Dynare

September 11, 2024

Contents

Part I

Home

Chapter 1

The Dynare Julia Reference Manual

Chapter 2

Introduction

> **Note**
>
> This documentation is also available in PDF format: Dynare.pdf.

DynareJulia is a rewriting of Dynare (https://www.dynare.org) that was initially written in Gauss in 1994 and rewritten in Matlab around 2000.

Dynare provides several algorithms to work with Dynamic Stochastic General Equilibrium (DSGE) models often used in macroeconomics. Among other features, it helps

- solving such models,
- simulating them,
- estimating the parameters,
- making forecasts.

The user of the package writes a text file, usually with an `.mod` extension, that contains the equations of the model and the computation tasks. Then, DynareJulia compiles the model and runs the computations.

DynareJulia honors a subset of commands valid in DynareMatlab. Tell us if one of your favorite command or option is missing.

For many computing tasks, DynareJulia provides also Julia functions that can be used in the `*.mod` file or issued interactively after having run the `*.mod` file. These Julia functions use keyword arguments for the options and you need only to enter them if you want to change the default value. The keyword arguments without a default value are required arguments. In the sections of this documentation, the Dynare Commands are presented first, then the Julia functions.

Dynare has benefited from many contributions over the years. Here is a list of the contributors:

2.1 The Dynare Team

Currently the development team of Dynare is composed of:

- Stéphane Adjemian (Le Mans Université, Gains)
- Michel Juillard (Banque de France)

- Sumudu Kankanamge (Toulouse School of Economics and CEPREMAP)
- Frédéric Karamé (Le Mans Université, Gains and CEPREMAP)
- Junior Maih (Norges Bank)
- Willi Mutschler (University of Tübingen)
- Johannes Pfeifer (Universität der Bundeswehr München)
- Marco Ratto (European Commission, Joint Research Centre - JRC)
- Normann Rion (CY Cergy Paris Université and CEPREMAP)
- Sébastien Villemot (CEPREMAP)

The following people contribute or have contributed to DynareJulia

- Satyanarayana Bade
- Petre Caraiani
- Lilith Hafner
- Michel Juillard
- Félix Ordoñez
- Louis Ponet
- Rohit Singh Rathaur
- Dawie van Lill

The following people used to be members of the Dynare team:

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

Part II

Installation and Configuration

Chapter 3

Installation and configuration

3.1 Software requirements

Dynare is available for all plateforms supported by the current stable version of Julia (https://julialang.org/downloads/#supported_platforms). It should also work with older versions of Julia starting with version 1.6.3

3.2 Installation of Dynare

In Julia, install the Dynare.jl package:

```
using Pkg
pkg"add Dynare"
```

3.3 Using Dynare

In order to start using Dynare in Julia, type

```
using Dynare
```

3.4 Optional extensions

Pardiso

If you want the solution of very large perfect foresight models and reduce the memory consumption, use the Pardiso package (https://github.com/JuliaSparse/Pardiso.jl) and type

```
using MKL, Pardiso
```

PATHSolver

If youw want to solve perfect foresight models with occasionally binding constraints use the PATHSolver package (https://github.com/chkwon/PATHSolver.jl) and type

```
using PATHSolver
```

Part III

Running Dynare

Chapter 4

Running Dynare

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 are described in The `model file`

4.1 Dynare invocation

Once the model file is written, Dynare is invoked using the `@dynare` Julia macro (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 Julia instructions are generated; in the second step, the program actually runs the computations. Both steps are triggered automatically by the `@dynare` macro:

```
context = @dynare "FILENAME [.mod ]" [OPTIONS... ]";
```

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

Dynare begins by launching the preprocessor on the `.mod file`.

Options

- `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 (@ref "Macro processing language")); the saved output will go in the file specified, or if no file is specified in FILENAME-macroexp.mod. See the (@ref "note on quotes") for info on passing a FILENAME argument containing spaces.

- onlymacro

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

- linemacro

  Instructs the macro preprocessor include @#line directives specifying the line on which macro directives were encountered and expanded from. Only useful in conjunction with savemacro <savemacro[=FILENAME]>.

- onlymodel

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

- nolog

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

- output=second\|third

  Instructs the preprocessor to output derivatives of the dynamic model at least up to the given order.

- language=matlab\|julia

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

- params_derivs_order=0\|1\|2

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

- transform_unary_ops

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

- json = parse\|transform\|compute

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

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

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

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

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

- jsonstdout

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

- onlyjson

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

- jsonderivsimple

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

- warn_uninit

Display a warning for each variable or parameter which is not initialized. See (@ref "Parameter initialization"), or (@ref "loadparamsandsteadystate") for initialization of parameters. See (@ref "Initial and Terminal conditions"), or (@ref "loadparamsandsteadystate") for initialization of endogenous and exogenous variables.

- nopreprocessoroutput

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

- -DMACRO_VARIABLE\[=MACRO_EXPRESSION\]

  Defines a macro-variable from the command line (the same effect as using the Macro directive @#define in a model file, see (@ref "Macro processing language")). See the (@ref "note on quotes") for info on passing a MACRO_EXPRESSION argument containing spaces. Note that an expression passed on the command line can reference variables defined before it. If MACRO_EXPRESSION is omitted, the variable is assigned the true logical value.

  Example

  Call dynare with command line defines

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

- -I\<\<path\>\>

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

- nostrict

  Allows Dynare to issue a warning and continue processing when

1. there are more endogenous variables than equations.

2. an undeclared symbol is assigned in initval or endval.

3. an undeclared symbol is found in the model block in this case, it is automatically declared exogenous.
4. exogenous variables were declared but not used in the model block.

- stochastic

  Tells Dynare that the model to be solved is stochastic. If no Dynare commands related to stochastic models (stoch_simul, estimation, ...) are present in the .mod file, Dynare understands by default that the model to be solved is deterministic.

- exclude_eqs=\<\<equation_tags_to_exclude\>\>

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

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

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

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

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

1. One equation per line of the file, where every line represents the value passed to the name tag. e.g., a file such as: julia eq1 eq 2 would exclude equations with tags [name='eq1'] and [name='eq 2'].
2. One equation per line of the file, where every line after the first line represents the value passed to the tag specified by the first line. e.g., a file such as: julia tagname= a tag a tag with a, comma would exclude equations with tags [tagname='a tag'] and [tagname='a tag with a, comma']. Here note that the first line must end in an equal sign.

