructs `dynare` not to clear the workspace. Note that starting with MATLAB 2015b `dynare` only deletes the global variables and the functions using persistent variables, in order to benefit from the JIT (Just In Time) compilation. In this case the option instructs `dynare` not to clear the globals and functions.

`onlyclearglobals`
By default, `dynare` will issue a `clear all` command to MATLAB versions before 2015b and to Octave, thereby deleting all workspace variables; this option instructs `dynare` to clear only the global variables (i.e. `M_`, `options_`, `oo_`, `estim_params_`, `bayestopt_`, and `dataset_`), leaving the other variables in the workspace.

`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 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 no create a logfile of this run in `FILENAME.log`. The default is to create the logfile.

output=dynamic|first|second|third
Instructs the preprocessor to output derivatives at the given order. Only works when `language=julia` has been passed.

language=matlab|julia
Instructs the preprocessor to write output for MATLAB or Julia. Default: MATLAB

params_derivs_order=0|1|2
When *identification*, *dynare_sensitivity* (with identification), or *estimation_cmd* 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.

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. 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.

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 waitbars 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 overriden 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 temporarly 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|mexw32|mexw64|mexmaci64|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 *The 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 *The 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_slave_open_mode`
Instructs Dynare to leave the connection to the slave 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 *The configuration file*, for more information about the configuration file.

`-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.

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 ]'
```

`-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`
Only useful with model option *`use_dll`*. Don't recompile the MEX files when running again the same model file and the lists of variables and the equations haven't changed. 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`
Tells Dynare that the model to be solved is stochastic. If no Dynare commands related to stochastic models (`stoch_simul`, `estimation`, ...) are present in the `.mod` file, Dynare understands by default that the model to be solved is deterministic.

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`. If they exist, `estim_params_`, `bayestopt_`, `dataset_`, `oo_recursive_` and `estimation_info` are saved in the same file.

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.

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 dynare called 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 gexecuted 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 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 `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 4

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]

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;
- 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*);

[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.

- VARIABLE_NAME (sometimes VAR_NAME) indicates a variable name starting with an alphabetical character and can't contain: '()+-*/^=!;:@#.' or accentuated characters;
- PARAMETER_NAME (sometimes PARAM_NAME) indicates a parameter name starting with an alphabetical character and can't contain: '()+-*/^=!;:@#.' or accentuated characters;
- 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;

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 `Sigma_e` to name your variable. Not conforming to this rule might yield hard-to-debug error messages or crashes. Second, to minimize interference with MATLAB or Octave functions that may be called by Dynare or user-defined steady state files, it is recommended to avoid using the name of MATLAB functions. In particular when working with 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_STR|NAME=QUOTED_STR)]...;
Command: **var**(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`.

Options

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.

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`.

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).

long_name = QUOTED_STR
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_STR
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_STR;`.

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_STR|NAME=QUOTED_STR)...];

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 ρa_{t-1}, while the innovation ε_t is to be declared with varexo.

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

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.

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_STR|NAME=QUOTED_STR)...]

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: **`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 ref:*conv* 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 or directly in an equation.

To declare a variable on-the-fly in an equation tag, simply state the type of variable to be declared (`endogenous`, `exogenous`, or `parameter` 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`, an `x`, or a `p`. 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.

[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.

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 or 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, 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.

`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:

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

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 rational for this mathematically unfounded definition, rely on the observation that the derivative of $\text{abs}(x)$ is equal to $\text{sign}(x)$ for any $x \neq 0$ in $\mathbb{R}$ and from the convention for the value of $\text{sign}(x)$ at $x = 0$).

Function: **sin**(x)

Function: **cos**(x)

Function: **tan**(x)

Function: **asin**(x)

Function: **acos**(x)

Function: **atan**(x)
Trigonometric functions.

Function: **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.

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 of 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.

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.

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;

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. 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 informations 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',mcp = 'r > -1.94478']
r = rho*r(-1) + (1-rho)*(gpi*Infl+gy*YGap) + e;

end;
```

More information on tags is available on the Dynare wiki.

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, i.e. a working `mex` command (see *Compiler installation* for more details). On MATLAB for Windows, you will need to also pass the compiler name at the command line. Using this option can result in faster simulations or estimations, at the expense of some initial compilation time.[3]

[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.

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 Dynare wiki 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. Only available with option `block`. Possible values:

`0`
All the endogenous variables are considered as feedback variables (Default).

`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 of 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 of 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.

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_VAR` will be created such that `AUX_DIFF_VAR=x-x(-1)`, and `x(+1)` will be replaced with `x+AUX_DIFF_VAR(+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 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 slave nodes when doing a parallel computation (see *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://archives.dynare.org/DynareWiki/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);
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;
```

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`**`(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,
- For a stochastic model, 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**(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 the 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 (see *steady_state_model*); 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 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, in the case of a stochastic model, 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).

The last type of auxiliary variables is 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`.

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 Dynare wiki 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...`).

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 preceeding 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.

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);
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;
```

(continues on next page)

(continued from previous page)

```
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_val`.

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 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 ;
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*).

Command: initval_file`(filename = FILENAME);`
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)

The command accepts three 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. Their length must be `periods + M_.maximum_lag + M_.maximum_lead`
- MAT-file (extension `.mat`): same as for M-files.
- Excel file (extension `.xls` or `.xlsx`): for each endogenous and exogenous, 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 (easily done via octave by typing '`pkg install -forge io`').

> **Warning:** The extension must be omitted in the command argument. Dynare will automatically figure out the extension and select the appropriate file type. If there are several files with the same name but different extensions, then the order of precedence is as follows: first `.m`, then `.mat`, `.xls` and finally `.xlsx`.

Command: **histval_file**(filename = FILENAME);

This command is equivalent to `histval`, except that it reads its input from a file, and is typically used in conjunction with `smoother2histval`.

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 normal distribution with zero mean, but it belongs to the user to specify the variability of these shocks. The non-zero elements of the matrix of variance-covariance of the shocks can be entered with the `shocks` command. Or, the entire matrix can be directly entered with `Sigma_e` (this use is however deprecated).

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.

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 VARIABLE_NAME;
periods INTEGER[:INTEGER] [[,] INTEGER[:INTEGER]]...;
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.

Example (with scalar values)

```
shocks;
```

(continues on next page)

(continued from previous page)

```
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;
```

In stochastic context

For stochastic simulations, the `shocks` block specifies the non zero elements of the covariance matrix 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;
  ```

In an estimation context, it is also possible to specify variances and covariances 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;
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`(overwrite);`

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 (as given by the last `initval` or `endval` block).

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.

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

Special variable: Sigma_e

This special variable specifies directly the covariance matrix of the stochastic shocks, as an upper (or lower) triangular matrix. Dynare builds the corresponding symmetric matrix. Each row of the triangular matrix, except the last one, must be terminated by a semi-colon ;. For a given element, an arbitrary *EXPRESSION* is allowed (instead of a simple constant), but in that case you need to enclose the expression in parentheses. The order of the covariances in the matrix is the same as the one used in the `varexo` declaration.

Example

```
varexo u, e;

Sigma_e = [ 0.81 (phi*0.9*0.009);
            0.000081];
```

This sets the variance of `u` to 0.81, the variance of `e` to 0.000081, and the correlation between `e` and `u` to `phi`.

Warning: The use of this special variable is deprecated and is strongly discouraged. You should use a `shocks` block instead.

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 exgonous 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.

4.9 Other general declarations

Command: **dsample** INTEGER [INTEGER];
Reduces the number of periods considered in subsequent output commands.

Command: **periods** INTEGER
This command is now deprecated (but will still work for older model files). It is not necessary when no simulation is performed and is replaced by an option `periods` in `perfect_foresight_setup`, `simul` and `stoch_simul`.

This command sets the number of periods in the simulation. The periods are numbered from 1 to INTEGER. In perfect foresight simulations, it is assumed that all future events are perfectly known at the beginning of period 1.

Example

```
periods 100;
```

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 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)`

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 Dynare's own nonlinear equation solver (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) (requires `bytecode` option, see *Model declaration*).

6

Newton algorithm with a sparse LU solver at each iteration (requires `bytecode` and/or `block` option, see *Model declaration*).

7

Newton algorithm with a Generalized Minimal Residual (GMRES) solver at each iteration (requires `bytecode` and/or `block` option, see *Model declaration*).

8

Newton algorithm with a Stabilized Bi-Conjugate Gradient (BICGSTAB) solver at each iteration (requires bytecode and/or block option, see *Model declaration*).

9

Trust-region algorithm on the entire model.

10

Levenberg-Marquardt mixed complementarity problem (LMMCP) solver (*Kanzow and Petra (2004)*).

11

PATH mixed complementarity problem solver of *Ferris and Munson (1999)*. The complementarity conditions are specified with an `mcp` equation tag, 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.

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:

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.

`homotopy_steps = INTEGER`
Defines the number of steps when performing a homotopy. See `homotopy_mode` option for more details.

`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.

`markowitz = DOUBLE`
Value of the Markowitz criterion, used to select the pivot. Only used when `solve_algo = 5`. Default: 0.5.

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 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 ;`
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 finding 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 preceeding `initval` 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` block).

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;
```

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;

...
// parameter calibration, (dynamic) model declaration, 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);
```

(continues on next page)

(continued from previous page)

```
  nust = psi*mst^2/( (1-alp)*(1-psi)*bet*gst^alp*khst^alp );

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

  l   = psi*mst*n/( (1-psi)*(1-n) );
  c   = mst/P;
  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;
```

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 normalization of the model: an endogenous variable is attributed to each equation of the model;
- The block structure of the model: for each block `model_info` indicates its type, the equations number and endogenous variables belonging to this block.

This command can only be used in conjunction with the `block` option of the `model` 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 endogenous variable attributed to the equation appears currently 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 endogenous variable attributed to the equation appears currently 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 endogenous variable attributed to the equation does not appear currently on the left hand side and where no forward looking endogenous variables appear. The block has the form: $g_j(y_{j,t}, y_t, y_{t-1}, \ldots, y_{t-k}) = 0$. x is equal to 'SIMPLE' if the block has only one equation. If several equation appears in the block, x is equal to 'COMPLETE'.

- 'SOLVE FORWARD x'

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

Options

'static'
: Prints out the block decomposition of the static model. Without 'static' option model_info displays the block decomposition of the dynamic model.

'incidence'
: Displays the gross incidence matrix and the reordered incidence matrix of the block decomposed model.

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.

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.

