Dynare Working Papers Series
http://www.dynare.org/wp/

Dynare: Reference Manual
Version 4

Stéphane Adjemian
Houtan Bastani
Michel Juillard
Ferhat Mihoubi
George Perendia
Marco Ratto
Sébastien Villemot

Working Paper no. 1

Initial revision: April 2011
This revision: March 2012

CEPREMAP
CENTRE POUR LA RECHERCHE ECONOMIQUE ET SES APPLICATIONS
142, rue du Chevaleret — 75013 Paris — France
http://www.cepremap.ens.fr

Dynare

Reference Manual, version 4.2.5

Stéphane Adjemian
Houtan Bastani
Michel Juillard
Ferhat Mihoubi
George Perendia
Marco Ratto
Sébastien Villemot

Table of Contents

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, Mac and Linux platforms and is fully documented through a user guide and 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. Currently the development team of Dynare is composed of Stéphane Adjemian (Université du Maine, Gains and Cepremap), Houtan Bastani (Cepremap), Michel Juillard (Banque de France), Frédéric Karamé (Université d'Évry, Epee and Cepremap), Junior Maih (Norges Bank), Ferhat Mihoubi (Université d'Évry, Epee and Cepremap), George Perendia, Marco Ratto (JRC) and Sébastien Villemot (Cepremap and Paris School of Economics). 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). The Dynare project also received funding through the Seventh Framework Programme for Research (FP7) of the European Commission's Socio-economic Sciences and Humanities (SSH) Program from October 2008 to September 2011 under grant agreement SSH-CT-2009-225149.

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. 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.

New users should rather begin with Dynare User Guide (*Mancini (2007)*), distributed with Dynare and also available from the official Dynare web site.

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

1.3 Citing Dynare in your research

If you would like to refer to Dynare in a research article, the recommended way is to cite the present manual, as follows:

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

Note that citing the Dynare Reference Manual in your research is a good way to help the Dynare project.

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

2 Installation and configuration

2.1 Software requirements

Packaged versions of Dynare are available for Windows XP/Vista/Seven, Debian GNU/Linux, Ubuntu and Mac OS X Leopard/Snow Leopard. 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 7.0 (R14) or above;
- GNU Octave version 3.0.0 or above.

Some installation instructions for GNU Octave can be found on the Dynare Wiki.

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;
- If under GNU Octave, the following Octave-Forge package: optim.

If you plan to use the `use_dll` option of the `model` command, you will need to install the necessary requirements for compiling MEX files on your machine. If you are using MATLAB under Windows, install a C++ compiler on your machine and configure it with MATLAB: see instructions on the Dynare wiki. Users of Octave under Linux should install the package for MEX file compilation (under Debian or Ubuntu, it is called '`octave3.2-headers`' or '`octave3.0-headers`'). If you are using Octave or MATLAB under Mac OS X, you should install the latest version of XCode: see instructions on the Dynare wiki. Mac OS X Octave users will also need to install gnuplot if they want graphing capabilities. Users of MATLAB under Linux and Mac OS X, and users of Octave under Windows, normally need to do nothing, since a working compilation environment is available by default.

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 Section 2.3.3 [Some words of warning], page 5).

2.2.2 On Debian GNU/Linux and Ubuntu

Please refer to the Dynare Wiki for detailed instructions.

Dynare will be installed under '`/usr/share/dynare`' and '`/usr/lib/dynare`'. Documentation will be under '`/usr/share/doc/dynare`'.

2.2.3 On Mac OS X

Execute the automated installer called '`dynare-4.x.y-macosx-10.5+10.6.pkg`' (where 4.*x.y* is the version number), and follow the instructions. The default installation directory is '`/Applications/Dynare/4.x.y`'.

Please refer to the Dynare Wiki for detailed instructions.

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

Note that you can have several versions of Dynare coexisting (for example in '`/Applications/Dynare`'), as long as you correctly adjust your path settings (see Section 2.3.3 [Some words of warning], page 5).

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 Dynare Wiki.

2.3 Configuration

2.3.1 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 Debian GNU/Linux or Ubuntu, type:

  ```
  addpath /usr/share/dynare/matlab
  ```

  Under Mac OS X, 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.

2.3.2 For GNU 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:

```
addpath c:\dynare\4.x.y\matlab
```

Under Debian GNU/Linux or Ubuntu, there is no need to use the `addpath` command; the packaging does it for you.

Under Mac OS X, 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
```

If you are using an Octave version strictly older than 3.2.0, you will also want to tell to Octave to accept the short syntax (without parentheses and quotes) for the `dynare` command, by typing:

```
mark_as_command dynare
```

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 by '`c:\Documents and Settings\USERNAME\`'). This file is run by Octave at every startup.

2.3.3 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 to your path can potentially create problems, if some of your M-files have the same names than Dynare files. Your files would then override Dynare files, and make Dynare unusable.

3 Dynare invocation

In order to give instructions to Dynare, the user has to write a *model file* whose filename extension must be '`.mod`'. This file contains the description of the model and the computing tasks required by the user. Its contents is described in Chapter 4 [The Model file], page 9.

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 GNU Octave instructions are generated; in the second step, the program actually runs the computations. Boths steps are triggered automatically by the `dynare` command.

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

Description

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 Chapter 4 [The Model file], page 9.

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

'*FILENAME*`.m`'
: Contains variable declarations, and computing tasks

'*FILENAME*`_dynamic.m`'
: Contains the dynamic model equations

'*FILENAME*`_static.m`'
: Contains the long run static model equations

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

`dynare` will then run the computing tasks by executing '*FILENAME*`.m`'.

Options

`noclearall`
: By default, `dynare` will issue a `clear all` command to MATLAB or Octave, thereby deleting all workspace variables; this options instructs `dynare` not to clear 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 Section 4.19 [Macro-processing language], page 54); the saved output will go in the file specified, or if no file is specified in '*FILENAME*`-macroexp.mod`'

`onlymacro`
: Instructs the preprocessor to only perform the macro-processing step, and stop just after. Mainly useful for debugging purposes or for using the macro-processor independently of the rest of Dynare toolbox.

`nolinemacro`
: Instructs the macro-preprocessor to omit line numbering information in the intermediary '`.mod`' file created after the maco-processing step. Useful in conjunction with `savemacro` when one wants that to reuse the intermediary '`.mod`' file, without having it cluttered by line numbering directives.

`warn_uninit`
: Display a warning for each variable or parameter which is not initialized. See Section 4.4 [Parameter initialization], page 15, or [load_params_and_steady_state], page 60 for initialization of parameters. See Section 4.7 [Initial and terminal conditions], page 19, or [load_params_and_steady_state], page 60 for initialization of endogenous and exogenous variables.

`console`
: Activate console mode: Dynare will not use graphical waitbars for long computations. Note that this option is only useful under MATLAB, since Octave does not provide graphical waitbar capabilities.

`cygwin`
: Tells Dynare that your MATLAB is configured for compiling MEX files with Cygwin (see Section 2.1 [Software requirements], page 3). This option is only available under Windows, and is used in conjunction with `use_dll`.

`msvc`
: Tells Dynare that your MATLAB is configured for compiling MEX files with Microsoft Visual C++ (see Section 2.1 [Software requirements], page 3). This option is only available under Windows, and is used in conjunction with `use_dll`.

`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 Chapter 5 [The Configuration File], page 61, for more information about the configuration file.

`conffile=`*FILENAME*
: Specifies the location of the configuration file if it differs from the default. See Chapter 5 [The Configuration File], page 61, for more information about the configuration file and its default location.

`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 Chapter 5 [The Configuration File], page 61, for more information about the configuration file.

Output

Depending on the computing tasks requested in the '`.mod`' file, executing command `dynare` will leave in the workspace variables containing results available for further processing. More details are given under the relevant computing tasks.

The `M_`, `oo_` and `options_` structures are also saved in a file called '*FILENAME*`_results.mat`'.

Example

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

The output of Dynare is left into three main variables in the MATLAB/Octave workspace:

`M_` [MATLAB/Octave variable]
Structure containing various informations about the model.

`options_` [MATLAB/Octave variable]
Structure contains the values of the various options used by Dynare during the computation.

`oo_` [MATLAB/Octave variable]
Structure containing the various results of the computations.

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;`.

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: '`[]`';
- repreated arguments are indicated by ellipses: "...";
- mutually exclusive arguments are separated by vertical bars: '`|`';
- *INTEGER* indicates an integer number;
- *DOUBLE* indicates a double precision number. The following syntaxes are valid: `1.1e3`, `1.1E3`, `1.1d3`, `1.1D3`;
- *EXPRESSION* indicates a mathematical expression valid outside the model description (see Section 4.3 [Expressions], page 12);
- *MODEL_EXPRESSION* indicates a mathematical expression valid in the model description (see Section 4.3 [Expressions], page 12 and Section 4.5 [Model declaration], page 16);
- *MACRO_EXPRESSION* designates an expression of the macro-processor (see Section 4.19.1 [Macro expressions], page 54);
- *VARIABLE_NAME* indicates a variable name starting with an alphabetical character and can't contain: '`()+-*/^=!;:@#.`' or accentuated characters;
- *PARAMETER_NAME* indicates a parameter name starting with an alphabetical character and can't contain: '`()+-*/^=!;:@#.`' or accentuated characters;
- *LATEX_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

Declarations of variables and parameters are made with the following commands:

`var` *VARIABLE_NAME* [*$LATEX_NAME$*]...; [Command]
`var` (*deflator* = *MODEL_EXPRESSION*) *VARIABLE_NAME* [*$LATEX_NAME$*]...; [Command]

Description

This required command declares the endogenous variables in the model. See Section 4.1 [Conventions], page 9, for the syntax of *VARIABLE_NAME* and *MODEL_EXPRESSION*. Optionally it is possible to give a LaTeX name to the variable or, if it is nonstationary, provide information regarding its deflator.

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

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_EXPRESSION*

The expression used to detrend an endogenous variable. All trend variables, endogenous variables and parameters referenced in *MODEL_EXPRESSION* must already have been declared by the `trend_var`, `var` and `parameters` commands.

Example