- include_eqs=\<\<equation_tags_to_include\>\>

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

- nocommutativity

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

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

4.2 Understanding Preprocessor Error Messages

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

The preprocessor issues error messages of the form:

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

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

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

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

In this case, the parser doesn't know a semicolon is missing at the end of the varexo command until it begins parsing the second line and bumps into the parameters command. This is because we allow commands to span multiple lines and, hence, the parser cannot know that the second line will not have a semicolon on it until it gets there. Once the parser begins parsing the second line, it realizes that it has encountered a keyword, parameters, which it did not expect. Hence, it throws an error of the form: ERROR: <<file.mod>>: line 2, cols 0-9: syntax error, unexpected PARAMETERS. In this case, you would simply place a semicolon at the end of line one and the parser would continue processing.

Part IV

Model File

Chapter 5

Syntax elements

5.1 Model File

Syntax elements

Conventions

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

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

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

Examples

```
// This is a single line comment

var x; // This is a comment about x

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

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

Note that these comment marks should not be used in native Julia code regions where the [#] should be preferred instead to introduce a comment. In a `verbatim` block, see `verbatim`, this would result in a crash since `//` is not a valid Julia statement).

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

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

- Optional arguments or options are indicated between square brackets: '[]';

- Repeated arguments are indicated by ellipses: "...";
- Mutually exclusive arguments are separated by vertical bars: '|';
- INTEGER indicates an integer number;
- INTEGER_VECTOR indicates a vector of integer numbers separated by spaces, enclosed by square brackets;
- DOUBLE indicates a double precision number. The following syntaxes are valid: `1.1e3`, `1.1E3`, `1.1d3`, `1.1D3`. In some places, infinite Values `Inf` and `-Inf` are also allowed;
- NUMERICAL_VECTOR indicates a vector of numbers separated by spaces, enclosed by square brackets;
- EXPRESSION indicates a mathematical expression valid outside the model description (see `expr`);
- MODEL_EXPRESSION (sometimes MODEL_EXP) indicates a mathematical expression valid in the model description (see `expr` and `model-decl`);
- MACRO_EXPRESSION designates an expression of the macro processor (see `macro-exp`);
- VARIABLE_NAME (sometimes VAR_NAME) indicates a variable name starting with an alphabetical character and can't contain: '()+-*/\^=!;:@#.' or accentuated characters;
- PARAMETER_NAME (sometimes PARAM_NAME) indicates a parameter name starting with an alphabetical character and can't contain: '()+-*/\^=!;:@#.' or accentuated characters;
- LATEX_NAME (sometimes TEX_NAME) indicates a valid LaTeX expression in math mode (not including the dollar signs);
- FUNCTION_NAME indicates a valid Julia function name;
- FILENAME indicates a filename valid in the underlying operating system; it is necessary to put it between quotes when specifying the extension or if the filename contains a non-alphanumeric character;
- QUOTED_STRING indicates an arbitrary string enclosed between (single) quotes. Note that Dynare commands call for single quotes around a string while in Julia strings are enclosed between double quotes.

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

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

The following special constants are also accepted in some contexts:

Constant: `inf`

Represents infinity.

Constant: nan

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

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.

Inside the model

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

Variables used in a `MODEL_EXPRESSION` denote current period values when neither a lead or a lag is given. A lead or a lag can be given by enclosing an integer between parenthesis just after the variable name: a positive integer means a lead, a negative one means a lag. Leads or lags of more than one period are allowed. For example, if c is an endogenous variable, then `c(+1)` is the variable one period ahead, and `c(-2)` is the variable two periods before.

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

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

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 `param-init`); for an endogenous or exogenous variable, it refers to the value given in the most recent `initval` or `endval` block.

Operators

The following operators are allowed in both MODEL_EXPRESSION and EXPRESSION:

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

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

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

Operator: **STEADYSTATE (MODELEXPRESSION)**

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

Exogenous and exogenous deterministic variables may not appear in `MODEL_EXPRESSION`.

Note

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

Operator: `EXPECTATION (INTEGER) (MODEL_EXPRESSION`

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

Functions

Built-in functions

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

Function: `exp(x)`

Natural exponential.

Function: `log(x)`

Function: `ln(x)`

Natural logarithm.

Function: `log10(x)`

Base 10 logarithm.

Function: `sqrt(x)`

Square root.

Function: `cbrt(x)`

Cube root.

Function: `sign(x)`

Signum function, defined as:

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

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

Function: `abs(x)`

Absolute value.

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

Function: `sin(x)`

Function: `cos(x)`

Function: `tan(x)`

Function: `asin(x)`

Function: `acos(x)`

Function: `atan(x)`

Trigonometric functions.

Function: `sinh(x)`

Function: `cosh(x)`

Function: `tanh(x)`

Function: `asinh(x)`

Function: `acosh(x)`

Function: `atanh(x)`

Hyperbolic functions.

Function: `max(a, b)`

Function: `min(a, b)`

Maximum and minimum of two reals.

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

Function: `normcdf(x)`

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

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

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

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

Function: `erf(x)`

Gauss error function.

Function: `erfc(x)`

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

A few words of warning in stochastic context

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

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

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

Footnotes

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

[2] Note that arbitrary Julia 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 Julia. Those constructions are not addresses in this section.

Chapter 6

Variables and parameters declaration

While Dynare allows the user to choose their own variable names, there are some restrictions to be kept in mind. First, variables and parameters must not have the same name as Dynare commands or built-in functions. In this respect, Dynare is not case-sensitive. For example, do not use `Ln` or `Sigma_e` to name your variable. Not conforming to this rule might yield hard-to-debug error messages or crashes.

var

command

```
var VAR_NAME [$TEX_NAME$][(long_name=QUOTED_STRINGNAME=QUOTED_STRING)]...;
```

```
var (deflator=MODEL_EXPR) VAR_NAME (... same options apply) var(log,deflator=MODEL_EXPR) VAR_NAME
↪  (... same options apply)
```

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

This required command declares the endogenous variables in the model. Optionally it is possible to give a LaTeX name to the variable or, if it is nonstationary, provide information regarding its deflator. The variables in the list can be separated by spaces or by commas. `var` commands can appear several times in the file and Dynare will concatenate them.

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

Options

- `log`

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

- `deflator = MODEL_EXPR`

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

- `log_deflator = MODEL_EXPR`

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

- `long_name = QUOTED_STRING`

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

Example

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

varexo

Command:

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

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

Options

- `long_name = QUOTED_STRING`

Example

```
    varexo m gov;
```

Remarks

````
An exogenous variable is an innovation, in the sense that this
variable cannot be predicted from the knowledge of the current
state of the economy. For instance, if logged TFP is a first order
autoregressive process:
```math
a_t=ρa_{t−1}+ε_t
```
````

then logged TFP is an endogenous variable to be declared with var, while the innovation `ε_t` is to be declared with varexo.

varexo_det

Command:

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

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

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

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

Options

- `long_name = QUOTED_STRING`
- `NAME = QUOTED_STRING`

Example

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

parameters

Command:

```
parameters PARAM_NAME [$TEX_NAME$] [(long_name=QUOTED_STRING|NAME=QUOTED_STRING)...];
```

```
This command declares parameters used in the model, in variable
initialization or in shocks declaration. Optionally it is possible to give a
LaTeX name to the parameter.
```

```
The parameters must subsequently be assigned values.