4.12 Deterministic simulation

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 nonlinar 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=6`) is based on a Newton-type algorithm first proposed by *Laffargue (1990)* and *Boucekkine (1995)*, which uses relaxation techniques. Thereby, the algorithm 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.

Command: perfect_foresight_setup ;
Command: perfect_foresight_setup`(OPTIONS...);`
: Prepares a perfect foresight simulation, by extracting the information in the `initval`, `endval` and `shocks` 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.

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

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.

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
Don't print anything. Useful for loops.

print
Print results (opposite of `noprint`).

stack_solve_algo = INTEGER
Algorithm used for computing the solution. Possible values are:

`0`
: Newton method to solve simultaneously all the equations for every period, using sparse matrices (Default).

`1`
: Use a Newton algorithm with a sparse LU solver at each iteration (requires `bytecode` and/or `block` option, see *Model declaration*).

`2`
: Use a Newton algorithm with a Generalized Minimal Residual (GMRES) solver at each iteration (requires `bytecode` and/or `block` option, see *Model declaration*)

`3`
: Use a Newton algorithm with a Stabilized Bi-Conjugate Gradient (BICGSTAB) solver at each iteration (requires `bytecode` and/or `block` option, see *Model declaration*).

`4`
: Use a Newton algorithm with a optimal path length at each iteration (requires `bytecode` and/or `block` option, see *Model declaration*).

5

Use a Newton algorithm with a sparse Gaussian elimination (SPE) solver at each iteration (requires `bytecode` option, see *Model declaration*).

6

Use the historical algorithm proposed in *Juillard (1996)*: it is slower than `stack_solve_algo=0`, but may be less memory consuming on big models (not available with `bytecode` and/or `block` options).

7

Allows the user to solve the perfect foresight model with the solvers available through option `solve_algo` (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.

robust_lin_solve
Triggers the use of a robust linear solver for the default `stack_solve_algo=0`.

solve_algo
See *solve_algo*. Allows selecting the solver used with `stack_solve_algo=7`.

no_homotopy
By default, the perfect foresight solver uses a homotopy technique if it cannot solve the problem. Concretely, it divides the problem into smaller steps by diminishing the size of shocks and increasing them progressively until the problem converges. This option tells Dynare to disable that behavior. Note that the homotopy is not implemented for purely forward or backward models.

markowitz = DOUBLE
Value of the Markowitz criterion, used to select the pivot. 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 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 requires a particular model setup as the goal is to get rid of any min/max operators and complementary slackness conditions that might introduce a singularity into the Jacobian. This is done by attaching an equation tag (see *Model declaration*) with the `mcp` keyword to affected equations. This tag states that the equation to which the tag is attached has to hold unless the expression within the tag is binding. For instance, a ZLB on the nominal interest rate would be specified as follows in the model block:

```
model;
   ...
   [mcp = 'r > -1.94478']
   r = rho*r(-1) + (1-rho)*(gpi*Infl+gy*YGap) + e;
   ...
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 `mcp` tag also avoids using `max(r,-1.94478)` for other occurrences of `r` in the rest of the model. It is important to keep in mind that, because the `mcp` tag effectively replaces a complementary slackness condition, it cannot be simply attached to any equation. Rather, it must be attached to the correct affected equation as otherwise the solver will solve a different problem than originally intended. Also, since the problem to be solved is nonlinear, the sign of the residuals of the dynamic equation matters. In the previous example, for the nominal interest rate rule, if the LHS and RHS are reversed the sign of the residuals (the difference between the LHS and the RHS) will change and it may happen that solver fails to identify the solution path. More generally, convergence of the nonlinear solver is not guaranteed when using mathematically equivalent representations of the same equation.

Note that in the current implementation, the content of the `mcp` equation tag is not parsed by the preprocessor. The inequalities must therefore be as simple as possible: an endogenous variable, followed by a relational operator, followed by a number (not a variable, parameter or expression).

`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 option `stack_solve_algo==0`.

`linear_approximation`
Solves the linearized version of the perfect foresight model. The model must be stationary. Only available with option `stack_solve_algo==0`.

Output

The simulated endogenous variables are available in global matrix `oo_.endo_simul`.

`Command: simul ;`
`Command: simul``(OPTIONS...);`
Short-form command for triggering the computation of a deterministic simulation of the model. 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 `simul`) 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 `periods` option.

`MATLAB/Octave variable: oo_.exo_simul`
This variable stores the path of exogenous variables during a simulation (computed by `perfect_foresight_solver`, `simul`, `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`!

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 (2001a)*, *Collard and Juillard (2001b)*, 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 of the 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 on the Dynare wiki.

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.

The `stoch_simul` command with a first order approximation can benefit from the block decomposition of the model (see *block*).

Options

ar = INTEGER
Order of autocorrelation coefficients to compute and to print. Default: `5`.

drop = INTEGER
Number of points (burnin) 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).

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` is unavailable). 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*).

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`. Default: `1`.

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

aim_solver
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.

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. 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, Kim, Schaumburg and Sims (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, Kim, Schaumburg and Sims (2008)*, while at third order its generalization by *Andreasen, Fernández-Villaverde and Rubio-Ramírez (2018)* is used. Not available above third order.

partial_information
Computes the solution of the model under partial information, along the lines of *Pearlman, Currie and Levine (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 here .

sylvester = OPTION
Determines the algorithm used to solve the Sylvester equation for block decomposed model. Possible values for OPTION are:

`default`

Uses the default solver for Sylvester equations (`gensylv`) based on Ondra Kamenik's algorithm (see here for more information).

`fixed_point`

Uses a fixed point algorithm to solve the Sylvester equation (`gensylv_fp`). This method is faster than the default one for large scale models.

Default value is `default`.

sylvester_fixed_point_tol = DOUBLE
The convergence criterion used in the fixed point Sylvester solver. Its default value is `1e-12`.

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 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 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`.

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_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.

hp_ngrid = INTEGER
Deprecated option. It has the same effect as *filtered_theoretical_moments_grid*.

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*), and also saves the simulated variables in MATLAB/Octave vectors of the global workspace with the same name as the endogenous variables.

If option `irf` is different from zero, sets `oo_.irfs` (see below) and also saves the IRFs in MATLAB/Octave vectors of the global workspace (this latter way of accessing the IRFs is deprecated and will disappear in a future version).

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 (or an approximation thereof for `order=2`), and simulated variance otherwise. 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 (or an approximation thereof for `order=2`), and simulated autocorrelations otherwise. 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 than `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.

In case `order=2`, the theoretical second moments are a second order accurate approximation of the true second moments, see conditional_variance_decomposition.

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 conductedof 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 (or an approximation thereof for `order=2`), and simulated contemporaneous correlations otherwise. 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 2π 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: get_irf`('EXOGENOUS_NAME' [, 'ENDOGENOUS_NAME']... );`
Given the name of an exogenous variables, returns the IRFs for the requested endogenous variable(s), as they are 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.

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.

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 a gaussian quadrature to take into account the effects of future uncertainty. If `order` $= S$ then the time series for the endogenous variables are generated by assuming that the agents believe that there will no more shocks after period $t + S$. This is an experimental feature and can be quite slow. Default: `0`.

hybrid

Use the constant of the second order perturbation reduced form to correct the paths generated by the (stochastic) extended path algorithm.

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
```

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

Vector of numerical indices identifying the state variables in the vector of declared variables. `M_.endo_names(M_.state_var)` therefore yields the name of all variables that are states in the model declaration, i.e. that show up with a lag.

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 variables 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`.

4.13.3 First-order approximation

The approximation has the stylized form:

$$y_t = y^s + Ay^h_{t-1} + Bu_t$$

where y^s is the steady state value of y and $y^h_t = y_t - y^s$.

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 may differ from `M_.state_var` in case a state variable drops from the model given the current parameterization, because it only gets 0 coefficients in the decision rules. See *M_.state_var*.

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.
- 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^h_t. 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:

$$y_t = y^s + 0.5\Delta^2 + Ay^h_{t-1} + Bu_t + 0.5C(y^h_{t-1} \otimes y^h_{t-1}) + 0.5D(u_t \otimes u_t) + E(y^h_{t-1} \otimes u_t)$$

where y^s is the steady state value of y, $y_t^h = y_t - y^s$, and Δ^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:

- Δ^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.

4.14 Estimation

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 *Rabanal and Rubio-Ramirez (2003)*, *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.

The estimation using a first order approximation can benefit from the block decomposition of the model (see *block*).

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_cmd*).

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 ;**

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.

Each line inside of 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;
```

Block: **estimated_params ;**

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 estimation, each line follows this syntax:

```
stderr VARIABLE_NAME | corr VARIABLE_NAME_1, VARIABLE_NAME_2 | PARAMETER_NAME
, INITIAL_VALUE [, LOWER_BOUND, UPPER_BOUND ];
```

In a Bayesian estimation, each line follows this syntax:

```
stderr VARIABLE_NAME | corr VARIABLE_NAME_1, VARIABLE_NAME_2 | 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 four 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.

- `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).

`UPPER_BOUND`
Same as `lower_bound`, but specifying an upper bound instead.

`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 and for the uniform distribution. Default: `0`.

PRIOR_4TH_PARAMETER
A fourth parameter of the prior used for generalized beta distribution and for the uniform distribution. Default: `1`.

SCALE_PARAMETER
A parameter specific scale parameter for the jumping distribution's covariance matrix of the Metropolis-Hasting algorithm.

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. Hence, the prior does not integrate to `1` anymore.

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;
```

Block: estimated_params_init ;
Block: estimated_params_init(OPTIONS...);
This block declares numerical initial values for the optimizer when these ones 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 | 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 | PARAMETER_NAME,␣
↪LOWER_BOUND, UPPER_BOUND;
```

See *estimated_params*, for the meaning and syntax of the various components.

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 *Pfeiffer (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` and `consider_only_observed` options below). 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.

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. If *mh_nblocks* is equal to one, the convergence diagnostics of *Geweke (1992,1999)* is computed. 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 used instead. 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 *Giordano et al.(2011)* based on Parzen windows as in e.g. *Andrews (1991)*.