```
var c gnp q1 q2;
var(deflator=A) i b;
```

`varexo` *VARIABLE_NAME* [*$LATEX_NAME$*]...; [Command]

Description

This optional command declares the exogenous variables in the model. See Section 4.1 [Conventions], page 9, for the syntax of *VARIABLE_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.

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

Example

```
varexo m gov;
```

`varexo_det` *VARIABLE_NAME* [*$LATEX_NAME$*]...; [Command]

Description

This optional command declares exogenous deterministic variables in a stochastic model. See Section 4.1 [Conventions], page 9, for the syntax of *VARIABLE_NAME*. Optionally it is possible to give a LaTeX name to the variable.

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.

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

Example

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

`parameters` *PARAMETER_NAME* [*$LATEX_NAME$*]...; [Command]

Description

This command declares parameters used in the model, in variable initialization or in shocks declarations. See Section 4.1 [Conventions], page 9, for the syntax of *PARAMETER_NAME*. Optionally it is possible to give a LaTeX name to the parameter.

The parameters must subsequently be assigned values (see Section 4.4 [Parameter initialization], page 15).

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

Example

```
parameters alpha, bet;
```

`change_type` (*var* | *varexo* | *varexo_det* | *parameters*) *VARIABLE_NAME* | *PARAMETER_NAME*...; [Command]

Description

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

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

`predetermined_variables` *VARIABLE_NAME*...; [Command]

Description

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.

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;
```

`trend_var` (*growth_factor* = *MODEL_EXPRESSION*) *VARIABLE_NAME* [*$LATEX_NAME$*]...; [Command]

Description

This optional command declares the trend variables in the model. See Section 4.1 [Conventions], page 9, for the syntax of *MODEL_EXPRESSION* and *VARIABLE_NAME*. Optionally it is possible to give a LaTeX name to the variable.

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_EXPRESSION* must already have been declared by the `var` and `parameters` commands.

Example

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

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[1].

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:

`inf` [Constant]

Represents infinity.

`nan` [Constant]

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

[1] 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 Section 4.4 [Parameter initialization], page 15) 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. Please refer to *Mancini-Griffoli (2007)* for more details and concrete examples.

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 Section 4.5 [Model declaration], page 16).

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 Section 4.4 [Parameter initialization], page 15); 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`): `<`, `>`, `<=`, `>=`, `==`, `!=`

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

`STEADY_STATE` (*MODEL_EXPRESSION*) [Operator]
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.

`EXPECTATION` (*INTEGER*) (*MODEL_EXPRESSION*) [Operator]
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 Section 4.6 [Auxiliary variables], page 18, 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*:

`exp` (*x*) [Function]
Natural exponential.

`log` (*x*) [Function]
`ln` (*x*) [Function]
Natural logarithm.

`log10` (*x*) [Function]
Base 10 logarithm.

`sqrt` (*x*) [Function]
Square root.

`sin` (*x*) [Function]
`cos` (*x*) [Function]
`tan` (*x*) [Function]
`asin` (*x*) [Function]
`acos` (*x*) [Function]
`atan` (*x*) [Function]
Trigonometric functions.

`max` (*a*, *b*) [Function]
`min` (*a*, *b*) [Function]
Maximum and minimum of two reals.

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

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

`erf` (*x*) [Function]
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 necessary for external functions used in an *EXPRESSION*.

`external_function` (*OPTIONS...*); [Command]

Description

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);
```

4.4 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, bet;

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

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

`M_.params` [MATLAB/Octave variable]
Contains the values of model parameters. The parameters are in the order that was used in the `parameters` command.

4.5 Model declaration

The model is declared inside a `model` block:

`model` *;* [Block]
`model` (*OPTIONS...*); [Block]

Description

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_policy`.

The syntax of equations must follow the conventions for *MODEL_EXPRESSION* as described in Section 4.3 [Expressions], page 12. 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 Section 4.2 [Variable declarations], page 9), 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. A model local variable declaration looks like:

```
# VARIABLE_NAME = MODEL_EXPRESSION;
```

Options

`linear` Declares the model as being linear. It spares oneself from having to declare initial values for computing the steady state, and it sets automatically `order=1` in `stoch_simul`.

`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 Section 2.1 [Software requirements], page 3 for more details). Using this option can result in faster simulations or estimations, at the expense of some initial compilation time.[2]

`block` Perform the block decomposition of the model, and exploit it in computations. See Dynare wiki for details on the algorithm.

`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:

[2] 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.

`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.

Example 1: 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 2: 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 3: 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 list of model equations to a LaTeX file, using the `write_latex_dynamic_model` command. The static model can also be written with the `write_latex_static_model` command.

`write_latex_dynamic_model ;` [Command]

Description

This command creates a LaTeX file containing the (dynamic) model.

If your '`.mod`' file is '*`FILENAME`*`.mod`', then Dynare will create a file called '*`FILENAME`*`_dynamic.tex`', containing the list of all the dynamic model equations.

If LaTeX names were given for variables and parameters (see Section 4.2 [Variable declarations], page 9), 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], page 11) 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 (see [expectation], page 13) will have been removed, replaced by auxiliary variables and new equations as explained in the documentation of the operator,
- 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.

`write_latex_static_model ;` [Command]

Description

This command creates a LaTeX file containing the static model.

If your '`.mod`' file is '*`FILENAME`*`.mod`', then Dynare will create a file called '*`FILENAME`*`_static.tex`', containing the list of all the equations of the steady state model.

If LaTeX names were given for variables and parameters (see Section 4.2 [Variable declarations], page 9), 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], page 17 for details).

Also note that this command will not output the contents of the optional `steady_state_model` block (see [steady_state_model], page 28); it will rather output a static version (*i.e.* without leads and lags) of the dynamic model declared in the `model` block.

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 occurence 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)`.

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 (determistic 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-loking 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, trigger a shock in first period, and study the trajectory of return at the initial equilbrium. To do that, one needs `initval` and `shocks` (see Section 4.8 [Shocks on exogenous variables], page 22.

Another one is to study, how an economy, starting from arbitrary initial conditions converges toward equilibrium. To do that, one needs `initval` and `endval`.

For models with lags on more than one period, the command `histval` permits to specify different historical initial values for periods before the beginning of the simulation.

`initval` *;* [Block]

Description

The `initval` block serves two purposes: declaring the initial (and possibly terminal) conditions in a simulation exercise, and providing guess values for non-linear solvers.

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

VARIABLE_NAME `=` *EXPRESSION*`;`

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

First, it provides the initial conditions for all the endogenous and exogenous variables at all the periods preceeding the first simulation period (unless some of these initial values are modified by `histval`).

Second, in the absence of an `endval` block, it sets the terminal conditions for all the periods succeeding the last simulation period.

Third, in the absence of an `endval` block, it provides initial guess values at all simulation dates for the non-linear solver implemented in `simul`.

For this last reason, it necessary to provide values for all the endogenous variables in an `initval` block (even though, theoretically, initial conditions are only necessary for lagged variables). If some exogenous variables are not mentionned in the `initval` block, a zero value is assumed.

Note that if the `initval` block is immediately followed by a `steady` command, its semantics is 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 `initval` block, and using the values declared for the endogenous as initial guess values for the non-linear solver. An `initval` block followed by `steady` is formally equivalent to an `initval` block with the same values for the exogenous, and with the associated steady state values for the endogenous.

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.

This steady state will be used as the initial condition at all the periods preceeding the first simulation period for the two possible types of simulations in stochastic mode:

- in `stoch_simul`, if the `periods` options is specified
- in `forecast` (in this case, note that it is still possible to modify some of these initial values with `histval`)

Example

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

steady;
```

`endval` *;* [Block]

Description

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

`VARIABLE_NAME = EXPRESSION;`

The `endval` block makes only sense in a determistic model, and serves two purposes.

First, it sets the terminal conditions for all the periods succeeding the last simulation period.

Second, it provides initial guess values at all the simulation dates for the non-linear solver implemented in `simul`.

For this last reason, it necessary to provide values for all the endogenous variables in an `endval` block (even though, theoretically, initial conditions are only necessary for forward variables). If some exogenous variables are not mentionned in the `endval` block, a zero value is assumed.

Note that if the `endval` block is immediately followed by a `steady` command, its semantics is 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, and using the values declared for the endogenous as initial guess values for the non-linear solver. An `endval` block followed by `steady` is formally equivalent to an `endval` block with the same values for the exogenous, and with the associated steady state values for the endogenous.

Example

```
var c k;
varexo x;
...
initval;
c = 1.2;
k = 12;
x = 1;
end;

steady;

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

steady;
```

The initial equilibrium is computed by `steady` for `x=1`, and the terminal one, for `x=2`.

`histval` *;* [Block]

Description

In models with lags on more than one period, the `histval` block permits to specify different historical initial values for different periods.

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.

If your lagged variables are linked by identities, be careful to satisfy these identities when you set historical initial values.

Variables not initialized in the `histval` block are assumed to have a value of zero at period 0 and before. Note that this behavior differs from the case where there is no `histval` block, where all variables are initialized at their steady state value at period 0 and before.

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;
```

`resid` *;* [Command]

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 Section 4.10 [Steady state], page 25).

`initval_file` (*filename* = *FILENAME*)*;* [Command]

Description

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 vector of the same name.
- MAT-file (extension '`.mat`'): same as for M-files.
- Excel file (extension '`.xls`'): for each endogenous and exogenous, the file must contain a column of the same name (not supported under Octave).

Warning

The extension must be omitted in the command argument. Dynare will automatically figure out the extension and select the appropriate file type.

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 direclty 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.

`shocks` *;* [Block]

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 must be matched by only one value in the `values` keyword: this syntax means that the exogenous variable will have a constant value over 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.

Here is an example:

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

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:

`var` *VARIABLE_NAME*`; stderr` *EXPRESSION*`;`
: Specifies the standard error of a variable.

`var` *VARIABLE_NAME* `=` *EXPRESSION*`;`
: Specifies the variance error of a variable.

`var` *VARIABLE_NAME*`,` *VARIABLE_NAME* `=` *EXPRESSION*`;`
: Specifies the covariance of two variables.

`corr` *VARIABLE_NAME*`,` *VARIABLE_NAME* `=` *EXPRESSION*`;`
: Specifies the correlation of two variables.

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.

Here is an example:

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

Mixing determininistic 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.

Here is an 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;
```

`mshocks` *;* [Block]

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 than `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.

`Sigma_e` [Special variable]

Warning

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

Description

This special variable specifies directly the covariance matrix of the stochastic shocks, as an upper (or lower) triangular matrix. Dynare builds the corresponding symmetrix 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`.

4.9 Other general declarations

`dsample` *INTEGER* [*INTEGER*]; [Command]

Reduces the number of periods considered in subsequent output commands.

`periods` *INTEGER;* [Command]

Description

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 `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 "steady state file".

4.10.1 Finding the steady state with Dynare nonlinear solver

`steady` *;* [Command]
`steady` (*OPTIONS...*); [Command]

Description

This command computes the steady state of a model using a nonlinear Newton-type solver.

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

`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

`2` Splits the model into recursive blocks and solves each block in turn

`3` Use Chris Sims' solver

`4` Similar to value `2`, except that it deals differently with nearly singular Jacobian

`5` Newton algorithm with a sparse Gaussian elimination (SPE) (requires `bytecode` option, see Section 4.5 [Model declaration], page 16)

`6` Newton algorithm with a sparse LU solver at each iteration (requires `bytecode` and/or `block` option, see Section 4.5 [Model declaration], page 16)

`7` Newton algorithm with a Generalized Minimal Residual (GMRES) solver at each iteration (requires `bytecode` and/or `block` option, see Section 4.5 [Model declaration], page 16; not available under Octave)

`8` Newton algorithm with a Stabilized Bi-Conjugate Gradient (BICGSTAB) solver at each iteration (requires `bytecode` and/or `block` option, see Section 4.5 [Model declaration], page 16)

Default value is `2`.

`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 boudaries for each parameter is divided in as many intervals as there are steps (as defined by `homotopy_steps` option); the problem is solves 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.

Example

See Section 4.7 [Initial and terminal conditions], page 19.

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

`oo_.steady_state` [MATLAB/Octave variable]

Contains the computed steady state.

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

`homotopy_setup ;` [Block]

Description

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 Using a steady state file

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`. If your MOD-file is called '*FILENAME*`.mod`', the steady state file should be called '*FILENAME*`_steadystate.m`'.

Again, there are two options for creating this file:

- You can write this file by hand. See '`fs2000_steadystate.m`' in the '`examples`' directory for an example. This is the option which gives the most flexibility, at the expense of a heavier programming burden.
- You can use the `steady_state_model` block, for a more user-friendly interface.