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

Options

- `long_name = QUOTED_STRING`
- `NAME = QUOTED_STRING`

Example

```
parameters alpha, bet;
```

Parameter initialization

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

The syntax is the following:

```
PARAMETER_NAME = EXPRESSION;
```

Here is an example of calibration:

```
parameters alpha, beta;

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

Internally, the parameter values are stored in `context.work.params`

Advanced commands

change_type

Command

```
change_type (var|varexo|varexo_det|parameters) VAR_NAME | PARAM_NAME...;
```

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

Example

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

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

var_remove

Command:

```
var_remove VAR_NAME | PARAM_NAME...;
```

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

predetermined_variables

Command:

```
predetermined_variables VAR_NAME...;
```

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

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

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

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

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

Example

The following two program snippets are strictly equivalent.

Using default Dynare timing convention:

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

Using the alternative timing convention:

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

trend_var

command:

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

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

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

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

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

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

Example

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

make_local_variables

command:

```
model_local_variable VARIABLE_NAME [LATEX_NAME]... ;
```

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

Example

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

On-the-fly Model Variable Declaration

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

To declare a variable on-the-fly in an equation tag, simply state the type of variable to be declared (endogenous, exogenous, or parameter followed by an equal sign and the variable name in single quotes. Hence, to declare a variable c as endogenous in an equation tag, you can type [endogenous='c'].

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

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

Example

The following two snippets are equivalent:

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

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

Chapter 7

Model declaration

Model declaration

The model is declared inside a `model` block:

Block: `model ;`

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

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

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

The syntax of equations must follow the conventions for `MODEL_EXPRESSION` as described in `expr`. 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 `var-decl`, but may have been declared by `model_local_variable`), an equal sign, and the expression for which this new variable will stand. Later on, every time this variable appears in the model, Dynare will substitute it by the expression assigned to the variable. Note that the scope of this variable is restricted to the model block; it cannot be used outside. To assign a LaTeX name to the model local variable, use the declaration syntax outlined by `model_local_variable`. A model local variable declaration looks like:

```
#VARIABLE_NAME = MODEL_EXPRESSION;
```

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

```
model;

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

end;
```

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

```
model;

[name='Taylor rule',mcp = 'r > -1.94478']
r = rho*r(-1) + (1-rho)*(gpi*Infl+gy*YGap) + e;

end;
```

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

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

Options

- `linear`

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

- `no_static`

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

- `differentiate_forward_vars` `differentiate_forward_vars = (VARIABLE_NAME [VARIABLE_NAME ...] )`

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

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

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

- `parallel_local_files = ( FILENAME [, FILENAME]... )`

Declares a list of extra files that should be transferred to follower nodes when doing a parallel computation (see `paral-conf`).

- `balanced_growth_test_tol = DOUBLE`

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

Example (Elementary RBC model)

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

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

Example (Use of model local variables)

The following program:

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

...is formally equivalent to:

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

Example (A linear model)

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

- `model_options (OPTIONS...);`

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

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

- `model_remove (TAGS...);`

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

The equations must be specified by a list of tag values, separated by commas. Each element of the list is either a simple quoted string, in which case it designates an equation by its `name` tag; or a tag name (without quotes), followed by an equal sign, then by the tag value (within quotes).

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

Example

```
var c k dummy1 dummy2;

model;
  c + k - aa*x*k(-1)^alph - (1-delt)*k(-1) + dummy1;
  c^(-gam) - (1+bet)^(-1)*(aa*alph*x(+1)*k^(alph-1) + 1 - delt)*c(+1)^(-gam);
  [ name = 'eq:dummy1', endogenous = 'dummy1' ]
  c*k = dummy1;
  [ foo = 'eq:dummy2' ]
  log(dummy2) = k + 2;
end;

  model_remove('eq:dummy1', foo = 'eq:dummy2');
```

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

- block: `model_replace (TAGS...);`

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

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

Example

```
var c k;

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

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

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

Auxiliary variables

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

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

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

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

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

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

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

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

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

The number of endogenous variables before the creation of auxiliary variables is stored in `context.models[1].orig_endo_nbr`, and the number of endogenous variables after the creation of auxiliary variables is stored in `context.models[1].endogenous_nbr`

Chapter 8

Steady state

There are three ways of computing the steady state (i.e. the static equilibrium) of a model. The first way is to provide the equations of the steady state in a `steady_state_model` block. When it is possible to derive the steady state by hand, this is the recommended way as it faster and more accurate.

The second way is to provide only a partial solution in the `steady_state_model` block and to compute the solution for the other variables numerically. Guess values for these other variables must be declared in a `initval` block. The less variables the better.

The third way is to compute the steady state value of all variables numerically. There is no `steady_state_model` block and a guess value must be declared for all variables. A guess value of 0 can be omitted, but be careful with variables appearing at the denominator of a fraction.