Options

datafile = FILENAME
: The datafile: a `.m` file, a `.mat` file, a `.csv` file, or a `.xls/.xlsx` file (under Octave, the io package from Octave-Forge is required for the `.csv` and `.xlsx` formats 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',...);
```

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 = NAME
: The name of the sheet with the data in an Excel file.

xls_range = RANGE
: The range with the data in an Excel file. For example, `xls_range=B2:D200`.

nobs = INTEGER
: The number of observations following `first_obs` to be used. Default: all observations in the file 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
The number 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.

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).

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*.

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.

lik_algo = INTEGER
For internal use and testing only.

conf_sig = DOUBLE
Confidence interval used for classical forecasting after estimation. See *conf_sig*.

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 Metropolis-Hastings algorithm. For the time being, `mh_replic` should be larger than 1200. Default: `20000`.

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 overriden by option `sub_draws`. Default: `1200`.

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). The default value is rarely satisfactory. 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`. 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: `0.2`.

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`.

mh_init_scale = DOUBLE

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_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.

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` 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_mode = INTEGER

...

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`. 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.

mode_compute = INTEGER | FUNCTION_NAME

Specifies the optimizer for the mode computation:

0

The mode isn't computed. When the `mode_file` option is specified, the mode is simply read from that file.

When `mode_file` 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` 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 from Octave-Forge, 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 from Octave-Forge 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 optimization routine (see Dynare wiki 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 from Octave-Forge 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, Salcedo and Feyo de Azevedo (1996)*.

11

This is not strictly speaking an optimization algorithm. The (estimated) parameters are treated as state variables and estimated 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.

`12`

Uses the `particleswarm` optimization routine (available under MATLAB if the Global Optimization Toolbox is installed; not available under Octave).

`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 `4`.

`silent_optimizer`
Instructs Dynare to run mode computing/optimization silently without displaying results or saving files in between. Useful when running loops.

`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`.

`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`.

prior_trunc = DOUBLE

Probability of extreme values of the prior density that is ignored when computing bounds for the parameters. Default: `1e-32`.

huge_number = DOUBLE

Value for replacing infinite values in the definition of (prior) bounds when finite values are required for computational reasons. Default: `1e7`.

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` 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.

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`

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 (1964)*). 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.

`'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. Default is `'previous'` if option `mode_file` is used, `'prior'` otherwise.

`'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.

`'MaxFunvEvalFactor'`

Set `MaxFunvEvals` equal to `MaxFunvEvalFactor` 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`.

`'MaxFunvEvals'`

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 stepsize 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),...);
```

nodiagnostic
Does not compute the convergence diagnostics for Metropolis-Hastings. Default: diagnostics are computed and displayed.

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).

relative_irf
See *relative_irf*.

dsge_var = DOUBLE
Triggers the estimation of a DSGE-VAR model, where the weight of the DSGE prior of the VAR model is calibrated to the value passed (see *Del Negro and Schorfheide (2004)*). It 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.

NB: The previous method of declaring `dsge_prior_weight` as a parameter and then calibrating it is now deprecated and will be removed in a future release of Dynare. Some of objects arising during estimation are stored with their values at the mode in `oo_.dsge_var.posterior_mode`.

`dsge_var`

Triggers the estimation of a DSGE-VAR model, where the weight of the DSGE prior of the VAR model will be estimated (as in *Adjemian et al.(2008)*). The prior on the weight of the DSGE prior, `dsge_prior_weight`, must be defined in the `estimated_params` section.

NB: The previous method of declaring `dsge_prior_weight` as a parameter and then placing it in `estimated_params` is now deprecated and will be removed in a future release of Dynare.

`dsge_varlag = INTEGER`

The number of lags used to estimate a DSGE-VAR model. Default: `4`.

`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.

`'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, Ratto, and Rossi (2015)*. Note that `'slice'` is incompatible with `prior_trunc=0`.

`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`

`'independent_metropolis_hastings'`

Takes the same options as in the case of `random_walk_metropolis_hastings`.

`'slice'`

`'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_tmp_file'`

See *save_tmp_file*. Default: `1`.

`'tailored_random_block_metropolis_hastings'`

`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`.

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.

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)

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.

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.

tex
See *tex*.

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.

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.

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 not yet compatible with *analytic_derivation*.

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_∞) and the rank of the covariance matrix of the non-stationary state variables (P_∞) 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.

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`.

filter_decomposition
Triggers the computation of the shock decomposition of the above k-step ahead filtered values. Stores results in `oo_.FilteredVariablesShockDecomposition`.

smoothed_state_uncertainty
Triggers the computation of the variance of smoothed estimates, i.e. $var_T(y_t)$. Stores results in `oo_.Smoother.State_uncertainty`.

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.

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).

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.

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`.

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 *Fernandez-Villaverde and Rubio-Ramirez (2005)*). Default is `1`, i.e. the likelihood of the linearized model is evaluated using a standard Kalman filter.

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.

aim_solver
See *aim_solver*.

sylvester = OPTION
See *sylvester*.

sylvester_fixed_point_tol = DOUBLE
See sylvester_fixed_point_tol .

lyapunov = OPTION
Determines the algorithm used to solve the Lyapunov equation to initialized the variance-covariance matrix of the Kalman filter using the steady-state value of state variables. 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 from Octave-Forge 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.

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 pratice, 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. The final hessian is also computed analytically. Only works for stationary models without missing observations, i.e. for kalman_algo<3.

ar = INTEGER
See *ar*. Only useful in conjunction with option moments_varendo.

endogenous_prior
Use endogenous priors as in *Christiano, Trabandt and Walentin (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, Trabandt and Walentin (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).

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`.

keep_kalman_algo_if_singularity_is_detected
With the default *use_univariate_filters_if_singularity_is_detected=1*, Dynare will switch to the univariate Kalman filter when it encounters a singular forecast error variance matrix during Kalman filtering. Upon encountering such a singularity for the first time, all subsequent parameter draws and computations will automatically rely on univariate filter, i.e. Dynare will never try the multivariate filter again. Use the `keep_kalman_algo_if_singularity_is_detected` option to have the `use_univariate_filters_if_singularity_is_detected` only affect the behavior for the current draw/computation.

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.

qz_zero_threshold = DOUBLE
See *qz_zero_threshold*.

taper_steps = [INTEGER1 INTEGER2 ...]
Percent tapering used for the spectral window in the *Geweke (1992,1999)* 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]`.

geweke_interval = [DOUBLE DOUBLE]
Percentage of MCMC draws at the beginning and end of the MCMC chain taken to compute the *Geweke (1992,1999)* convergence diagnostics (requires `mh_nblocks=1`) after discarding the first `mh_drop = DOUBLE` 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 deliver 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 burnin (`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]`.

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_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.

number_of_particles = INTEGER
Number of particles used when evaluating the likelihood of a non linear state space model. Default: `1000`.

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`.

filter_algorithm = OPTION
Sets the particle filter algorithm. Possible values for OPTION are:

`sis`

Sequential importance sampling algorithm, this is the default value.

`apf`

Auxiliary particle filter.

`gf`

Gaussian filter.

`gmf`

Gaussian mixture filter.

`cpf`

Conditional particle filter.

`nlkf`

Use a standard (linear) Kalman filter algorithm with the nonlinear measurement and state equations.

proposal_approximation = OPTION
Sets the method for approximating the proposal distribution. Possible values for OPTION are: `cubature`, `montecarlo` and `unscented`. Default value is `unscented`.

distribution_approximation = OPTION
Sets the method for approximating the particle distribution. Possible values for OPTION are: `cubature`, `montecarlo` and `unscented`. Default value is `unscented`.

cpf_weights = OPTION

Controls the method used to update the weights in conditional particle filter, possible values are `amisanotristani` (*Amisano et al. (2010)*) or `murrayjonesparslow` (*Murray et al. (2013)*). Default value is `amisanotristani`.

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).

Note

If no `mh_jscale` parameter is used for a parameter in `estimated_params`, the procedure uses `mh_jscale` for all parameters. If `mh_jscale` option isn't set, the procedure uses `0.2` for all parameters. Note that 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.

"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.

Output

After running estimation, the parameters `M_.params` and the variance matrix `M_.Sigma_e` of the shocks are set to the mode for maximum likelihood estimation or posterior mode computation without Metropolis iterations. After estimation with Metropolis iterations (option `mh_replic > 0` or option `load_mh_file` set) the parameters `M_.params` and the variance matrix `M_.Sigma_e` of the shocks 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. (2010)*, p.17).

[4] See option *conf_sig* to change the size of the HPD interval.
[5] See option *conf_sig* to change the size of the HPD interval.

`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`

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 *Skoeld 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.

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 `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 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) 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.SteadyState
Variable set by the estimation command (if used with the smoother) 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) 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 theh 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
```

[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_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. Fields are of the form:

```
oo_.posterior_mode.ESTIMATED_OBJECT.VARIABLE_NAME
```

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 *Del Negro and Schorfheide (2004)*).

`SIGMA_u_tilde`

Posterior covariance matrix of the DSGE-BVAR at the mode (equation (29) of *Del 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 *Del Negro and Schorfheide (2004)*).

`SIGMA_star`

Prior covariance matrix of the DSGE-BVAR at the mode (equation (23) of *Del Negro and Schorfheide (2004)*).

`ArtificialSampleSize`

Size of the artifical 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 (Γ^*_{xx} in *Del 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 (see *nobs*).

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 when used with `mh_nblocks=1` option (see *mh_nblocks*).

Fields are of the form:

[7] See *forecast* for more information.

```
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.

`nse_x`

Numerical standard error (NSE) when using an x% taper.

`rne_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.

Command: **`unit_root_vars`** `VARIABLE_NAME...;`

This command is deprecated. Use `estimation` option `diffuse_filter` instead for estimating a model with non-stationary observed variables or `steady` option `nocheck` to prevent `steady` to check the steady state returned by your steady state file.

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.

4.15 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.

4.16 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 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`
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*.

`use_shock_groups [= STRING]`
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.

`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*).

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.

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.

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 $[T+1, \ldots, \texttt{nobs}]$. 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 [= STRING]
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 $T = [\texttt{presample+1}\ldots\texttt{nobs}]$.

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
Runs the smoother only twice: once for the last in-sample and once for the last out-of-sample data point, where the provided integer defines the last observation (equivalent to `nobs`). Default: not enabled.

with_epilogue
See *with_epilogue*.

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 $t = [T \ldots T+k]$. 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 [= STRING]
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]

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

[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.

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);
```

(continues on next page)

(continued from previous page)

```
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.[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.

4.17 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`.

Options

datafile = FILENAME
See *datafile*.

filtered_vars
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
See *first_obs*.

filter_decomposition
See *filter_decomposition*.

diffuse_filter = INTEGER
See *diffuse_filter*.

diffuse_kalman_tol = DOUBLE
See *diffuse_kalman_tol*.

xls_sheet = NAME
See *xls_sheet*.

xls_range = RANGE
See *xls_range*.