`steady_state_model` *;* [Block]

Description

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 using the information provided in this block.

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);
  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;
```

4.11 Getting information about the model

`check` *;* [Command]
`check` (*solve_algo* = *INTEGER*) *;* [Command]

Description

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.

Options

`solve_algo` = *INTEGER*
: See [solve_algo], page 26, for the possible values and their meaning.

Output

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

`oo_.dr.eigval` [MATLAB/Octave variable]
Contains the eigenvalues of the model, as computed by the `check` command.

`model_info ;` [Command]

Description

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 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 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`'.

`print_bytecode_dynamic_model ;` [Command]
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.

`print_bytecode_static_model ;` [Command]
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, Dynare uses a Newton-type algorithm, first proposed by *Laffargue (1990)* and *Boucekkine (1995)*, instead of a first order technique like the one proposed by *Fair and Taylor (1983)*, and used in earlier generation simulation programs. We believe this approach to be in general both faster and more robust. The details of the algorithm can be found in *Juillard (1996)*.

`simul` *;* [Command]
`simul` (*OPTIONS...*)*;* [Command]

Description

Triggers the computation of a deterministic simulation of the model for the number of periods set in the option `periods`.

Options

`periods` = *INTEGER*
: Number of periods of the simulation

`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, see *Juillard (1996)* (Default).

  `1` Use a Newton algorithm with a sparse LU solver at each iteration (requires `bytecode` and/or `block` option, see Section 4.5 [Model declaration], page 16).

  `2` Use a Newton algorithm with a Generalized Minimal Residual (GMRES) solver at each iteration (requires `bytecode` and/or `block` option, see Section 4.5 [Model declaration], page 16; not available under Octave)

  `3` Use a Newton algorithm with a Stabilized Bi-Conjugate Gradient (BICGSTAB) solver at each iteration (requires `bytecode` and/or `block` option, see Section 4.5 [Model declaration], page 16).

  `4` Use a Newton algorithm with a optimal path length at each iteration (requires `bytecode` and/or `block` option, see Section 4.5 [Model declaration], page 16).

  `5` Use a Newton algorithm with a sparse Gaussian elimination (SPE) solver at each iteration (requires `bytecode` option, see Section 4.5 [Model declaration], page 16).

`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`.

`datafile = FILENAME`
: If the variables of the model are not constant over time, their initial values, stored in a text file, could be loaded, using that option, as initial values before a deteministic simulation.

Output

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

`oo_.endo_simul` [MATLAB/Octave variable]

This variable stores the result of a deterministic simulation (computed by `simul`) or of a stochastic simulation (computed by `stoch_simul` with the `periods` option).

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.

4.13 Stochastic solution and simulation

In a stochastic context, Dynare computes one or several simulations corresponding to a random draw of the shocks. Dynare uses 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)*.

4.13.1 Computing the stochastic solution

`stoch_simul` [*VARIABLE_NAME...*]; [Command]
`stoch_simul` (*OPTIONS...*) [*VARIABLE_NAME...*]; [Command]

Description

`stoch_simul` solves a stochastic (*i.e.* rational expectations) model, using perturbation techniques.

More precisely, `stoch_simul` computes a Taylor approximation of the decision and transition functions for the 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 Section 4.10 [Steady state], page 25).

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.

Variance decomposition, correlation, autocorrelation are only displayed for variables with 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%.

Currently, the IRFs are only plotted for 12 variables. Select the ones you want to see, if your model contains more than 12 endogenous variables.

The covariance matrix of the shocks is specified with the `shocks` command (see Section 4.8 [Shocks on exogenous variables], page 22).

When a list of *VARIABLE_NAME* is specified, results are displayed only for these variables.

Options

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

`drop = INTEGER`
: Number of points dropped at the beginning of simulation before computing the summary statistics. Default: `100`.

`hp_filter = DOUBLE`
: Uses HP filter with $\lambda = DOUBLE$ before computing moments. Default: no filter.

`hp_ngrid = INTEGER`
: Number of points in the grid for the discrete Inverse Fast Fourier Transform used in the HP filter computation. 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 IRF's. Default: `40`.

`relative_irf`
: Requests the computation of normalized IRFs in percentage of the standard error of each shock.

`linear` Indicates that the original model is linear (put it rather in the `model` command).

`nocorr` Don't print the correlation matrix (printing them is the default).

`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.` Doesn't do the graphs. Useful for loops.

`noprint` Don't print anything. Useful for loops.

`print` Print results (opposite of `noprint`).

`order = INTEGER`
: Order of Taylor approximation. Acceptable values are `1`, `2` and `3`. Note that for third order, `k_order_solver` option is implied and only empirical moments are available (you must provide a value for `periods` option). Default: `2`.

`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 Section 4.5 [Model declaration], page 16. Default: disabled for order 1 and 2, enabled otherwise

`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], page 32). Default: `0`.

`qz_criterium = DOUBLE`
: Value used to split stable from unstable eigenvalues in reordering the Generalized Schur decomposition used for solving 1^st order problems. Default: `1.000001` (except when estimating with `lik_init` option equal to `1`: the default is `0.999999` in that case; see Section 4.14 [Estimation], page 38).

`replic = INTEGER`
: Number of simulated series used to compute the IRFs. Default: `1` if `order=1`, and `50` otherwise.

`solve_algo = INTEGER`
: See [solve_algo], page 26, 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`
: See below.

`conditional_variance_decomposition = [INTEGER1:INTEGER2]`
: See below.

`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], page 38).

`pruning` Discard higher order terms when iteratively computing simulations of the solution, as in *Kim, Kim, Schaumburg and Sims (2008)*.

`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 at `http://www.dynare.org/DynareWiki/PartialInformation`.

Output

This command sets `oo_.dr`, `oo_.mean`, `oo_.var` and `oo_.autocorr`, which are described below.

If option `periods` is present, sets `oo_.endo_simul` (see [oo_.endo_simul], page 32), and also saves the simulated variables in MATLAB/Octave vectors of the global workspace with the same name as the endogenous variables.

If options `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).

Example 1

```
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 2

```
stoch_simul(linear,irf=60) y k;
```

Performs the simulation of a linear model and displays impulse response functions on 60 periods for variables `y` and `k`.

`oo_.mean` [MATLAB/Octave variable]

After a run of `stoch_simul`, contains the mean of the endogenous variables. Contains theoretical mean if the `periods` option is not present, and empirical mean otherwise. The variables are arranged in declaration order.

`oo_.var` [MATLAB/Octave variable]

After a run of `stoch_simul`, contains the variance-covariance of the endogenous variables. Contains theoretical variance if the `periods` option is not present, and empirical variance otherwise. The variables are arranged in declaration order.

`oo_.autocorr` [MATLAB/Octave variable]

After a run of `stoch_simul`, contains a cell array of the autocorrelation matrices of the endogenous variables. The element number of the matrix in the cell array corresponds to the order of autocorrelation. The option `ar` specifies the number of autocorrelation matrices available. Contains theoretical autocorrelations if the `periods` option is not present, and empirical autocorrelations otherwise.

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}`.

`oo_.gamma_y` [MATLAB/Octave variable]

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/co-variance matrix.

`oo_.gamma{i+1} (for i=1:ar)`
: Autocorrelation function. see [oo_.autocorr], page 35 for more details. Beware, this is the *autocorrelation* function, not the *autocovariance* function.

`oo_.gamma{nar+2}`
: Variance decomposition.

`oo_.gamma{nar+3}`
: If a second order approximation has been requested, contains the vector of the mean correction terms.