Providing the steady state to Dynare

If you know how to compute the steady state for your model, you can provide a `steady_state_model` block, which is described below in more details. The steady state file generated by Dynare will be called `+FILENAME/output/julia/FILEN`

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

Steady_state_model

Block: `steady\_state\_model ;`

When the analytical solution of the model is known, this command can be used to help Dynare find the steady state in a more efficient and reliable way, especially during estimation where the steady state has to be re-computed 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;
```

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

Example

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

parameters alp bet gam mst rho psi del;

...
// parameter calibration, (dynamic) model declaration, shock calibration...
...

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

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

Finding the steady state with Dynare nonlinear solver

Dynare commands

initval

- Block: `initval ;`

The `initval` block provides guess values for steady state computations.

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

```
VARIABLE_NAME = EXPRESSION;
```

endval

- Block: `endval ;`

The `endval` block can be used in a deterministic model to provide the guess values for computing a terminal steady state that is different from the initial steady state

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

```
VARIABLE_NAME = EXPRESSION;
```

steady

- Command: `steady ;`
- Command: `steady (OPTIONS...);`

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

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

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

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

Options

- `maxit = INTEGER`

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

- `tolf = DOUBLE`

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

- `tolx = DOUBLE`

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

- `homotopy_steps = INTEGER`

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

Output

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

Julia variable: `context.results.model_results[1].trends.endogenous_steady_state`

Contains the computed steady state. Endogenous variables are ordered in the order of declaration used in the `var` command,

Example

```
var c k;
varexo x;

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

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

steady;

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

steady;
```

Homotopy setup

Block: `homotopy_setup ;`

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

The idea of homotopy 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);
```

Julia function

steadystate!

`Dynare.steadystate!` – Function.

```
steadystate!(; context::Context=context, display::Bool = true,
        maxit::Int = 50, nocheck::Bool = false, tolf::Float64 = cbrt(eps()),
        tolx::Float64 = 0.0)
```

Keyword arguments

- `context::Context=context`: context in which the simulation is computed
- `homotopy_steps::Int=0`: number of homotopy steps
- `display::Bool=true`: whether to display the results
- `maxit::Int=50` maximum number of iterations
- `nocheck::Bool=false`: don't check the steady state
- `tolf::Float64=cbrt(eps())`: tolerance for the norm of residualts
- `tolx::Float64=0`: tolerance for the norm of the change in the result

source

Replace some equations during steady state computations

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

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

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

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

Example

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

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

Chapter 9

Shocks on exgogenous variables

To study the effects of a temporary shock after which the system goes back to the original equilibrium (if the model is stable...) one uses a temporary shock. 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 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. In Dynare this is done with `initval`, `endval` and `steady`.

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.

Dynare commands

shocks

- block: `shocks ;`
- block: `shocks(overwrite);`

Options

- `overwrite`: By default, if there are several `shocks` blocks

in the same `.mod` file, then they are cumulative: all the shocks declared in all the blocks are considered; however, if a `shocks` block is declared with the `overwrite` option, then it replaces all the previous `shocks` blocks.

In a deterministic context

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

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

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

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

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

Example 1

```
shocks;

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

end;
```

Example 2

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

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

In a stochastic context

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

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

- Specification of the standard error of an exogenous variable.

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

- Specification of the variance of an exogenous variable.

```
var VARIABLE_NAME = EXPRESSION;
```

- Specification the covariance of two exogenous variables.

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

- Specification of the correlation of two exogenous variables.

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

In an estimation context, it is also possible to specify variances and covariances on endogenous variables: in that case, these values are interpreted as the calibration of the measurement errors on these variables. This requires the `varobs` command to be specified before the `shocks` block.

Example

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

Remark

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.

In stochastic optimal policy context

When computing conditional welfare in a `ramsey_model` or `discretionary_policy` context, welfare is conditional on the state values inherited by planner when making choices in the first period. The information set of the first period includes the respective exogenous shock realizations. Thus, their known value can be specified using the perfect foresight syntax. Note that i) all other values specified for periods than period 1 will be ignored and ii) the value of lagged shocks (e.g. in the case of news shocks) is specified with `histval`.

Example

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

Mixing deterministic and stochastic shocks

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

Example

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

stoch_simul(irf=0);

forecast;
```

Julia function

scenario!()

The Julia function `scenario!()` lets you

- declare shocks on exogenous variables as the `shocks` block
- set the future value of endogenous variables (for conditional forecasts)
- add the date at which the above information is made available to the agents in the model

`Dynare.scenario!` – Function.

```
scenario!(; name=Symbol, period::PeriodSinceEpoch, value<:Number, context::Context=context,
  exogenous::Symbol=Symbol(), infoperiod::PeriodSinceEpoch=Undated(1))
```

Keyword arguments

- `name::Symbol`: the name of an endogenous or exogenous variable [required]
- `period::PeriodSinceEpoch`: the period in which the value is set
- `value<:PeriodSinceEpoch`: the value of the endogenous or exogenous variables
- context: the context is which the function operates (optional, default = context)
- exogenous: when an endogenous variable is set, the name of the exogenous that must be freed (required when an endogenous variables is set)
- infoperiod: the period in which the information is learned (optional, default = Undated(1))

source

Examples

```
scenario!(name = :e, value = 0.1, period = 2)
```

Exogenous variable e, takes value 0.1 in period 2.

```
scenario!(name = :y, value = 0.2, period=2, exogenous = :u)
```

Endogenous variable y is set to 0.2 in period 2 and exogenous variable u is treated as endogenous in the same period. Agents in the model know at the beginning of period 1 that this will happen.

```
scenario!(infoperiod = 2, name = :y, value = 0.2, period = 2,
                exogenous = :u)
```

Endogenous variable y is set to 0.2 in period 2 and exogenous variable u is treated as endogenous in the same period. Agents in the model only learn at the beginning of period 2 that this will happen.

Chapter 10

Deterministic simulations

When the framework is deterministic, Dynare can be used for models with the assumption of perfect foresight. The system is supposed to be in a given state before a period 1 (often a steady state) 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 equilibrium. This return to equilibrium is only an asymptotic phenomenon, which one must approximate by an horizon of simulation far enough in the future. Another exercise for which Dynare is well suited is to study the transition path to a new equilibrium following a permanent shock. For deterministic simulations, the numerical problem consists of solving a nonlinear system of simultaneous equations in n endogenous variables in T periods. Dynare uses a Newton-type method to solve the simultaneous equation system. Because the resulting Jacobian is in the order of n by T and hence will be very large for long simulations with many variables, Dynare makes use of the sparse matrix code .

Dynare commands

perfect_foresight_setup

Command: `perfect\_foresight\_setup;`

Command: `perfect\_foresight\_setup (OPTIONS...);`

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

This command must always be called before running the simulation with `perfect\_foresight\_solver`.

Options

- `periods = INTEGER`

Number of periods of the simulation.

- `datafile = FILENAME`

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

Output

The paths for the exogenous variables are stored into `context.results.model_resultst[1].simulations`.

The initial and terminal conditions for the endogenous variables and the initial guess for the path of endogenous variables are stored into `context.results.model_results[1].simulations`.

perfect_foresight_solver

Command: `perfect\_foresight\_solver ;`

Command: `perfect\_foresight\_solver (OPTIONS...);`

Computes the perfect foresight (or deterministic) simulation of the model.

Note that `perfect\_foresight\_setup` must be called before this command, in order to setup the environment for the simulation.

Options

- `maxit = INTEGER`

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

- `tolf = DOUBLE`

Convergence criterion for termination based on the function value. Iteration will cease when it proves impossible to improve the function value by more than `tolf`. Default: `1e-5`

- `tolx = DOUBLE`

Convergence criterion for termination based on the change in the function argument. Iteration will cease when the solver attempts to take a step that is smaller than `tolx`. Default: `1e-5`

- `noprint`

Don't print anything. Useful for loops.

- `print`

Print results (opposite of `noprint`).

- `lmmcp`

Solves mixed complementarity problems (the term refers to the LMMCP solver (Kanzow and Petra, 2004), that is used by DynareMatlab. DynareJulia uses the PATHSovler package)

- `endogenous_terminal_period`

The number of periods is not constant across Newton iterations when solving the perfect foresight model. The size of the nonlinear system of equations is reduced by removing the portion of the paths (and associated equations) for which the solution has already been identified (up to the tolerance parameter). This strategy can be interpreted as a mix of the shooting and relaxation approaches. Note that round off errors are more important with this mixed strategy (user should check the reported value of the maximum absolute error). Only available with option `stack_solve_algo==0`.

Remark

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

Example

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

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

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

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

will work.

Julia function

`Dynare.perfect_foresight!` – Function.