4.18 Forecasting

On a calibrated model, forecasting is done using the `forecast` command. On an estimated model, use the `forecast` option of `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;
```

(continues on next page)

(continued from previous page)

```
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 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.

`HPDsup`

Upper bound of a 90% HPD forecast interval due to parameter uncertainty, but ignoring the effect of measurement error on observed variables.

`HPDinf_ME`

Lower bound of a 90% HPD interval[10] of forecast for observed variables due to parameter uncertainty and measurement error.

`HPDsup_ME`

Upper bound of a 90% HPD interval of forecast for observed variables due to parameter uncertainty and measurement error.

`Mean`

Mean of the posterior distribution of forecasts.

`Median`

Median of the posterior distribution of forecasts.

`Std`

Standard deviation of the posterior distribution of forecasts.

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:

[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.

```
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 an 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 ε_t are split up into controlled and uncontrolled ones to get:

$$\begin{aligned} y_t(contr\_vars) = Ty_{t-1}(contr\_vars) + R(contr\_vars, uncontr\_shocks)\varepsilon_t(uncontr\_shocks) \\ +R(contr\_vars, contr\_shocks)\varepsilon_t(contr\_shocks) \end{aligned}$$

which can be solved algebraically for $\varepsilon_t(contr\_shocks)$.

Using these controlled shocks, the state-space representation can be used for forecasting. A few things need to be noted. First, it is assumed that controlled exogenous variables are fully under control of the policy maker for all forecast periods and not just for the periods where the endogenous variables are controlled. For all uncontrolled periods, the controlled exogenous variables are assumed to be 0. This implies that there is no forecast uncertainty arising from these exogenous variables in uncontrolled periods. Second, 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. Third, although controlled exogenous variables are taken as instruments perfectly under the control of the policy-maker, they are nevertheless random and unforeseen shocks from the perspective of the households. That is, households are in each period surprised by the realization of a shock that keeps the controlled endogenous variables at their respective level. Fourth, keep in mind that 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 factorize 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, and their constrained future path. Option `controlled_varexo` is used to specify the structural shocks which will be matched to generate the constrained path.

Use *plot_conditional_forecast* to graph the results.

Options

parameter_set = OPTION

See *parameter_set* for possible values. No default value, mandatory option.

controlled_varexo = (VARIABLE_NAME...)

Specify the exogenous variables to use as control variables. No default value, mandatory option.

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. Default: `5000`.

conf_sig = DOUBLE

Level of significance for confidence interval. Default: `0.80`.

Output

The results are stored in `oo_.conditional_forecast`, which is described below.

Example

```
var y a;
varexo e u;
...
estimation(...);

conditional_forecast_paths;
var y;
periods 1:3, 4:5;
values 2, 5;
var a;
periods 1:5;
values 3;
end;

conditional_forecast(parameter_set = calibration, controlled_varexo =␣
↪(e, u), replic = 3000);

plot_conditional_forecast(periods = 10) a y;
```

MATLAB/Octave variable: oo_.conditional_forecast.cond
Variable set by the `conditional_forecast` command. It stores the conditional forecasts. Fields are `periods+1` by `1` vectors storing the steady state (time 0) and the subsequent `periods` forecasts periods. Fields are of the form:

```
oo_.conditional_forecast.cond.FORECAST_MOMENT.VARIABLE_NAME
```

where FORECAST_MOMENT is one of the following:

`Mean`

Mean of the conditional forecast distribution.

`ci`

Confidence interval of the conditional forecast distribution. The size corresponds to `conf_sig`.

MATLAB/Octave variable: oo_.conditional_forecast.uncond
Variable set by the `conditional_forecast` command. It stores the unconditional forecasts. Fields are of the form:

```
oo_.conditional_forecast.uncond.FORECAST_MOMENT.VARIABLE_NAME
```

MATLAB/Octave variable: forecasts.instruments
Variable set by the `conditional_forecast command`. Stores the names of the exogenous instruments.

MATLAB/Octave variable: oo_.conditional_forecast.controlled_variables
Variable set by the `conditional_forecast` command. Stores the position of the constrained endogenous variables in declaration order.

MATLAB/Octave variable: oo_.conditional_forecast.controlled_exo_variables
Variable set by the `conditional_forecast` command. Stores the values of the controlled exogenous variables underlying the conditional forecasts to achieve the constrained endogenous variables. Fields are `[number of constrained periods]` by `1` vectors and are of the form:

```
oo_.conditional_forecast.controlled_exo_variables.FORECAST_MOMENT.SHOCK_
↪NAME
```

MATLAB/Octave variable: oo_.conditional_forecast.graphs
Variable set by the `conditional_forecast` command. Stores the information for generating the conditional forecast plots.

Block: conditional_forecast_paths ;
Describes the path of constrained endogenous, before calling `conditional_forecast`. The syntax is similar to deterministic shocks in `shocks`, see `conditional_forecast` for an example.

The syntax of the block is the same as for the deterministic shocks in the `shocks` blocks (see *Shocks on exogenous variables*). Note that you need to specify the full path for all constrained endogenous variables between the first and last specified period. If an intermediate period is not specified, a value of 0 is assumed. That is, if you specify only values for periods 1 and 3, the values for period 2 will be 0. Currently, it is not possible to have uncontrolled intermediate periods.

It is however possible to have different number of controlled periods for different variables. In that case, the order of declaration of endogenous controlled variables and of `controlled_varexo` matters: if the second endogenous variable is controlled for less periods than the first one, the second `controlled_varexo` isn't set for the last 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.

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.

If the model contains strong non-linearities or if some perfectly expected shocks are considered, the forecasts and the conditional forecasts can be computed using an extended path method. The forecast scenario describing the shocks and/or the constrained paths on some endogenous variables should be build. The first step is the forecast scenario initialization using the function `init_plan`:

MATLAB/Octave command: HANDLE = init_plan`(DATES);`
Creates a new forecast scenario for a forecast period (indicated as a dates class, see *dates class members*). This function return a handle on the new forecast scenario.

The forecast scenario can contain some simple shocks on the exogenous variables. This shocks are described using the function `basic_plan`:

MATLAB/Octave command: HANDLE = basic_plan`(HANDLE, 'VAR_NAME', 'SHOCK_TYPE', DATES, MA`
Adds to the forecast scenario a shock on the exogenous variable indicated between quotes in the second argument. The shock type has to be specified in the third argument between quotes: 'surprise' in case of an unexpected shock or 'perfect_foresight' for a perfectly anticipated shock. The fourth argument indicates the period of the shock using a dates class (see *dates class members*). The last argument is the shock path indicated as a MATLAB vector of double. This function return the handle of the updated forecast scenario.

The forecast scenario can also contain a constrained path on an endogenous variable. The values of the related exogenous variable compatible with the constrained path are in this case computed. In other words, a conditional forecast is performed. This kind of shock is described with the function `flip_plan`:

MATLAB/Octave command: **HANDLE = flip_plan**(HANDLE, `VAR_NAME', `VAR_NAME', `SHOCK_TYPE',
Adds to the forecast scenario a constrained path on the endogenous variable specified between quotes in the second argument. The associated exogenous variable provided in the third argument between quotes, is considered as an endogenous variable and its values compatible with the constrained path on the endogenous variable will be computed. The nature of the expectation on the constrained path has to be specified in the fourth argument between quotes: 'surprise' in case of an unexpected path or 'perfect_foresight' for a perfectly anticipated path. The fifth argument indicates the period where the path of the endogenous variable is constrained using a dates class (see *dates class members*). The last argument contains the constrained path as a MATLAB vector of double. This function return the handle of the updated forecast scenario.

Once the forecast scenario if fully described, the forecast is computed with the command `det_cond_forecast`:

MATLAB/Octave command: **DSERIES = det_cond_forecast**(HANDLE[, DSERIES [, DATES]]);
Computes the forecast or the conditional forecast using an extended path method for the given forecast scenario (first argument). The past values of the endogenous and exogenous variables provided with a dseries class (see *dseries class members*) can be indicated in the second argument. By default, the past values of the variables are equal to their steady-state values. The initial date of the forecast can be provided in the third argument. By default, the forecast will start at the first date indicated in the `init_plan command`. This function returns a dset containing the historical and forecast values for the endogenous and exogenous variables.

Example

```
% conditional forecast using extended path method
% with perfect foresight on r path

var y r;
varexo e u;
...
smoothed = dseries('smoothed_variables.csv');

fplan = init_plan(2013Q4:2029Q4);
fplan = flip_plan(fplan, 'y', 'u', 'surprise', 2013Q4:2014Q4,  [1 1.1 1.2␣
↪1.1 ]);
fplan = flip_plan(fplan, 'r', 'e', 'perfect_foresight', 2013Q4:2014Q4, ␣
↪[2 1.9 1.9 1.9 ]);

dset_forecast = det_cond_forecast(fplan, smoothed);

plot(dset_forecast.{'y','u'});
plot(dset_forecast.{'r','e'});
```

Command: **smoother2histval ;**
Command: **smoother2histval**(OPTIONS...);
The purpose of this command is to construct initial conditions (for a subsequent simulation) that are the smoothed values of a previous estimation.

More precisely, after an estimation run with the `smoother` option, `smoother2histval` 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`).

Options

period = INTEGER
Period number to use as the starting point for the subsequent simulation. It should be between 1 and the number of observations that were used to produce the smoothed values. Default: the last observation.

infile = FILENAME
Load the smoothed values from a `_results.mat` file created by a previous Dynare run. Default: use the smoothed values currently in the global workspace.

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 three possible ways of using this command:

- Everything in a single file: run an estimation with a smoother, then run `smoother2histval` (without the `infile` and `outfile` options), then run a stochastic simulation.
- 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.
- In two files: in the first file, run the smoother; in the second file, run `smoother2histval` with the `infile` option equal to the `_results.mat` file created by the first file, and then run a (deterministic or stochastic) simulation.

4.19 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.

4.19.1 Optimal policy under commitment (Ramsey)

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
: Declares or reassigns the discount factor of the central planner `optimal_policy_discount_factor`. Default: `1.0`.

planner_discount_latex_name = LATEX_NAME
: Sets the LaTeX name of the `optimal_policy_discount_factor` parameter.

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: evaluate_planner_objective ;
: This command computes, displays, and stores the value of the planner objective function under Ramsey policy in `oo_.planner_objective_value`, given the initial values of the endogenous state variables. If not specified with `histval`, they are taken to be at their steady state values. The result is a 1 by 2 vector, where the first entry 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*).

In contrast, the second entry stores the value of the planner objective with initial Lagrange multipliers of the planner's problem 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 is the value of implementating optimal policy for the first time and committing not to re-optimize in the future.

Because it entails computing at least a second order approximation, the computation of the planner objective value is skipped with a message when the model is too large (more than 180 state variables, including lagged Lagrange multipliers).

Command: ramsey_policy `[VARIABLE_NAME...];`
Command: ramsey_policy`(OPTIONS...) [VARIABLE_NAME...];`
: This command is formally equivalent to the calling sequence