`oo_.irfs` [MATLAB/Octave variable]
: 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`.

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 oberved at the beginning of the period. The decision rules are stored in the structure `oo_.dr` which is described below.

4.13.2 Typology and ordering of variables

Dynare distinguishes four types of endogenous variables:

Purely backward (or purely predetermined) variables
: Those that appear only at current and past period in the model, but not at future period (*i.e.* at t and $t-1$ but not $t+1$). The number of such variables is equal to `oo_.dr.npred - oo_.dr.nboth`.

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 `oo_.dr.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 `oo_.dr.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 `oo_.dr.nstatic`.

Note that all endogenous variables fall into one of these four categories, since after the creation of auxiliary variables (see Section 4.6 [Auxiliary variables], page 18), all endogenous have at most one lead and one lag. We therefore have the following identity:

```
oo_.dr.npred + oo_.dr.nfwrd + oo_.dr.nstatic = M_.endo_nbr
```

Internally, Dynare uses two orderings of the endogenous variables: the order of declaration (which is reflected in `M_.endo_names`), and an order based on the four types described above, which we will call the DR-order ("DR" stands for decision rules). Most of the time, the declaration order is used, but for elements of the decision rules, the DR-order is used.

The DR-order is the following: static variables appear first, then purely backward variables, then mixed variables, and finally purely forward variables. Inside each category, variables are arranged according to the declaration order.

Variable `oo_.dr.order_var` maps DR-order to declaration order, and variable `oo_.dr.inv_order_var` 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 orderer in DR-order when they appear in decision rules elements. There are `oo_.dr.npred` such variables.

4.13.3 First order approximation

The approximation has the form:

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

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

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.

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, $y^h_t = y_t - y^s$, and Δ^2 is the shift effect of the variance of future shocks.

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 $n_z =$ `oo_.dr.npred + 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.

`oo_.conditional_variance_decomposition` [MATLAB/Octave variable]

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 forecast horizons (as declared with the option), the second dimension corresponds to endogenous variables (in the order of declaration), the third dimension corresponds to exogenous variables (in the order of declaration).

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.

`varobs` *VARIABLE_NAME...;* [Command]

Description

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], page 41).

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 macroprocessor can be used as shown in the second example below.

Simple example

```
varobs C y rr;
```

Example with a loop

```
varobs
@#for co in countries
  GDP_@{co}
@#endfor
;
```

`observation_trends` *;* [Block]

Description

This block specifies *linear* trends for observed variables as functions of model parameters.

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;
```

`estimated_params` *;* [Block]

Description

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 three following alternatives:

`stderr` *VARIABLE_NAME*
: Indicates that the standard error of the exogenous variable *VARIABLE_NAME*, or of the observation error 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 associated with endogenous observed variables *VARIABLE_NAME1* and *VARIABLE_NAME2*, is to be estimated

PARAMETER_NAME
: The name of a model parameter to be estimated

`DSGE_PRIOR_WEIGHT`
: ...

The rest of the line consists of the following fields, some of them being optional:

INITIAL_VALUE
: Specifies a starting value for maximum likelihood estimation

LOWER_BOUND
: Specifies a lower bound for the parameter value in maximum likelihood estimation

UPPER_BOUND
: Specifies an upper bound for the parameter value in maximum likelihood estimation

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`. 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 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
: The scale parameter to be used for the jump distribution 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.

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.

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 Section 4.5 [Model declaration], page 16).

Example

```
parameters bet;

model;
# sig = 1/bet;
c = sig*c(+1)*mpk;
```

```
end;

estimated_params;
bet, normal_pdf, 1, 0.05;
end;
```

`estimated_params_init` *;* [Block]

This block declares numerical initial values for the optimizer when these ones are different from the prior mean.

Each line has the following syntax:

```
stderr VARIABLE_NAME | corr VARIABLE_NAME_1, VARIABLE_NAME_2 | PARAMETER_NAME
, INITIAL_VALUE;
```

See [estimated_params], page 39, for the meaning and syntax of the various components.

`estimated_params_bounds` *;* [Block]

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], page 39, for the meaning and syntax of the various components.

`estimation` [*VARIABLE_NAME...*]*;* [Command]
`estimation` (*OPTIONS...*) [*VARIABLE_NAME...*]*;* [Command]

Description

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 density
- mean and shortest confidence interval from posterior simulation
- Metropolis-Hastings convergence graphs that still need to be documented
- graphs with prior, posterior and mode
- graphs of smoothed shocks, smoothed observation errors, smoothed and historical variables

Options

`datafile` = *FILENAME*
: The datafile (a '`.m`' file, a '`.mat`' file or, under MATLAB, a '`.xls`' 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

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

`nobs` = [*INTEGER1*:*INTEGER2*]
: Runs a recursive estimation and forecast for samples of size ranging of *INTEGER1* to *INTEGER2*. Option `forecast` must also be specified

`first_obs` = *INTEGER*
: The number of the first observation to be used. Default: `1`

`prefilter` = *INTEGER*
: A value of `1` means that the estimation procedure will demean the data. Default: `0`, *i.e.* no prefiltering

`presample` = *INTEGER*
: The number of observations to be skipped before evaluating the likelihood. Default: `0`

`loglinear`
: Computes a log-linear approximation of the model instead of a linear approximation. The data must correspond to the definition of the variables used in the model. Default: computes a linear approximation

`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 choosen values for the prior standard density can result in absurd prior densities.

  Default value is `1`.

`nograph` No graphs should be plotted

`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

  `3` For nonstationary models: ...

  Default value is `1`.

`lik_algo` = *INTEGER*
: ...

`conf_sig` = *DOUBLE*
: See [conf_sig], page 49.

`mh_replic` = *INTEGER*
: Number of replications for Metropolis-Hastings algorithm. For the time being, `mh_replic` should be larger than `1200`. Default: `20000`

`mh_nblocks` = *INTEGER*
: Number of parallel chains for Metropolis-Hastings algorithm. Default: `2`

`mh_drop` = *DOUBLE*
: The fraction of initially generated parameter vectors to be dropped before using posterior simulations. Default: `0.5`

`mh_jscale` = *DOUBLE*
: The scale to be used for the jumping distribution in Metropolis-Hastings algorithm. The default value is rarely satisfactory. This option must be tuned to obtain, ideally, an acceptation rate of 25% in the Metropolis-Hastings algorithm. Default: `0.2`

`mh_init_scale` = *DOUBLE*
: The scale to be used for drawing the initial value of the Metropolis-Hastings chain. Default: 2*`mh_scale`

`mh_recover`
: Attempts to recover a Metropolis-Hastings simulation that crashed prematurely. Shouldn't be used together with `load_mh_file`

`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`) in a file called '*MODEL_FILENAME*`_mode.mat`'

`mode_compute = INTEGER | FUNCTION_NAME`
: Specifies the optimizer for the mode computation:

  `0` The mode isn't computed. `mode_file` option must be specified

  `1` Uses `fmincon` optimization routine (not available under Octave)

  `2` Value no longer used

  `3` Uses `fminunc` optimization routine

  `4` Uses Chris Sims's `csminwel`

  `5` Uses Marco Ratto's `newrat`

  `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)

  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`.

`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

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

`load_mh_file`
: Tells Dynare to add to previous Metropolis-Hastings simulations instead of starting from scratch. Shouldn't be used together with `mh_recover`

`optim = (fmincon options)`
: Can be used to set options for `fmincon`, the optimizing function of MATLAB Optimization toolbox. Use MATLAB's syntax for these options. Default: `('display','iter','LargeScale','off','MaxFunEvals',100000,'TolFun',1e-8,'TolX',1e-6)`

`nodiagnostic`
: Doesn't 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)

`dsge_var`
: Triggers the estimation of a DSGE-VAR model, where the weight of the DSGE prior of the VAR model will be estimated. 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_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. 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.

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

`moments_varendo`
: Triggers the computation of the posterior distribution of the theoretical moments of the endogenous variables. Results are stored in `oo_.PosteriorTheoreticalMoments` (see below for a description of this variable)

`filtered_vars`
: Triggers the computation of the posterior distribution of filtered endogenous variables and shocks. Results are stored in `oo_.FilteredVariables` (see below for a description of this variable)

`smoother`
: Triggers the computation of the posterior distribution of smoothered endogenous variables and shocks. Results are stored in `oo_.SmoothedVariables`, `oo_.SmoothedShocks` and `oo_.SmoothedMeasurementErrors` (see below for a description of these variables)

`forecast = `*`INTEGER`*
: Computes the posterior distribution of a forecast on *INTEGER* periods after the end of the sample used in estimation. The result is stored in variable `oo_.forecast` (see Section 4.15 [Forecasting], page 49)

`tex`
: Requests the printing of results and graphs in TeX tables and graphics that can be later directly included in LaTeX files (not yet implemented)

`kalman_algo = `*`INTEGER`*
: ...

`kalman_tol = `*`DOUBLE`*
: ...

`filter_covariance`
: Saves the series of one step ahead error of forecast covariance matrices.

`filter_step_ahead = [`*`INTEGER1`*`:`*`INTEGER2`*`]`
: Triggers the computation k-step ahead filtered values.

`filter_decomposition`
: Triggers the computation of the shock decomposition of the above k-step ahead filtered values.

`constant` ...

`noconstant`
: ...

`diffuse_filter`
: ...

`selected_variables_only`
: Only run the smoother on the variables listed just after the `estimation` command. 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. Default is `1`.

`solve_algo = INTEGER`
: See [solve_algo], page 26.

`order = INTEGER`
: See [order], page 33.

`irf = INTEGER`
: See [irf], page 33. Only used if [bayesian_irf], page 44 is passed.

`aim_solver`
: See [aim_solver], page 34.

Note

If no `mh_jscale` parameter is used 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.

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.

Running the smoother with calibrated parameters

It is possible to compute smoothed value of the endogenous variables and the shocks with calibrated parameters, without estimation proper. For this usage, there should be no `estimated_params` block. Observed variables must be declared. A dataset must be specified in the `estimation` instruction. In addition, use the following options: `mode_compute=0,mh_replic=0,smoother`. Currently, there is no specific output for this usage of the `estimation` command. The results are made available in fields of `oo_` structure. An example is available in '`./tests/smoother/calibrated_model.mod`'.

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[3]

`HPDsup` Upper bound of a 90% HPD interval

`Mean` Mean of the posterior distribution

`Median` Median of the posterior distribution

`Std` Standard deviation of the posterior distribution

`deciles` Deciles of the distribution.

`density` Non parametric estimate of the posterior density. 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

`oo_.MarginalDensity.LaplaceApproximation` [MATLAB/Octave variable]
Variable set by the `estimation` command.

`oo_.MarginalDensity.ModifiedHarmonicMean` [MATLAB/Octave variable]
Variable set by the `estimation` command, if it is used with `mh_replic > 0` or `load_mh_file` option.

`oo_.FilteredVariables` [MATLAB/Octave variable]
Variable set by the `estimation` command, if it is used with the `filtered_vars` option. Fields are of the form:

`oo_.FilteredVariables.`*`VARIABLE_NAME`*

`oo_.PosteriorIRF.dsge` [MATLAB/Octave variable]
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`*

`oo_.SmoothedMeasurementErrors` [MATLAB/Octave variable]
Variable set by the `estimation` command, if it is used with the `smoother` option. Fields are of the form:

`oo_.SmoothedMeasurementErrors.`*`VARIABLE_NAME`*

`oo_.SmoothedShocks` [MATLAB/Octave variable]
Variable set by the `estimation` command, if it is used with the `smoother` option. Fields are of the form:

`oo_.SmoothedShocks.`*`VARIABLE_NAME`*

[3] See option [conf_sig], page 49 to change the size of the HPD interval

`oo_.SmoothedVariables` [MATLAB/Octave variable]
Variable set by the `estimation` command, if it is used with the `smoother` option. Fields are of the form:

`oo_.SmoothedVariables.`*`VARIABLE_NAME`*

`oo_.PosteriorTheoreticalMoments` [MATLAB/Octave variable]
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

`correlation`
: Correlation between endogenous variables

`VarianceDecomposition`
: Decomposition of variance[4]

`ConditionalVarianceDecomposition`
: Only if the `conditional_variance_decomposition` option has been specified

`oo_.posterior_density` [MATLAB/Octave variable]
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`*

`oo_.posterior_hpdinf` [MATLAB/Octave variable]
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`*

`oo_.posterior_hpdsup` [MATLAB/Octave variable]
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`*

`oo_.posterior_mean` [MATLAB/Octave variable]
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`*

`oo_.posterior_mode` [MATLAB/Octave variable]
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_mode.`*`ESTIMATED_OBJECT`*`.`*`VARIABLE_NAME`*

`oo_.posterior_std` [MATLAB/Octave variable]
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`*

[4] When the shocks are correlated, it is the decomposition of orthogonalized shocks via Cholesky decompostion according to the order of declaration of shocks (see Section 4.2 [Variable declarations], page 9)

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
```

`model_comparison` *FILENAME*[(*DOUBLE*)]...; [Command]
`model_comparison` (*marginal_density = laplace | modifiedharmonicmean*) [Command]
FILENAME[(*DOUBLE*)]...;

Description

This command computes odds ratios and estimate a posterior density over a collection of models. The priors over models can be specified as the *DOUBLE* values, otherwise a uniform prior is assumed.

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`.

`shock_decomposition` [*VARIABLE_NAME*]...; [Command]
`shock_decomposition` (*OPTIONS*...) [*VARIABLE_NAME*]...; [Command]

Description

This command computes and displays shock decomposition according to the model for a given sample.

Options

`parameter_set` = *PARAMETER_SET*
: Specify the parameter set to use for running the smoother. The *PARAMETER_SET* can take one of the following five values: `prior_mode`, `prior_mean`, `posterior_mode`, `posterior_mean`, `posterior_median`. Default value: `posterior_mean` if Metropolis has been run, else `posterior_mode`.

`unit_root_vars` *VARIABLE_NAME*...; [Command]
`unit_root_vars` is used to declare a list of unit-root endogenous variables of a model so that dynare won't check the steady state levels (defined in the steadystate file) file for these variables. The information given by this command is no more used for the initialization of the diffuse kalman filter (as described in *Durbin and Koopman (2001)* and *Koopman and Durbin (2003)*).

When `unit_root_vars` is used the `lik_init` option of `estimation` has no effect.

When there are nonstationary variables in a model, there is no unique deterministic steady state. The user must supply a MATLAB/Octave function that computes the steady state values of the stationary variables in the model and returns dummy values for the nonstationary ones. The function should be called with the name of the '`.mod`' file followed by '`_steadystate`'. See '`fs2000_steadystate.m`' in '`examples`' directory for an example.

Note that the nonstationary variables in the model must be integrated processes (their first difference or k-difference must be stationary).

Dynare also has the ability to estimate Bayesian VARs:

`bvar_density` ; [Command]
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 Forecasting

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

`forecast` [*VARIABLE_NAME...*]; [Command]
`forecast` (*OPTIONS...*) [*VARIABLE_NAME...*]; [Command]

Description

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 conditionaly 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: `40`

`conf_sig` = *DOUBLE*
: Level of significance for confidence interval. Default: `0.90`

`nograph`
: Don't display graphics.