```
perfect_foresight!(; periods, context = context, display = true,
                     linear_solve_algo=ilu, maxit = 50, mcp = false,
                     tolf = 1e-5, tolx = 1e-5)
```

Keyword arguments

- `periods::Int`: number of periods in the simulation [required]
- `context::Context=context`: context in which the simulation is computed
- `display::Bool=true`: whether to display the results
- `linear_solve_algo::LinearSolveAlgo=ilu`: algorithm used for the solution of the linear problem. Either `ilu` or `pardiso`. `ilu` is the sparse linear solver used by default in Julia. To use the Pardiso solver, write `using Pardiso` before running Dynare.
- `maxit::Int=50` maximum number of iterations
- `mcp::Bool=falseL` whether to solve a mixed complementarity problem with occasionally binding constraints
- `tolf::Float64=1e-5`: tolerance for the norm of residualts
- `tolx::Float64=1e-5`: tolerance for the norm of the change in the result

source

Output

The simulated endogenous variables are available in `context.results.model_results[1].simulations`. This is a vector of `AxisArrayTable`, one for each simulations stored in `context`. Each `AxisArrayTable` contains the trajectories for endogenous and exogenous variables

Solving mixed complementarity problems

requires a particular model setup as the goal is to get rid of any min/max operators and complementary slackness conditions that might introduce a singularity into the Jacobian. This is done by attaching an equation tag (see `model-decl`) with the `mcp` keyword to affected equations. The format of the `mcp` tag is

```
[mcp = 'VARIABBLENAME OP CONSTANT']
```

where VARIABLENAME is an endogenous variable and OP is either > or <. For complicated occasionally binding constraints, it may be necessary to declare a new endogenous variable.

This tag states that the equation to which the tag is attached has to hold unless the expression within the tag is binding. For instance, a ZLB on the nominal interest rate would be specified as follows in the model block:

```
model;
   ...
   [mcp = 'r > -1.94478']
   r = rho*r(-1) + (1-rho)*(gpi*Infl+gy*YGap) + e;
   ...
end;
```

where `r` is the nominal interest rate in deviation from the steady state. This construct implies that the Taylor rule is operative, unless the implied interest rate `r<=-1.94478`, in which case the `r` is fixed at `-1.94478`. This is equavalant to

$$(r_t > -1.94478) \ \perp \ r_t = \rho r_{t-1} + (1-\rho)(g_\pi Infl_t + g_y YGap_t) + e_t$$

By restricting the value of `r` coming out of this equation, the `mcp` tag also avoids using `max(r,-1.94478)` for other occurrences of `r` in the rest of the model. It is important to keep in mind that, because the `mcp` tag effectively replaces a complementary slackness condition, it cannot be simply attached to any equation.

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

Chapter 11

Local approximation

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

The main algorithm for solving stochastic models relies on a Taylor approximation, up to second order, of the solution function (see Judd (1996), Collard and Juillard (2001a, 2001b), and Schmitt-Grohé and Uríbe (2004)). The details of the Dynare implementation of the first order solution are given in Villemot (2011). Such a solution is computed using the `stoch_simul` command.

Dynare commands

stoch_simul

Command: ‘stoch_simul;

Command: ‘stoch_simul (OPTIONS...);

Solves a stochastic (i.e. rational expectations) model, using perturbation techniques.

More precisely, `stoch_simul` computes a Taylor approximation of the model around the deterministic steady state and solves of the the decision and transition functions for the approximated model. Using this, it computes impulse response functions and various descriptive statistics (moments, variance decomposition, correlation and autocorrelation coefficients). For correlated shocks, the variance decomposition is computed as in the VAR literature through a Cholesky decomposition of the covariance matrix of the exogenous variables. When the shocks are correlated, the variance decomposition depends upon the order of the variables in the `varexo` command.

The IRFs are computed as the difference between the trajectory of a variable following a shock at the beginning of period 1 and its steady state value. More details on the computation of IRFs can be found at https://archives.dynare.org/DynareWiki/IrFs.

Variance decomposition, correlation, autocorrelation are only displayed for variables with strictly positive variance. Impulse response functions are only plotted for variables with response larger than 10^{-10}.

Variance decomposition is computed relative to the sum of the contribution of each shock. Normally, this is of course equal to aggregate variance, but if a model generates very large variances, it may happen that, due to numerical error, the two differ by a significant amount. Dynare issues a warning if the maximum relative difference between the sum of the contribution of each shock and aggregate variance is larger than `0.01%`.

The covariance matrix of the shocks is specified with the `shocks` command (see `shocks-exo`).

Options

- ar = INTEGER

Order of autocorrelation coefficients to compute. Default: 5

- `irf = INTEGER`

Number of periods on which to compute the IRFs. Setting `irf=0` suppresses the plotting of IRFs. Default: `40`.

- `nonstationary`: declares the model as nonstationary.
- `noprint`: don't print the results
- `order = INTEGER`

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

- `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 Julia variable `context.results.model_results[1].simulation`. Default: `0`.

- `dr = OPTION`

Determines the method used to compute the decision rule. Possible values for OPTION are:

`default`

Uses the default method to compute the decision rule based on the generalized Schur decomposition (see Villemot (2011) for more information).

`cycle_reduction`

Uses the cycle reduction algorithm to solve the polynomial equation for retrieving the coefficients associated to the endogenous variables in the decision rule. This method is faster than the default one for large scale models.

Default value is `default`.

Output

The derivatives of the approximated solution function are availabe in the vector of matrices `context.results.model_results[1]` The first element contains the matrix of first order derivatives. The second element, the matrix of second order derivatives.

The matrix of first order derivatives is a $nx(n_s + n_x + 1)$ matrix where n is the number of endogenous variables, n_s, the number of state variables (variables appearing in the model with a lag), and n_x, the number of exogenous variables. An element of this matrix is

$$
\begin{aligned}
X_{i,j} &= \frac{\partial g_i}{\partial y_j},\ \ j = 1, \ldots, n_s \\
X_{i,n_s+j} &= \frac{\partial g_i}{\partial x_j},\ \ j = 1, \ldots, n_x \\
X_{i,n_s+n_k+1} &= \frac{\partial g_i}{\partial \sigma} = 0
\end{aligned}
$$

where g_i is the solution function for variable `i`, y, the vector of endogenous variables, x, the vector en exogenous variables and σ the stochastic scale of the model. Note that at first order, this derivative is alwasy equal to zero.

The matrix of second order derivatives is $n \times n^2$ matrix where each column contains derivatives with respect to a pair of endogenous variables

- eigenvalues::Vector{Complex{Float64}}
- g1::Matrix{Float64} # full approximation
- gs1::Matrix{Float64} # state transition matrices: states x states
- hs1::Matrix{Float64} # states x shocks
- gns1::Matrix{Float64} # non states x states
- hns1::Matrix{Float64} # non states x shocsks
- g1_1::SubArray{Float64, 2, Matrix{Float64}, Tuple{Base.Slice{Base.OneTo{Int}}, UnitRange{Int}}, true} # solution first order derivatives w.r. to state variables
- g1_2::SubArray{Float64, 2, Matrix{Float64}, Tuple{Base.Slice{Base.OneTo{Int}}, UnitRange{Int}}, true} # solution first order derivatives w.r. to current exogenous variables
- endogenous_variance::Matrix{Float64}
- stationary_variables::Vector{Bool}

Example 1