```
ramsey_model;
stoch_simul(order=1);
evaluate_planner_objective;
```

It computes the first order 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.

This first order approximation to the optimal policy conducted by Dynare is not to be confused with a naive linear quadratic approach to optimal policy that can lead to spurious welfare rankings (see *Kim and Kim (2003)*). In the latter, the optimal policy would be computed subject to the first order approximated FOCs of the private economy. In contrast, Dynare first computes the FOCs of the Ramsey planner's problem subject to the nonlinear constraints that are the FOCs of the private economy and only then approximates these FOCs of planner's problem to first order. Thereby, the second order terms that are required for a second-order correct welfare evaluation are preserved.

Note that the variables in the list after the `ramsey_policy` command can also contain multiplier names. 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.

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.

Note that only a first order approximation of the optimal Ramsey policy is available, leading to a second-order accurate welfare ranking (i.e. `order=1` must be specified).

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 *`histval`*.

Steady state

See *`Ramsey steady state`*.

4.19.2 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 should ensure that your model is linear and your objective is quadratic. Also, you should set the `linear` option of the `model` block.

It is possible to use the `estimation` command after the `discretionary_policy` command, in order to estimate the model with optimal policy under discretion.

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`.

4.19.3 Optimal Simple Rules (OSR)

Command: **osr** `[VARIABLE_NAME...];`
Command: **osr**`(OPTIONS...) [VARIABLE_NAME...];`
This command computes optimal simple policy rules for 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;
- γ 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`- ommand;
- 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` 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` will automatically select `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 subset of the model parameters over which the optimal simple rule is to be optimized, γ, must be listed with `osr_params`.

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: `e-7`.

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.

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;
alpha =  0.48;
sigma = -0.06;

gammarr = 0;
```

(continues on next page)

(continued from previous page)

```
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.

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`.

4.20 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.

4.20.1 Performing sensitivity analysis

Command: dynare_sensitivity ;
Command: dynare_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_τ 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). 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 ρ 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 ρ 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 ρ: 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*.

4.20.2 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 `dynare_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.

Example

```
moment_calibration;
y_obs,y_obs, [0.5, 1.5]; //[unconditional variance]
y_obs,y_obs(-(1:4)), +; //[sign restriction for first year acf with␣
↪logical OR]
@#for ilag in -2:2
y_obs,R_obs(@{ilag}), -; //[-2:2 ccf with logical AND]
@#endfor
@#for ilag in -4:4
y_obs,pie_obs(@{ilag}), -; //[-4_4 ccf with logical AND]
@#endfor
end;
```

4.20.3 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 dynare_sensitivity options). If equal to FILENAME uses the provided path to a specific user defined sample file. Default: 0.

diffuse_filter
Deals with non-stationary cases. See *diffuse_filter*.

Numerical Options

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.

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.

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`.

lik_init = INTEGER
See *lik_init*.

kalman_algo = INTEGER
See *kalman_algo*.

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`.

4.20.4 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.

4.20.4.1 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`.

4.20.4.2 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.

4.20.4.3 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.

4.20.4.4 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 reduce 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 θ 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 θ_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 θ_i;
- <mod_file>_prior_<namendo>_vs_<namexo>_map.mat, <mod_file>_<namendo>_vs_<namlagendo>_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>_threshold.fig: graphical outputs;
- <mod_file>_prior_<namendo>_vs_<namexo>_threshold.mat, <mod_file>_<namendo>_vs_<namlagendo>_threshold.mat: info on the analysis;

4.20.4.5 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 dynare_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.

4.20.4.6 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.

4.20.4.7 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;
dynare_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.

4.21 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, Waggoner and Zha (2008)*.[11] Having done this, you can create forecasts and compute the marginal data density, regime probabilities, IRFs, and variance decomposition of the model.

[11] If you want to align the paper with the description herein, please note that A is A^0 and F is A^+.

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:

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.

[12] An example can be found at https://git.dynare.org/Dynare/dynare/blob/master/tests/ms-dsge/test_ms_dsge.mod.

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://www.dynare.org/DynareWiki/SbvarOptions

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; VARIABLE_NAME [,VARIABLE_NAME...];
EXCLUSION LAG INTEGER; EQUATION INTEGER, VARIABLE_NAME [,VARIABLE_NAME...];
RESTRICTION EQUATION INTEGER, EXPRESSION = EXPRESSION;
```

To be documented. For now, see the wiki: https://archives.dynare.org/DynareWiki/MarkovSwitchingInterface

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 ζ 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 = NAME
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 λ 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,
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 ζ, 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*.

4.22 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;

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;
end;
```

4.23 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')
```

4.24 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. 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 any of the previous types).

4.24.1 Macro expressions

Macro-expressions can be used in two places:

- Inside macro directives, directly;

- In the body of the `.mod` file, between an at-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 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 parenthesis and elements 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, 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]`

4.24.2 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_EXPRESSION
Macro directive: **`@#define`** MACRO_FUNCTION = MACRO_EXPRESSION

Defines a macro-variable or macro function.

Example

```
@#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
```

(continues on next page)

(continued from previous page)

```
@#define t = ("US" in w)           // Equals 1 (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]
@#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, you can zero or more `@#elseif` branches. An `@#elseif` condition is only evaluated if the preceding `@#if` or `@#elseif` condition evaluated to `false`. The `@#else` branch is optional and is 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 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
```

(continues on next page)

(continued from previous page)

```
...
end;
```

This would result in:

```
...
model;
  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`. Even though `linear_mon_pol` contains the value `false` because `@#ifdef` only checks that the variable has been defined, the linear monetary policy is output:

```
@#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;
```

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/parameters declaration, 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);
@#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)
```

4.24.3 Typical usages

4.24.3.1 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 (at the beginning) or changes to the model (during development).

4.24.3.2 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;
```

4.24.3.3 Multi-country models

Here is a skeleton 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;
```

4.24.3.4 Endogeneizing parameters

When calibrating the model, it may be useful to consider a parameter as an endogenous variable (and vice-versa).

For example, suppose production is defined by a CES function:

$$y = \left(\alpha^{1/\xi}\ell^{1-1/\xi} + (1-\alpha)^{1/\xi}k^{1-1/\xi}\right)^{\xi/(\xi-1)}$$

and the labor share in GDP is defined as:

$$\text{lab\_rat} = (w\ell)/(py)$$

In the model, α is a (share) parameter and `lab_rat` is an endogenous variable.

It is clear that calibrating α is not straightforward; on the contrary, we have real world data for `lab_rat` and it is clear that these two variables 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, α will be made an endogenous variable and `lab_rat` will be made a parameter. An economically relevant value will be calibrated for `lab_rat`, and the solution algorithm will deduce the implied value for α.

An implementation could consist of the following files:

`modeqs.mod`

> This file contains variable declarations and model equations. The code for the declaration of α and `lab_rat` would look like:
>
> ```
> @#if steady
>   var alpha;
>   parameter lab_rat;
> @#else
>   parameter alpha;
>   var lab_rat;
> @#endif
> ```

`steady.mod`

This file computes the steady state. It begins with:

```
@#define steady = 1
@#include "modeqs.mod"
```

Then it initializes parameters (including `lab_rat`, excluding α), computes the steady state (using guess values for endogenous, including α), then saves values of parameters and endogenous at steady state in a file, using the `save_params_and_steady_state` command.

`simul.mod`

This file computes the simulation. It begins with:

```
@#define steady = 0
@#include "modeqs.mod"
```

Then it loads values of parameters and endogenous at steady state from file, using the `load_params_and_steady_state` command, and computes the simulations.

4.24.4 MATLAB/Octave loops versus macro processor loops

Suppose you have a model with a parameter ρ 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)
  rho = rhos(i);
  stoch_simul(order=1);
end
```

Here the loop is not unrolled, MATLAB/Octave manages the iterations. This is interesting when there are a lot of iterations.

With a macro processor loop (case 1)

```
rhos = [ 0.8, 0.9, 1];
@#for i in 1:3
  rho = rhos(@{i});
  stoch_simul(order=1);
@#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);
@#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.

4.25 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 `<mod_file>.m` file contains tokens recognized by the Dynare preprocessor.

Example

```
verbatim;
% Anything contained in this block will be passed
% directly to the <modfile>.m file, including comments
var = 1;
end;
```

4.26 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, and morefloats.

CHAPTER 5

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 default location of the configuration file is `$HOME/.dynare`, while on Windows it is `%APPDATA%\dynare.ini` (typically `c:\Users\USERNAME\AppData\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

[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 in 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/R2010b/bin/matlab`).

`BOOLEAN`

: Is `true` or `false`.

5.1 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 the global initialization file to be run at the end of `global_initialization.m`.

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
```

5.2 Parallel Configuration

This section explains how to configure Dynare for parallelizing some tasks which 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 master and slave 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 filesystems. 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 master and of one or more slaves):

For a Windows grid:

- a standard Windows network (SMB) must be in place;

- the PsTools suite must be installed in the path of the master Windows machine;
- the Windows user on the master machine has to be user of any other slave 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 master and on the slave machines;
- SSH keys must be installed so that the SSH connection from the master to the slaves can be done without passwords, or using an SSH agent.

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 parenthesis 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`.

NumberOfThreadsPerJob = INTEGER
For Windows nodes, sets the number of threads assigned to each remote MATLAB/Octave run. The default value is 1.

SingleCompThread = BOOLEAN
Whether or not to disable MATLAB's native multithreading. The default value is `false`. Option meaningless 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
```

5.3 Windows Step-by-Step Guide

This section outlines the steps necessary on most Windows systems to set up Dynare for parallel execution.