Output

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

Example

```
varexo_det tau;
varexo e;

...

shocks;
var e; stderr 0.01;
var tau;
periods 1:9;
values -0.15;
end;

stoch_simul(irf=0);

forecast;
```

`oo_.forecast` [MATLAB/Octave variable]

Variable set by the `forecast` command, or by the `estimation` command if used with the `forecast` option. 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[5] of forecast due to parameter uncertainty

`HPDsup` Lower bound of a 90% HPD interval due to parameter uncertainty

`HPDTotalinf`
Lower bound of a 90% HPD interval of forecast due to parameter uncertainty and future shocks (only with the `estimation` command)

`HPDTotalsup`
Lower bound of a 90% HPD interval due to parameter uncertainty and future shocks (only with the `estimation` command)

`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

`conditional_forecast (`*`OPTIONS...`*`) [`*`VARIABLE_NAME...`*`];` [Command]

Description

This command computes forecasts on an estimated model for a given constrained path of some 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. This command has to be called after estimation.

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 = prior_mode | prior_mean | posterior_mode | posterior_mean | posterior_median`
Specify the parameter set to use for the forecasting. 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 less 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`

[5] See option [conf_sig], page 49 to change the size of the HPD interval

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

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

`conditional_forecast_paths` *;* [Block]

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 than the deterministic shocks in the `shocks` blocks (see Section 4.8 [Shocks on exogenous variables], page 22).

`plot_conditional_forecast` [*VARIABLE_NAME...*]*;* [Command]
`plot_conditional_forecast` (*periods* = *INTEGER*) [*VARIABLE_NAME...*]*;* [Command]

Description

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`.

`bvar_forecast` *;* [Command]

This command computes in-sample or out-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.

4.16 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_policy` or for optimal simple rule with `osr`.

`osr` [*VARIABLE_NAME...*]; [Command]
`osr` (*OPTIONS...*) [*VARIABLE_NAME...*]; [Command]

Description

This command computes optimal simple policy rules for linear-quadratic problems of the form:

$\max_\gamma E(y_t' W y_t)$

such that:

$A_1 E_t y_{t+1} + A_2 y_t + A_3 y_{t-1} + C e_t = 0$

where:

- γ are parameters to be optimized. They must be elements of matrices A_1, A_2, A_3;
- y are the endogenous variables;
- e are the exogenous stochastic shocks;

The parameters to be optimized must be listed with `osr_params`.

The quadratic objectives must be listed with `optim_weights`.

This problem is solved using a numerical optimizer.

Options

This command accept the same options than `stoch_simul` (see Section 4.13.1 [Computing the stochastic solution], page 32).

`osr_params` *PARAMETER_NAME...*; [Command]
This command declares parameters to be optimized by `osr`.

`optim_weights` *;* [Block]
This block specifies quadratic objectives for optimal policy problems

More precisely, this block specifies the nonzero elements of the quadratic weight matrices for the objectives in `osr`.

A 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*;

`ramsey_policy` [*VARIABLE_NAME...*]; [Command]
`ramsey_policy` (*OPTIONS...*) [*VARIABLE_NAME...*]; [Command]

Description

This command computes the first order approximation of the policy that maximizes the policy maker objective function submitted to the constraints provided by the equilibrium path of the economy.

The planner objective must be declared with the `planner_objective` command.

Options

This command accepts all options of `stoch_simul`, plus:

`planner_discount` = *DOUBLE*
Declares the discount factor of the central planner. Default: `1.0`

Note that only first order approximation is available (*i.e.* `order=1` must be specified).

Output

This command generates all the output variables of `stoch_simul`.

In addition, it stores the value of planner objective function under Ramsey policy in `oo_.planner_objective_value`.

`planner_objective` *MODEL_EXPRESSION*; [Command]

This command declares the policy maker objective, for use with `ramsey_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_policy`.

Note that with this command you are not limited to quadratic objectives: you can give any arbitrary nonlinear expression.

4.17 Sensitivity and identification analysis

`dynare_sensitivity` ; [Command]
`dynare_sensitivity` (*OPTIONS...*); [Command]

This function is an interface to the global sensitivity analysis (GSA) toolbox developed by the Joint Research Center (JRC) of the European Commission. The GSA toolbox needs to be downloaded separately from the JRC web site.

Please refer to the documentation of the GSA toolbox on the official website for more details on the usage of this command.

`identification` ; [Command]
`identification` (*OPTIONS...*); [Command]

Description

This command triggers identification analysis.

Options

`ar` = *INTEGER*
: Number of lags of computed autocorrelations (theoretical moments). Default: `3`

`useautocorr` = *INTEGER*
: If equal to `1`, compute derivatives of autocorrelation. If equal to `0`, compute derivatives of autocovariances. Default: `1`

`load_ident_files` = *INTEGER*
: If equal to `1`, allow Dynare to load previously computed analyzes. Default: `0`

`prior_mc` = *INTEGER*
: Size of Monte Carlo sample. Default: `2000`

`lik_init` = *INTEGER*
: See [lik_init], page 42

4.18 Displaying and saving results

Dynare has comments to plot the results of a simulation and to save the results.

`rplot` *VARIABLE_NAME...*; [Command]

Plots the simulated path of one or several variables, as stored in *oo_.endo_simul* by either *simul* (see Section 4.12 [Deterministic simulation], page 31) or *stoch_simul* with option *periods* (see Section 4.13.1 [Computing the stochastic solution], page 32). The variables are plotted in levels.

`dynatype` (*FILENAME*) [*VARIABLE_NAME*...]; [Command]
This command prints the listed variables in a text file named *FILENAME*. If no *VARIABLE_NAME* is listed, all endogenous variables are printed.

`dynasave` (*FILENAME*) [*VARIABLE_NAME*...]; [Command]
This command saves the listed variables in a binary file named *FILENAME*. If no *VARIABLE_NAME* are listed, all endogenous variables are saved.

In MATLAB or Octave, variables saved with the `dynasave` command can be retrieved by the command:

```
load -mat FILENAME
```

4.19 Macro-processing language

It is possible to use "macro" commands in the '`.mod`' file for doing the following tasks: 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 inserted inside '`.mod`' files. It features:

- file inclusion
- loops (`for` structure)
- conditional inclusion (`if/then/else` structures)
- expression substitution

Technically, 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 Chapter 3 [Dynare invocation], page 6).

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. In case of need, two anti-slashes (\\) at the end of the line indicates that the directive is continued on the next line. The main directives are:

- `@#include`, for file inclusion,
- `@#define`, for defining a macro-processor variable,
- `@#if`, `@#else`, `@#endif` for conditional statements,
- `@#for`, `@#endfor` for constructing loops.

The macro-processor maintains its own list of variables (distinct of model variables and of MATLAB/Octave variables). These macro-variables are assigned using the `@#define` directive, and can be of four types: integer, character string, array of integers, array of strings.

4.19.1 Macro expressions

It is possible to construct macro-expressions which can be assigned to macro-variables or used within a macro-directive. The expressions are constructed using literals of the four basic types (integers, strings, arrays of strings, arrays of integers), macro-variables names and standard operators.

String literals have to be enclosed between **double** quotes (like `"name"`). Arrays are enclosed within brackets, and their elements are separated by commas (like `[1,2,3]` or `["US", "EA"]`).

Note that there is no boolean type: *false* is represented by integer zero and *true* is any non-null integer.