```
shocks;
var e;
stderr 0.0348;
end;

stoch_simul;
```

Performs the simulation of the 1st-order approximation of a model with a single stochastic shock `e`, with a standard error of `0.0348`.

Example 2

```
stoch_simul(irf=60);
```

Performs the simulation of a model and displays impulse response functions on 60 periods.

Julia function

`Dynare.localapproximation!` – Function.

```
localapproximation!(; context::Context=context, display = true,
                    dr_algo::String = "GS", irf::Int = 40,
                    LRE_options = LinearRationalExpectationsOptions(),
                    nar::Int = 5, nonstationary::Bool = false,
                    order::Int = 1, periods::Int = 0 )
```

computes a local approximation of a model contained in `context`

Keyword arguments

- `context::Context=context`: context in which the simulation is computed
- `display::Bool=true`: whether to display the results
- `dr_algo::String`: solution algorithm, either "GS" for generalized Schur decomposition (default) or "CR" for cyclic reduction
- `irf::Int = 40`: number of periods for IRFs. Use 0 for no IRF computation
- `LRE_options::LinearRationalExpectationsOptions = LinearRationalExpectationsOptions()`: options passed to the LinearRationalExpectation package
- `nar::Int = 5`: numnber of periods for autocorrelations. Use 0 for no autocorrelation computation
- `nonstationary::Bool = false`: to specify a nonstationary model
- `periods::Int = 0`: number of periods for an optional Monte Carlo simulation of the model

source

First-order approximation

The approximation has the stylized form:

$$y_t = y^s + A\phi(y_{t-1}) + Bu_t$$

where y^s is the steady state value of y and $\phi(y_{t-1}) = y_{t-1} - y^s$. Matrices of coefficients A and B are computed by Dynare.

Second-order approximation

The approximation has the form:

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

where y^s is the steady state value of y, $\phi(y_{t-1}) = y_{t-1} - y^s$, and Δ^2 is the shift effect of the variance of future shocks. Matrices of coefficients A, B, C, D and E are computed by Dynare.

Chapter 12

State space, filtering and smoothing

From a statistical point of view, DSGE models are unobserved components models: only a few variables are observed. Filtering or smoothing provide estimate of the unobserved variables given the observations.

The model is put in state space form

$$y^o_t = Ms_t + N\epsilon_t$$
$$s_t = Ts_{t-1} + R\eta_t$$

where y^o_t represents observed variable at period `t`. The coefficient matrices of the transition equation, T and R are provided by the solution of the linear(-isze) rational expectation model. ϵ_t are possible measurement errors and η_t the structural shocks. Most often matrix `M` is a selection matrix.

Filtering provides estimates conditional only on past observations:

$$\mathbb{E}(y^{no}_t|Y^o_{t-1})$$

where y^{no}_t are unobserved variables at period `t` and Y^o_{t-1} represent the set observations until period `t-1` included.

Smoothing provides estimates of unobserved variables conditional on the entire sample of observations:

$$\mathbb{E}(y^{no}_t|Y^o_T)$$

where Y^o_T represents the all observations in the sample.

Dynare command

varobs

Observed variables are declared with the `varobs` command

Command: `varobs VARIABLE_NAME...;`

This command lists the name of observed endogenous variables for the estimation procedure. These variables must be available in the data file (see `estimation_cmd <estim-comm>`).

Alternatively, this command is also used in conjunction with the `partial_information` option of `stoch_simul`, for declaring the set of observed variables when solving the model under partial information.

Only one instance of `varobs` is allowed in a model file. If one needs to declare observed variables in a loop, the macro processor can be used as shown in the second example below.

Example

```
varobs C y rr;
```

observation_trends

It is possible to declare a deterministic linear trend that is removed for the computations and added back in the results

Block: `observation_trends ;`

This block specifies linear trends for observed variables as functions of model parameters. In case the `loglinear` option is used, this corresponds to a linear trend in the logged observables, i.e. an exponential trend in the level of the observables.

Each line inside of the block should be of the form:

```
VARIABLE_NAME(EXPRESSION);
```

In most cases, variables shouldn't be centered when `observation_trends` is used.

Example