1. Write a configuration file containing the options you want. A mimimum 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 somwhere. 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 whitespaces. 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/2016-05-10/matlab
```

CHAPTER 6

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 basic 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.

6.1 Dates

6.1.1 Dates in a mod file

Dynare understands dates in a mod file. Users can declare annual, quarterly, or monthly dates using the following syntax:

```
1990Y
1990Q3
1990M11
```

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 secmond 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 zeros and ones.

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 zeros and ones.

lt operator (less than, <)

Tests if a `dates` object preceeds 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 zeros and ones.

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 zeros and ones.

le operator (less or equal, <=)

Tests if a `dates` object preceeds 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 zeros and ones.

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 zeros and ones.

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. 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 script.

6.1.2 The dates class

Dynare class: dates

arg int freq equal to 1, 4, or 12 (resp. for annual, quarterly, or monthly dates).

arg int ndat the number of declared dates in the object.

arg int time a `ndat*2` array, the years are stored in the first column, the subperiods (1 for annual dates, 1-4 for quarterly dates, and 1-12 for monthly dates) are stored in the second column.

Each member is private, one can display the content of a member but cannot change its value directly. Note 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, 'Q' for quarterly dates, or 'M' for monthly 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), 4 (quarterly), or 12 (monthly). 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.

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'`, `'1990Q1'`, `'1990M2'`), 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.

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)
where `FREQ` is a single character ('Y', 'A', 'Q', 'M') or integer (1, 4, or 12) specifying the frequency, `YEAR` and `SUBPERIOD` are `n*1` vectors of integers. Returns a `dates` object with `n` elements. 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);
```

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: **C = 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
```

(continues on next page)

(continued from previous page)

```
>> 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)
```

(continues on next page)

(continued from previous page)

```
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()

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 logicals. 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 logicals. 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 logicals. 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

   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
```

(continues on next page)

(continued from previous page)

```
  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 logicals. 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 logicals. 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
```

(continues on next page)

(continued from previous page)

```
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 `dseries` object and a scalar integer. Returns `dates` object `A` replicated `B` times.

Example

```
>> d = dates('1950Q1');
>> d*2

ans = <dates: 1950Q1, 1950Q1>
```

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 logicals. 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: **C = pop_**(A)
Method: **C = 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: **C = 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: **B = 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;
>> strings(A)

  ans =

    1x2 cell array

    {'1950Q1'}    {'1950Q2'}
```

Method: **B = subperiod**(A)

Returns the subperiod of a date (an integer scalar between 1 and `A.freq`).

Example

```
>> A = dates('1950Q2');
>> A.subperiod()

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: **B = unique_**(A)

Overloads the MATLAB/Octave `unique` function. Returns a `dates` object with repetitions removed (only the last occurence 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:

arg name A `nobs*1` cell of strings or a `nobs*p` character array, the names of the variables.

arg tex A `nobs*1` cell of strings or a `nobs*p` character array, the tex names of the variables.

arg dates dates An object with `nobs` elements, the dates of the sample.

arg double data A `nobs` by `vobs` array, the data.

arg ops The history of operations on the variables.

arg tags The user-defined tags on the variables.

`data`, `name`, `tex` 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])
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` and `.xls`/`.xlsx` (Octave only supports `.xlsx` files and the io package from Octave-Forge 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'};
TAGS__ = struct()
DATA__ = {}

azert = randn(100,1);
yuiop = randn(100,1);
```

If a `.mat` file is used instead, it should provide the same informations, except that the data should not be given as a set of vectors, but as a single matrix of doubles named `DATA__`. This array should have as many columns as elements in `NAMES__` (the number of variables). Note that 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 in the `.mat` or `.m` file, the initial is by default set equal to `dates('1Y')`. 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 usefull if `INIT__` is not provided in the data file.

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 the MATLAB Table 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. This feature is not available under Octave or MATLAB R2013a or earlier.

Example

Various ways to create a `dseries` object:

```
do1 = dseries(1999Q3);
do2 = dseries('filename.csv');
do3 = dseries([1; 2; 3], 1999Q3, {'var123'}, {'var_{123}'});

>> do1 = dseries(dates('1999Q3'));
>> do2 = dseries('filename.csv');
>> do3 = dseries([1; 2; 3], dates('1999Q3'), {'var123'}, {'var_
↪{123}'});
```

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`.

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
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
```

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

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
```

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
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
```

(continues on next page)

(continued from previous page)

```
>> 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
```

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 informations about the error.

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:

   | 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
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

```
>> 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

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
```

(continues on next page)

(continued from previous page)

```
>> 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: **dentrend_**(A, m)

Detrends `dseries` object `A` with a fitted polynomial of order `m`. Note that each variable is detrended with a different polynomial.

Method: **B = diff**(A)
Method: **diff_**(A)

Returns the first difference of `dseries` object `A`.

Method: **disp**(A)

Overloads the MATLAB/Octave disp function for `dseries` object.

Method: **display**(A)

Overloads the MATLAB/Octave display function for `dseries` object. `display` is the function called by MATLAB to print the content of an object if a semicolon is missing at the end of a MATLAB 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. `dseries` objects `A` and `B` must have the same number of observations (say, T) and variables (N). The returned argument is a $T \times N$ matrix of logicals. Element (i, j) of `C` is equal to `true` if and only if observation i for variable j in `A` and `B` are the same.

Example

```
>> ts0 = dseries(2*ones(3,1));
>> ts1 = dseries([2; 0; 2]);
>> ts0==ts1

ans =

   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();
```

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:
```

(continues on next page)

(continued from previous page)

```
       | 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
```

Method: **f = firstdate**(A)

Returns the first period in `dseries` object `A`.

Method: **f = firstobservedperiod**(A)

Returns the first period where all the variables in `dseries` object `A` are observed (non NaN).

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: **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'});
>> 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: **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 ‘‘ 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`.

Example

```
>> ts0 = dseries(transpose(1:4), '1950Q1')

ts0 is a dseries object:

       | Variable_1
1950Q1 | 1
1950Q2 | 2
1950Q3 | 3
1950Q4 | 4

>> ts1 = ts0.lag()

ts1 is a dseries object:

           | Variable_1
    1950Q1 | NaN
```

(continues on next page)

(continued from previous page)

```
    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)`

Returns the last period in `dseries` object `B`.

Example

```
>> ts = dseries(randn(3,2),'1973Q1');
>> ts.lastdate()

ans = <dates: 1973Q3>
```

Method: **`f = lastobservedperiod`**`(A)`

Returns the last period where all the variables in `dseries` object `A` are observed (non NaN).

Method: **`B = lead`**`(A[, p])`
Method: **`lead_`**`(A[, p])`

Returns lead time series. Default value of integer scalar `p`, the number of leads, is `1`. As in the `lag` method, the `dseries` class overloads the parenthesis so that `ts.lead(p)` is 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 overloading of the parenthesis for `dseries` objects, allows to easily create new `dseries` objects by copying/pasting equations declared in the `model` block. For instance, if an Euler equation is defined in the `model` block:

```
model;
...
1/C - beta/C(1)*(exp(A(1))*K^(alpha-1)+1-delta) ;
...
end;
```

and if variables , ``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)

Computes monthly growth rates of variables in `dseries` object `A`.

Method: **B = mean**(A[, geometric])

Overloads the MATLAB/Octave `mean` function for `dseries` objects. Returns the mean of each variable in `dseries` object `A`. If the second argument is `true` the geometric mean is computed, otherwise (default) the arithmetic mean is reported.

Method: **C = merge**(A, B[, legacy])

Merges two `dseries` objects `A` and `B` in `dseries` object `C`. Objects `A` and `B` need to have common frequency but can be defined on different time ranges. If a variable, say `x`, is defined both in `dseries` objects `A` and `B`, then the `merge` will select the variable `x` as defined in the second input argument, `B`, except for the NaN elements in `B` if corresponding elements in `A` (ie same periods) are well defined numbers. This behaviour can be changed by setting the optional argument `legacy` equal to true, in which case the second variable overwrites the first one even if the second variable has NaNs.

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
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, element by element subtraction. If 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 `1` and $N_A = N_B$, then the `minus` operator will compute for each couple (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 "broadcast" across the larger `dseries` (`A`) so that they have compatible shapes, the `minus` operator will subtract the variable defined in `B` from each variable in `A`. If `B` is a double scalar, then the method `minus` will subtract `B` from all the observations/variables in `A`. If `B` is a row vector of length N_A, then the `minus` method will subtract `B(i)` from all the observations of variable `i`, for $i = 1, ..., N_A$. If `B` is a column vector of length T_A, then 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

>> 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 and computes element-by-element power. `A` is a `dseries` object with `N` variables and `T` observations. If `B` is a real scalar, then `mpower(A,B)` returns a `dseries` object `C` with `C.data(t,n)=A.data(t,n)^C`. If `B` is a `dseries` object with `N` variables and `T` observations then `mpower(A,B)` returns a `dseries` object `C` with `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, element by element division (like the `./` MATLAB/Octave operator). If 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 `1` and $N_A = N_B$, then the `mrdivide` operator will compute for each couple (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 "broadcast" across the larger `dseries` (`A`) so that they have compatible shapes. In this case the `mrdivide` operator will divide each variable defined in A by the variable in B, observation per observation. If B is a double scalar, then `mrdivide` will divide all the observations/variables in `A` by `B`. If `B` is a row vector of length N_A, then `mrdivide` will divide all the observations 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 a division of all the variables by `B`, element by element.

Example

```
>> ts0 = dseries(rand(3,2))

ts0 is a dseries object:

   | 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 and the Hadammard product (the .* MATLAB/Octave operator). If 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 $_B$ 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 `1` and $N_A = N_B$, then the `mtimes` operator will compute for each couple (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 "broadcast" across the larger `dseries` (`A`) so that they have compatible shapes, `mtimes` operator will multiply each variable defined in `A` by the variable in `B`, observation per observation. If `B` is a double scalar, then the method `mtimes` will multiply all the observations/variables in `A` by `B`. If `B` is a row vector of length N_A, then the `mtimes` method will multiply all the observations of variable `i` by `B(i)`, for $i = 1, ..., N_A$. If `B` is a column vector of length T_A, then the `mtimes` method will perform a multiplication of all the variables by `B`, element by element.

Method: **B = nanmean**(A[, geometric])
Overloads the MATLAB/Octave `nanmean` function for `dseries` objects. Returns the mean of each variable in `dseries` object `A` ignoring the NaN values. If the second argument is `true` the geometric mean is computed, otherwise (default) the arithmetic mean is reported.

Method: **C = ne**(A, B)
Overloads the MATLAB/Octave `ne` (not equal, ~=) operator. `dseries` objects `A` and `B` must have the same number of observations (say, T) and variables (N). The returned argument is a T by N matrix of zeros and ones. Element (i, j) of `C` is equal to `1` if and only if observation i for variable j in `A` and `B` are not equal.