The following operators can be used on integers:

- arithmetic operators: `+`, `-`, `*`, `/`
- comparison operators: `<`, `>`, `<=`, `>=`, `==`, `!=`
- logical operators: `&&`, `||`, `!`
- integer ranges, using the following syntax: *`INTEGER1`*`:`*`INTEGER2`* (for example, `1:4` is equivalent to integer array `[1,2,3,4]`)

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

The following operators can be used on arrays:

- dereferencing: if *`v`* is an array, then *`v`*`[2]` is its 2nd element
- concatenation of two arrays: `+`
- difference `-`: returns the first operand from which the elements of the second operand have been removed
- 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`)

Macro-expressions can be used at 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.

In the following, *MACRO_EXPRESSION* designates an expression constructed as explained above.

4.19.2 Macro directives

`@#include "`*`FILENAME`*`"` [Macro directive]

This directive simply includes the content of another file at the place where it is inserted. It is exactly equivalent to a copy/paste of the content of the included file. Note that it is possible to nest includes (*i.e.* to include a file from an included file).

Example

```
@#include "modelcomponent.mod"
```

`@#define` *`MACRO_VARIABLE`* = *`MACRO_EXPRESSION`* [Macro directive]

Defines a macro-variable.

Example 1

```
@#define x = 5              // Integer
@#define y = "US"           // String
@#define v = [ 1, 2, 4 ]    // Integer array
@#define w = [ "US", "EA" ] // String array
@#define z = 3 + v[2]       // Equals 5
@#define t = ("US" in w)    // Equals 1 (true)
```

Example 2

```
@#define x = [ "B", "C" ]
@#define i = 2

model;
  A = @{x[i]};
end;
```

is strictly equivalent to:

```
model;
  A = C;
end;
```

`@#if` *MACRO_EXPRESSION* [Macro directive]
`@#else` [Macro directive]
`@#endif` [Macro directive]

Conditional inclusion of some part of the '`.mod`' file. The lines between `@#if` and the next `@#else` or `@#endif` is executed only if the condition evaluates to a non-null integer. The `@#else` branch is optional and, if present, is only evaluated if the condition evaluates to `0`.

Example

Choose between two alternative monetary policy rules using a macro-variable:

```
@#define linear_mon_pol = 0 // or 1
...
model;
@#if linear_mon_pol
  i = w*i(-1) + (1-w)*i_ss + w2*(pie-piestar);
@#else
  i = i(-1)^w * i_ss^(1-w) * (pie/piestar)^w2;
@#endif
...
end;
```

`@#for` *MACRO_VARIABLE* *in* *MACRO_EXPRESSION* [Macro directive]
`@#endfor` [Macro directive]

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;
```

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;
```

`@#echo` *MACRO_EXPRESSION* [Macro directive]

Asks the preprocessor to display some message on standard output. The argument must evaluate to a string.

`@#error` *MACRO_EXPRESSION* [Macro directive]
Asks the preprocessor to display some error message on standard output and to abort. The argument must evaluate to a string.

4.19.3 Typical usages

4.19.3.1 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 parameters 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 it is no longer needed to manually copy/paste the whole model (at the beginning) or changes to the model (during development).

4.19.3.2 Indexed sums or 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 = 1/5*(
        +x(-2)
        +x(-1)
        +x(0)
        +x(1)
        +x(2)
       );
...
end;
```

4.19.3.3 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;
```

4.19.3.4 Endogeneizing parameters

When doing the steady state calibration of the model, it may be useful to consider a parameter as an endogenous (and vice-versa).

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

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

The labor share in GDP is defined as:

`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 straigthforward; but 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 consist 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.

4.19.4 MATLAB/Octave loops versus macro-processor loops

Suppose you have a model with a parameter ρ, and you want to make 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 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 list of values directly given in the loop construct. Note that values are given as character strings (the macro-processor does not know floating point values. The inconvenient is that you can not reuse an array stored in a MATLAB/Octave variable.

4.20 Misc commands

`set_dynare_seed` (*INTEGER*) [Command]
`set_dynare_seed` (*'default'*) [Command]
`set_dynare_seed` (*'reset'*) [Command]
`set_dynare_seed` (*'ALGORITHM', INTEGER*) [Command]

Sets the seed used for random number generation.

`save_params_and_steady_state (`*FILENAME*`);` [Command]

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 Section 4.19 [Macro-processing language], page 54).

`load_params_and_steady_state (`*FILENAME*`);` [Command]

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, 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.

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 Mac OS X, the default location of the configuration file is '`$HOME/.dynare`', while on Windows it is '`%APPDATA%\dynare.ini`' (typically '`C:\Documents and Settings\`*`USERNAME`*`\Application Data\dynare.ini`' under Windows XP, or '`C:\Users\`*`USERNAME`*`\AppData\dynare.ini`' under Windows Vista or Windows 7). You can specify a non standard location using the `conffile` option of the `dynare` command (see Chapter 3 [Dynare invocation], page 6).

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 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 Metropolis-Hastings algorithm;
- 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 see Chapter 3 [Dynare invocation], page 6.

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;
- PsTools 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.

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:

`[cluster]` [Configuration block]

Description

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 NODE_NAME ...`
: A list of nodes that comprise the cluster. Each node is separated by at least one space. At the current time, all nodes specified by `Members` option must run the same type of operating system (*i.e.* all Windows or all Linux/Mac OS X). The platform versions don't matter (*i.e.* you can mix Windows XP and 7).

Example

```
[cluster]
Name = c1
Members = n1 n2 n3
```

`[node]` [Configuration block]

Description

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 Mac OS X, 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).

`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`.

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

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
CPUnbr = [2:4]
```

```
UserName = usern
RemoteDirectory = /home/usern/Remote
DynarePath = /home/usern/dynare/matlab
MatlabOctavePath = matlab
```

6 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).

'`fs2000.mod`'
: A cash in advance model, estimated by *Schorfheide (2000)*.

'`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)*.

7 Bibliography

- Backus, David K., Patrick J. Kehoe, and Finn E. Kydland (1992): "International Real Business Cycles," *Journal of Political Economy*, 100(4), 745–775.
- Boucekkine, Raouf (1995): "An alternative methodology for solving nonlinear forward-looking models," *Journal of Economic Dynamics and Control*, 19, 711–734.
- 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.
- Durbin, J. and S. J. Koopman (2001), *Time Series Analysis by State Space Methods*, Oxford University Press.
- Fair, Ray and John Taylor (1983): "Solution and Maximum Likelihood Estimation of Dynamic Nonlinear Rational Expectation Models," *Econometrica*, 51, 1169–1185.
- Fernandez-Villaverde, Jesus and Juan Rubio-Ramirez (2004): "Comparing Dynamic Equilibrium Economies to Data: A Bayesian Approach," *Journal of Econometrics*, 123, 153–187.
- Ireland, Peter (2004): "A Method for Taking Models to the Data," *Journal of Economic Dynamics and Control*, 28, 1205–26.
- 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, 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.
- 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.
- Laffargue, Jean-Pierre (1990): "Résolution d'un modèle macroéconomique avec anticipations rationnelles", *Annales d'Économie et Statistique*, 17, 97–119.
- 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.
- Mancini-Griffoli, Tommaso (2007): "Dynare User Guide: An introduction to the solution and estimation of DSGE models".
- Pearlman, Joseph, David Currie, and Paul Levine (1986): "Rational expectations models with partial information," *Economic Modelling*, 3(2), 90–105.
- 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.
- 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.

- 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.
- Villemot, Sébastien (2011): "Solving rational expectations models at first order: what Dynare does," *Dynare Working Papers*, 2, CEPREMAP

Command and Function Index

T

U

V

W

Variable Index

M

O

S