```
observation_trends;
Y (eta);
P (mu/eta);
end;
```

calib_smoother

This command triggers the computation of the filter and smoother for calibrated models

Command: `calib_smoother [VARIABLE_NAME]...;`

Command: `calib_smoother (OPTIONS...)[VARIABLE_NAME]...;`

This command computes the smoothed variables (and possible the filtered variables) on a calibrated model.

A datafile must be provided, and the observable variables declared with `varobs`. The smoother is based on a first-order approximation of the model.

By default, the command computes the smoothed variables and shocks and stores the results in `oo_.SmoothedVariables` and `oo_.SmoothedShocks`. It also fills `oo_.UpdatedVariables`.

Options

- `datafile = FILENAME`: file containing the observation in CSV format.
- `filtered_vars`: triggers the computation of filtered variables.
- `first_obs = INTEGER`: first observation
- `diffuse_filter = INTEGER`: use a diffuse filter for nonstationary models.

Julia functions

`Dynare.calibsmoother!` – Function.

```
calibsmoother!(; context::Context=context,
                 datafile::String = "",
                     data::AxisArrayTable = AxisArrayTable(Matrix{Float64}(undef, 0, 0),
                     ↪  Undated[], Symbol[]),
                 first_obs::PeriodSinceEpoch = Undated(typemin(Int)),
                 last_obs::PeriodSinceEpoch = Undated(typemin(Int)),
                 nobs::Int = 0
               )
```

Compute the smoothed values of the variables for an estimated model

#Keyword arguments

- `periods::Integer`: number of forecasted periods [required]
- `datafile::String`: file with the observations for the smoother
- `data::AxisArrayTable`: AxisArrayTable containing observed variables (can't be used at the same time as `datafile`)
- `first_obs::PeriodSinceEpoch`: first period used by smoother (default: first observation in the dataset)
- `last_obs::PeriodSinceEpoch`: last period used by smoother (default: last observation in the dataset)
- `nobs::Int`: number of observations (default: entire dataset)

source

Chapter 13

Estimation

Provided that you have observations on some endogenous variables, it is possible to use Dynare to estimate some or all parameters. Bayesian techniques (as in Fernández-Villaverde and Rubio-Ramírez (2004), Rabanal and Rubio-Ramirez (2003), Schorfheide (2000) or Smets and Wouters (2003)) are available. Using Bayesian methods, it is possible to estimate DSGE models.

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.

Before using the estimation commands described below, you need to define some elements of the state space representation of the model. At the minimum, you need to declare the observed variables with `var_obs` and, possibly, deterministic trends with `observation_trends` (see the previous section: `State space, filtering and smoothing`)

Dynare commands

estimated_params

Block: `estimated_params ;`

Block: `estimated_params (overwrite) ;`

This block lists all parameters to be estimated and specifies bounds and priors as necessary.

Each line corresponds to an estimated parameter.

In a maximum likelihood or a method of moments estimation, each line follows this syntax:

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

In a Bayesian MCMC or a penalized method of moments estimation, each line follows this syntax:

```
stderr VARIABLE_NAME | corr VARIABLE_NAME_1, VARIABLE_NAME_2 | PARAMETER_NAME |
↪  DSGE_PRIOR_WEIGHT
[, INITIAL_VALUE [, LOWER_BOUND, UPPER_BOUND]], PRIOR_SHAPE,
PRIOR_MEAN, PRIOR_STANDARD_ERROR [, PRIOR_3RD_PARAMETER [,
PRIOR_4TH_PARAMETER [, SCALE_PARAMETER ] ] ];
```

The first part of the line consists of one of the four following alternatives:

- `stderr VARIABLE_NAME`

  Indicates that the standard error of the exogenous variable VARIABLENAME, or of the observation error/measurement errors associated with endogenous observed variable VARIABLENAME, is to be estimated.

- `corr VARIABLE_NAME1, VARIABLE_NAME2`

  Indicates that the correlation between the exogenous variables VARIABLENAME1 and VARIABLENAME2, or the correlation of the observation errors/measurement errors associated with endogenous observed variables VARIABLENAME1 and VARIABLENAME2, is to be estimated. Note that correlations set by previous shocks-blocks or estimation-commands are kept at their value set prior to estimation if they are not estimated again subsequently. Thus, the treatment is the same as in the case of deep parameters set during model calibration and not estimated.

- `PARAMETER_NAME`

  The name of a model parameter to be estimated

- `DSGE_PRIOR_WEIGHT`

  Special name for the weigh of the DSGE model in DSGE-VAR model.

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

- `INITIAL_VALUE`

Specifies a starting value for the posterior mode optimizer or the maximum likelihood estimation. If unset, defaults to the prior mean.

- `LOWER_BOUND`

Currently not supported

- `UPPER_BOUND`

Currently not supported

- `PRIOR_SHAPE`

A keyword specifying the shape of the prior density. The possible values are: `beta_pdf`, `gamma_pdf`, `normal_pdf`, `uniform_pdf`, `inv_gamma_pdf`, `inv_gamma1_pdf`, `inv_gamma2_pdf` and `weibull_pdf`. Note that `inv_gamma_pdf` is equivalent to `inv_gamma1_pdf`.

- `PRIOR_MEAN`

The mean of the prior distribution.

- `PRIOR_STANDARD_ERROR`

The standard error of the prior distribution.

- `PRIOR_3RD_PARAMETER`

Currently not supported except for the Uniform

- `PRIOR_4TH_PARAMETER`

Currently not supported

- `SCALE_PARAMETER`

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

Note that INITIALVALUE, LOWERBOUND, UPPERBOUND, PRIORMEAN, PRIORSTANDARDERROR, PRIOR3RDPARAMETER, PRIOR4THPARAMETER and SCALE_PARAMETER can be any valid EXPRESSION. Some of them can be empty, in which Dynare will select a default value depending on the context and the prior shape.

In case of the uniform distribution, it can be specified either by providing an upper and a lower bound using `PRIOR_3RD_PARAMETER` and `PRIOR_4TH_PARAMETER` or via mean and standard deviation using `PRIOR_MEAN`, `PRIOR_STANDARD_ERROR`. The other two will automatically be filled out. Note that providing both sets of hyper-parameters will yield an error message.

As one uses options more towards the end of the list, all previous options must be filled: for example, if you want to specify SCALEPARAMETER, you must specify ‘PRIOR3RDPARAMETERandPRIOR4TH_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.

Example

```
parameters bet;

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

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

It is possible to have several `estimated_params` blocks. By default, subsequent blocks are concatenated with the previous ones; this can be useful when building models in a modular fashion (see also `estimated_params_remove` for that use case). However, if an `estimated_params` block has the `overwrite` option, its contents becomes the new list of estimated parameters, cancelling previous blocks; this can be useful when doing several estimations in a single `.mod` file.

estimated_params_init

Block: `estimated_params_init ;`

Block: `estimated_params_init (OPTIONS...);`

This block declares numerical initial values for the optimizer when these ones are different from the prior mean. It should be specified after the `estimated_params` block as otherwise the specified starting values are overwritten by the latter.