Example

```
>> ts0 = dseries(2*ones(3,1));
>> ts1 = dseries([2; 0; 2]);
>> ts0~=ts1

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` `A` object using a one sided HP filter (with a Kalman filter) and returns a `dseries` object, `B`. The default value for `lambda`, the smoothing parameter, is `1600`. By default, if `init` is not provided, the initial value is based on the first two observations.

Method: **B = onesidedhptrend**(A[, lambda[, init]])
Method: **onesidedhptrend_**(A[, lambda[, init]])
Extracts the trend component from a `dseries` `A` object using a one sided HP filter (with a Kalman filter) and returns a `dseries` object, `B`. The default value for `lambda`, the smoothing parameter, is `1600`. By default, if `init` is not provided, the initial value is based on the first two observations.

Method: **h = plot**(A)

Method: `h = plot(A, B)`
Method: `h = plot(A[, ...])`
Method: `h = plot(A, B[, ...])`

Overloads MATLAB/Octave's `plot` function for `dseries` objects. Returns a MATLAB/Octave plot handle, that can be used to modify the properties of the plotted time series. If only one `dseries` object, `A`, is passed as argument, then the plot function will put the associated dates on the x-abscissa. If this `dseries` object contains only one variable, additional arguments can be passed to modify the properties of the plot (as one would do with the MATLAB/Octave's version of the plot function). If `dseries` object `A` contains more than one variable, it is not possible to pass these additional arguments and the properties of the plotted time series must be modified using the returned plot handle and the MATLAB/Octave `set` function (see example below). If two `dseries` objects, `A` and `B`, are passed as input arguments, the plot function will plot the variables in `A` against the variables in `B` (the number of variables in each object must be the same otherwise an error is issued). Again, if each object contains only one variable, additional arguments can be passed to modify the properties of the plotted time series, otherwise the MATLAB/Octave `set` command has to be used.

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'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, element by element addition. If 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 1 and $N_A = N_B$, then the `plus` operator will compute for each couple (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 "broadcast" across the larger `dseries` (`A`) so that they have 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 $i = 1, ..., N_A$. 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 variable B from dseries object A. By default, if the second argument is not provided, 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 = qdiff**(A)
Method: **B = qgrowth**(A)
Method: **qdiff_**(A)
Method: **qgrowth_**(A)

Computes quarterly differences or growth rates.

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()

ts1 is a dseries object:

        | Variable_1
1950M1  | NaN
1950M2  | NaN
1950M3  | NaN
1950M4  | 3
1950M5  | 3
1950M6  | 3
```

Method: **C = remove**(A, B)
Method: **remove_**(A, B)

Alias for the pop method with two arguments. Removes variable B from dseries object A.

Example

```
>> ts0 = dseries(ones(3,3));
>> ts1 = ts0.remove('Variable_2');

ts1 is a dseries object:

   | Variable_1 | Variable_3
1Y | 1          | 1
```

(continues on next page)

(continued from previous page)

```
2Y | 1          | 1
3Y | 1          | 1
```

A shorter syntax is available: `remove(ts,'Variable_2')` is equivalent to `ts{'Variable_2'} = []` (`[]` can be replaced by any empty object). This alternative syntax is useful if more than one variable has to be removed. For instance:

```
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)

Rename variable `oldname` to `newname` in `dseries` object `A`. Returns a `dseries` object. If more than one variable needs to be renamed, it is possible to pass cells of char arrays as 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 passed in the cell string array `newname`. `newname` must have the same number of elements as `dseries` object `A` has variables. Returns a `dseries` object.

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: **save**(A, basename[, format])

Overloads the MATLAB/Octave `save` function and saves `dseries` object `A` to disk. Possible formats are `mat` (this is the default), `m` (MATLAB/Octave script), and `csv` (MATLAB binary data file). The name of the file without 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 variables in `dseries` object `A` and returns a `dseries` object `B` with new names `s1`, `s2`, ... The number of input arguments after the first one (`dseries` object `A`) must be equal to `A.vobs` (the number of variables in `A`). `s1` will be the name of the first variable in `B`, `s2` the name of the second variable in `B`, and so on.

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's `size` function. Returns the number of observations in `dseries` object `A` (i.e. `A.nobs`) and the number of variables (i.e. `A.vobs`). If a second input argument is passed, the `size` function returns the number of observations if `dim=1` or the number of variables if `dim=2` (for all other values of `dim` an error is issued).

Example

```
>> ts0 = dseries(ones(1,3));
>> ts0.size()

ans =

     1     3
```

Method: **`B = std`**`(A[, geometric])`

Overloads the MATLAB/Octave `std` function for `dseries` objects. Returns the standard deviation of each variable in `dseries` object `A`. If the second argument is `true` the geometric standard deviation is computed (default value of the second argument is `false`).

Method: **`A = tag`**`(A, a[, b, c])`

Add 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)

Redefines the tex name of variable `name` to `newtexname` in `dseries` object `A`. Returns a `dseries` object.

With only two arguments `A` and `newtexname`, it redefines the tex names of the `A` to those contained in `newtexname`. Here, `newtexname` is a cell string array with the same number of entries as variables in `A`.

Method: **B = uminus**(A)

Overloads `uminus` (−, unary minus) for `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` MATLAB/Octave method for `dseries` objects. This method is used to append more observations to a `dseries` object. Returns a `dseries` object `D` containing the variables in `dseries` objects passed as inputs. All the input arguments must be `dseries` objects with the same variables defined on 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 number of variables in `dseries` object `A`.

Example

```
>> ts0 = dseries(randn(10,2));
>> ts0.vobs

ans =

    2
```

Method: **B = ydiff**(A)
Method: **B = ygrowth**(A)
Method: **ydiff_**(A)
Method: **ygrowth_**(A)

Computes yearly differences or growth rates.

CHAPTER 7

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 or not 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. Alows 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 or not to display the major grid on the graph. Default: `true`.

showLegend, BOOLEAN
Whether or not 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 or not 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 or not there is a common scaling factor for the y axis. Default: `true`.

yTickLabelZeroFill, BOOLEAN
Whether or not 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 or not 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 or not to show vertical lines separating the columns. Default: `false`.

writeCSV, BOOLEAN
Whether or not 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 or not to display this series as a bar graph as oppsed 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. Usefull 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 or not 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');
```

(continues on next page)

(continued from previous page)

```
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, ...
                'tableShowMarkers', true, 'tableAlignRight', true);
end

%% Write & Compile Report
rep.write();
rep.compile();
```

Once compiled, the report looks like:

My Page Title

My Page Subtitle

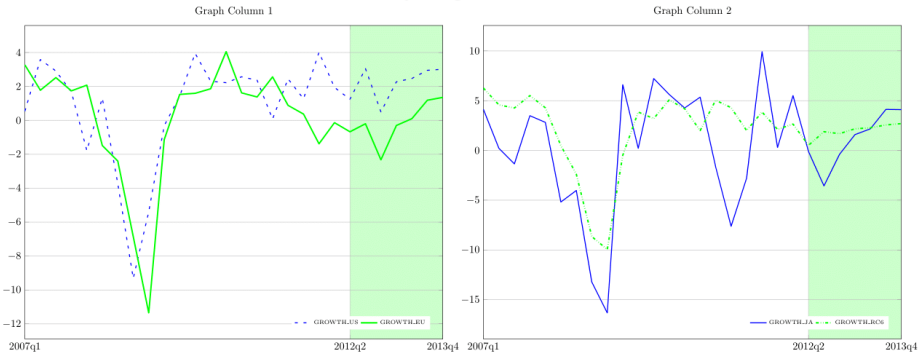

Table 1

	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 8

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. You should have these files in the `examples` subdirectory of your distribution.

Here is a short list of the examples included. For a more complete description, please refer to the comments inside the files themselves.

`ramst.mod`

An elementary real business cycle (RBC) model, simulated in a deterministic setup.

`example1.mod` `example2.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).

`example3.mod`

A small RBC model in a stochastic setup, presented in *Collard (2001)*. The steady state is solved analytically using the `steady_state_model` block (see *steady_state_model*).

`fs2000.mod`

A cash in advance model, estimated by *Schorfheide (2000)*. The file shows how to use Dynare for estimation.

`fs2000_nonstationary.mod`

The same model than `fs2000.mod`, but written in non-stationary form. Detrending of the equations is done by Dynare.

`bkk.mod`

Multi-country RBC model with time to build, presented in *Backus, Kehoe and Kydland (1992)*. The file shows how to use Dynare's macro processor.

`agtrend.mod`

Small open economy RBC model with shocks to the growth trend, presented in *Aguiar and Gopinath (2004)*.

`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.

`Ramsey_Example.mod`

File demonstrating how to conduct optimal policy experiments in a simple New Keynesian model either under commitment (Ramsey) or using optimal simple rules (OSR)

CHAPTER 9

Dynare misc commands

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_,bayestopt_,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*.

Command: **generate_trace_plots**(CHAIN_NUMBER);

Generates trace plots of the MCMC draws for all estimated parameters and the posterior density in the specified Markov Chain CHAIN_NUMBER.

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 informations 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
```

`--info`

Prints on screen the internal documentation of ROUTINENAME (if this routine exists and if this routine has a texinfo internal documentation header). The path to `ROUTINENAME` has to be provided, if the routine is not in the current directory.

Example

```
>> internals --doc ../matlab/fr/ROUTINENAME
```

At this time, will work properly for only a small number of routines. At the top of the (available) MATLAB/Octave routines a commented block for the internal documentation is written in the GNU texinfo documentation format. This block is processed by calling texinfo from MATLAB. Consequently, texinfo has to be installed on your machine.

`--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 informations 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.

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 maximising 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.

CHAPTER 10

Bibliography

- Abramowitz, Milton and Irene A. Stegun (1964): "Handbook of Mathematical Functions", Courier Dover Publications.
- Adjemian, Stéphane, Matthieu Darracq Parriès and Stéphane Moyen (2008): "Towards a monetary policy evaluation framework", *European Central Bank Working Paper*, 942.
- Aguiar, Mark and Gopinath, Gita (2004): "Emerging Market Business Cycles: The Cycle is the Trend," *NBER* Working Paper, 10734.
- Amisano, Gianni and Tristani, Oreste (2010): "Euro area inflation persistence in an estimated nonlinear DSGE model", *Journal of Economic Dynamics and Control*, 34(10), 1837–1858.
- Andreasen, Martin M., Jesús Fernández-Villaverde, and Juan Rubio-Ramírez (2018): "The Pruned State-Space System for Non-Linear DSGE Models: Theory and Empirical Applications," *Review of Economic Studies*, 85(1), pp. 1-49.
- Andrews, Donald W.K (1991): "Heteroskedasticity and autocorrelation consistent covariance matrix estimation", *Econometrica*, 59(3), 817–858.
- Backus, David K., Patrick J. Kehoe, and Finn E. Kydland (1992): "International Real Business Cycles," *Journal of Political Economy*, 100(4), 745–775.
- Baxter, Marianne 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.
- Boucekkine, Raouf (1995): "An alternative methodology for solving nonlinear forward-looking models," *Journal of Economic Dynamics and Control*, 19, 711–734.
- Brooks, Stephen P., and Andrew Gelman (1998): "General methods for monitoring convergence of iterative simulations," *Journal of Computational and Graphical Statistics*, 7, pp. 434–455.
- Cardoso, Margarida F., R. 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.
- Chib, Siddhartha and Srikanth Ramamurthy (2010): "Tailored randomized block MCMC methods with application to DSGE models," *Journal of Econometrics*, 155, 19–38.
- Christiano, Lawrence J., 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.
- Christoffel, Kai, Günter Coenen and Anders Warne (2010): "Forecasting with DSGE models," *ECB Working Paper Series*, 1185.

- Collard, Fabrice (2001): "Stochastic simulations with Dynare: A practical guide".
- Collard, Fabrice and Michel Juillard (2001a): "Accuracy of stochastic perturbation methods: The case of asset pricing models," *Journal of Economic Dynamics and Control*, 25, 979–999.
- Collard, Fabrice and Michel Juillard (2001b): "A Higher-Order Taylor Expansion Approach to Simulation of Stochastic Forward-Looking Models with an Application to a Non-Linear Phillips Curve," *Computational Economics*, 17, 125–139.
- Corona, Angelo, M. Marchesi, Claudio Martini, and Sandro Ridella (1987): "Minimizing multimodal functions of continuous variables with the "simulated annealing" algorithm", *ACM Transactions on Mathematical Software*, 13(3), 262–280.
- Del Negro, Marco and Franck Schorfheide (2004): "Priors from General Equilibrium Models for VARs", *International Economic Review*, 45(2), 643–673.
- Dennis, Richard (2007): "Optimal Policy In Rational Expectations Models: New Solution Algorithms", *Macroeconomic Dynamics*, 11(1), 31–55.
- Durbin, J. and S. J. Koopman (2012), *Time Series Analysis by State Space Methods*, Second Revised Edition, Oxford University Press.
- Fair, Ray and John Taylor (1983): "Solution and Maximum Likelihood Estimation of Dynamic Nonlinear Rational Expectation Models," *Econometrica*, 51, 1169–1185.
- Fernández-Villaverde, Jesús and Juan Rubio-Ramírez (2004): "Comparing Dynamic Equilibrium Economies to Data: A Bayesian Approach," *Journal of Econometrics*, 123, 153–187.
- Fernández-Villaverde, Jesús and Juan Rubio-Ramírez (2005): "Estimating Dynamic Equilibrium Economies: Linear versus Nonlinear Likelihood," *Journal of Applied Econometrics*, 20, 891–910.
- Fernández-Villaverde, Jesús (2010): "The econometrics of DSGE models," *SERIEs*, 1, 3–49.
- Ferris, Michael C. and Todd S. Munson (1999): "Interfaces to PATH 3.0: Design, Implementation and Usage", *Computational Optimization and Applications*, 12(1), 207–227.
- Geweke, John (1992): "Evaluating the accuracy of sampling-based approaches to the calculation of posterior moments," in J.O. Berger, J.M. Bernardo, A.P. Dawid, and A.F.M. Smith (eds.) Proceedings of the Fourth Valencia International Meeting on Bayesian Statistics, pp. 169–194, Oxford University Press.
- Geweke, John (1999): "Using simulation methods for Bayesian econometric models: Inference, development and communication," *Econometric Reviews*, 18(1), 1–73.
- Giordani, Paolo, Michael Pitt, and Robert Kohn (2011): "Bayesian Inference for Time Series State Space Models" in: *The Oxford Handbook of Bayesian Econometrics*, ed. by John Geweke, Gary Koop, and Herman van Dijk, Oxford University Press, 61–124.
- Goffe, William L., Gary D. Ferrier, and John Rogers (1994): "Global Optimization of Statistical Functions with Simulated Annealing," *Journal of Econometrics*, 60(1/2), 65–100.
- Hansen, Nikolaus and Stefan Kern (2004): "Evaluating the CMA Evolution Strategy on Multimodal Test Functions". In: *Eighth International Conference on Parallel Problem Solving from Nature PPSN VIII*, Proceedings, Berlin: Springer, 282–291.
- Harvey, Andrew C. and Garry D.A. Phillips (1979): "Maximum likelihood estimation of regression models with autoregressive-moving average disturbances," *Biometrika*, 66(1), 49–58.
- Herbst, Edward (2015): "Using the "Chandrasekhar Recursions" for Likelihood Evaluation of DSGE Models," *Computational Economics*, 45(4), 693–705.
- Ireland, Peter (2004): "A Method for Taking Models to the Data," *Journal of Economic Dynamics and Control*, 28, 1205–26.
- Iskrev, Nikolay (2010): "Local identification in DSGE models," *Journal of Monetary Economics*, 57(2), 189–202.
- Judd, Kenneth (1996): "Approximation, Perturbation, and Projection Methods in Economic Analysis", in *Handbook of Computational Economics*, ed. by Hans Amman, David Kendrick, and John Rust, North Holland Press, 511–585.

- Juillard, Michel (1996): "Dynare: A program for the resolution and simulation of dynamic models with forward variables through the use of a relaxation algorithm," CEPREMAP, *Couverture Orange*, 9602.
- Kim, Jinill and Sunghyun Kim (2003): "Spurious welfare reversals in international business cycle models," *Journal of International Economics*, 60, 471–500.
- Kanzow, Christian and Stefania Petra (2004): "On a semismooth least squares formulation of complementarity problems with gap reduction," *Optimization Methods and Software*, 19, 507–525.
- Kim, Jinill, 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(11), 3397–3414.
- Komunjer, Ivana and Ng, Serena (2011): "Dynamic identification of dynamic stochastic general equilibrium models", *Econometrica*, 79, 1995–2032.
- Koop, Gary (2003), *Bayesian Econometrics*, John Wiley & Sons.
- Koopman, S. J. and J. Durbin (2000): "Fast Filtering and Smoothing for Multivariate State Space Models," *Journal of Time Series Analysis*, 21(3), 281–296.
- Koopman, S. J. and J. Durbin (2003): "Filtering and Smoothing of State Vector for Diffuse State Space Models," *Journal of Time Series Analysis*, 24(1), 85–98.
- Kuntsevich, Alexei V. and Franz Kappel (1997): "SolvOpt - The solver for local nonlinear optimization problems (version 1.1, Matlab, C, FORTRAN)", University of Graz, Graz, Austria.
- Laffargue, Jean-Pierre (1990): "Résolution d'un modèle macroéconomique avec anticipations rationnelles", *Annales d'Économie et Statistique*, 17, 97–119.
- Liu, Jane and Mike West (2001): "Combined parameter and state estimation in simulation-based filtering", in *Sequential Monte Carlo Methods in Practice*, Eds. Doucet, Freitas and Gordon, Springer Verlag.
- Lubik, Thomas and Frank Schorfheide (2007): "Do Central Banks Respond to Exchange Rate Movements? A Structural Investigation," *Journal of Monetary Economics*, 54(4), 1069–1087.
- Murray, Lawrence M., 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, 494–521.
- Mutschler, Willi (2015): "Identification of DSGE models - The effect of higher-order approximation and pruning", *Journal of Economic Dynamics & Control*, 56, 34-54.
- Pearlman, Joseph, David Currie, and Paul Levine (1986): "Rational expectations models with partial information," *Economic Modelling*, 3(2), 90–105.
- Planas, Christophe, Marco Ratto and Alessandro Rossi (2015): "Slice sampling in Bayesian estimation of DSGE models".
- Pfeifer, Johannes (2013): "A Guide to Specifying Observation Equations for the Estimation of DSGE Models".
- Pfeifer, Johannes (2014): "An Introduction to Graphs in Dynare".
- Qu, Zhongjun and Tkachenko, Denis (2012): "Identification and frequency domain quasi-maximum likelihood estimation of linearized dynamic stochastic general equilibrium models", *Quantitative Economics*, 3, 95–132.
- Rabanal, Pau and Juan Rubio-Ramirez (2003): "Comparing New Keynesian Models of the Business Cycle: A Bayesian Approach," Federal Reserve of Atlanta, *Working Paper Series*, 2003-30.
- Raftery, Adrian E. and Steven Lewis (1992): "How many iterations in the Gibbs sampler?," in *Bayesian Statistics, Vol. 4*, ed. J.O. Berger, J.M. Bernardo, A.P. * Dawid, and A.F.M. Smith, Clarendon Press: Oxford, pp. 763-773.
- Ratto, Marco (2008): "Analysing DSGE models with global sensitivity analysis", *Computational Economics*, 31, 115–139.

- Ratto, Marco and Iskrev, Nikolay (2011): "Identification Analysis of DSGE Models with DYNARE.", *MONFISPOL* 225149.
- Schorfheide, Frank (2000): "Loss Function-based evaluation of DSGE models," *Journal of Applied Econometrics*, 15(6), 645–670.
- Schmitt-Grohé, Stephanie 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.
- Schnabel, Robert B. and Elizabeth Eskow (1990): "A new modified Cholesky algorithm," *SIAM Journal of Scientific and Statistical Computing*, 11, 1136–1158.
- Sims, Christopher A., Daniel F. Waggoner and Tao Zha (2008): "Methods for inference in large multiple-equation Markov-switching models," *Journal of Econometrics*, 146, 255–274.
- Skoeld, Martin and Gareth O. Roberts (2003): "Density Estimation for the Metropolis-Hastings Algorithm," *Scandinavian Journal of Statistics*, 30, 699–718.
- Smets, Frank 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.
- Stock, James H. and Mark W. Watson (1999). "Forecasting Inflation,", *Journal of Monetary Economics*, 44(2), 293–335.
- Uhlig, Harald (2001): "A Toolkit for Analysing Nonlinear Dynamic Stochastic Models Easily," in *Computational Methods for the Study of Dynamic Economies*, Eds. Ramon Marimon and Andrew Scott, Oxford University Press, 30–61.
- Villemot, Sébastien (2011): "Solving rational expectations models at first order: what Dynare does," *Dynare Working Papers*, 2, CEPREMAP.

Index

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

Y