Each line has the following syntax:

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

estimation

Command: `estimation [VARIABLE_NAME...];`

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

This command runs Bayesian estimation.

Options

- `datafile = FILENAME`

The datafile must be in CSV format

```
estimation(datafile='../fsdat_simul.csv',...);
```

- `nobs = INTEGER`

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

- `first_obs = INTEGER`

The number of the first observation to be used. In case of estimating a DSGE-VAR, `first_obs` needs to be larger than the number of lags. Default: `1`.

- `plot_priors = INTEGER`: Control the plotting of priors, `0`, no prior plot, `1`, pPrior density for each estimated parameter is plotted. It is important to check that the actual shape of prior densities matches what you have in mind. Ill-chosen values for the prior standard density can result in absurd prior densities (default valueL `1`).
- `mh_replic = INTEGER`

Number of replications for each chain of the Metropolis-Hastings algorithm. The number of draws should be sufficient to achieve convergence of the MCMC and to meaningfully compute posterior objects. Default: `20000`.

- `mh_nblocks = INTEGER`

Number of parallel chains for Metropolis-Hastings algorithm. Default: 2.

- `mh_jscale = DOUBLE`

The scale parameter of the jumping distribution's covariance matrix. The default value is rarely satisfactory. This option must be tuned to obtain, ideally, an acceptance ratio of 25%-33%. Basically, the idea is to increase the variance of the jumping distribution if the acceptance ratio is too high, and decrease the same variance if the acceptance ratio is too low. In some situations it may help to consider parameter-specific values for this scale parameter. This can be done in the `estimated_params` block. Default: `0.2`.

Julia functions

`Dynare.covariance` – Function.

covariance(chain:Chains)

computes the covariance matrix of MCMC chain

source

`Dynare.mode_compute!` – Function.

```
mode_compute!(;
            context=context,
            data = AxisArrayTable(AxisArrayTables.AxisArray(Matrix(undef, 0, 0))),
            datafile = "",
            diffuse_filter::Bool = false,
            display::Bool = false,
            fast_kalman_filter::Bool = true,
            first_obs::PeriodsSinceEpoch = Undated(typemin(Int)),
            initial_values = get_initial_value_or_mean(),
            last_obs::PeriodsSinceEpoch = Undated(typemin(Int)),
            mode_check::Bool = false,
            nobs::Int = 0,
            order::Int = 1,
            presample::Int = 0,
            transformed_parameters = true)
```

computes the posterior mode.

Keyword arguments

- `context::Context=context`: context of the computation
- `data::AxisArrayTable`: AxisArrayTable containing observed variables
- `datafile::String: data filename (can't be used as the same time asdataset`')
- `first_obs::PeriodsSinceEpoch`: first observation (default: first observation in the dataset)
- `initial_values`: initival parameter values for optimization algorithm (default: `estimated_params_init` block if present or prior mean)
- `last_obs::PeriodsSinceEpoch`: last period (default: last period of the dataset)
- `nobs::Int = 0`: number of observations (default: entire dataset)

- `transformed_parameters = true`: whether to transform estimated parameter so as they take their value on R

Either `data` or `datafile` must be specified.

source

`Dynare.plot_priors` – Function.

```
plot_priors(; context::Context = context, n_points::Int = 100)
```

plots prior density

Keyword arguments

- `context::Context = context`: context in which to take the date to be ploted
- `n_points::Int = 100`: number of points used for a curve

source

`Dynare.plot_prior_posterior` – Function.

```
plot_prior_posterior(chains; context::Context=context)
```

plots priors posterios and mode if computed on the same plots

Keyword arguments

- `context::Context=context`: context used to get the estimation results

Output

- the plots are saved in `./<modfilename>/Graphs/PriorPosterior_<x>.png`

source

`Dynare.prior!` – Function.

```
prior!(s::Symbol; shape::{<:Distributions}, initialvalue::Union{Real,Missing}=missing,
↪  mean::Union{Real,Missing}=missing, stdev::Union{Real,Missing}=missing, domain=[], variance
↪  ::Union{Real,Missing}=missing, context::Context=context)
prior!(s::stdev; shape::{<:Distributions}, initialvalue::Union{Real,Missing}=missing,
↪  mean::Union{Real,Missing}=missing, stdev::Union{Real,Missing}=missing, domain=[], variance
↪  ::Union{Real,Missing}=missing, context::Context=context)
prior!(s::variance; shape::{<:Distributions}, initialvalue::Union{Real,Missing}=missing,
↪  mean::Union{Real,Missing}=missing, stdev::Union{Real,Missing}=missing, domain=[], variance
↪  ::Union{Real,Missing}=missing, context::Context=context)
prior!(s::corr; shape::{<:Distributions}, initialvalue::Union{Real,Missing}=missing,
↪  mean::Union{Real,Missing}=missing, stdev::Union{Real,Missing}=missing, domain=[], variance
↪  ::Union{Real,Missing}=missing, context::Context=context)
```

generates a prior for a symbol of a parameter, the standard deviation (`stdev`) or the variance (`variance`) of an exogenous variable or an endogenous variable (measurement error) or the correlation (`corr`) between 2 endogenous or exogenous variables

Keyword arguments

- `shape <: Distributions`: the shape of the prior distribution (`Beta`, `InvertedGamma`, `InvertedGamma1`, `Gamma`, `Normal`, `Uniform`, `Weibull`) [required]
- `context::Context=context`: context in which the prior is declared
- `domain::Vector{<:Real}=Float64[]`: domain for a uniform distribution
- `initialvalue::Union{Real,Missing}=missing`: initialvalue for mode finding or MCMC iterations
- `mean::Union{Real,Missing}=missing`: mean of the prior distribution
- `stdev::Union{Real,Missing}=missing`: stdev of the prior distribution
- `variance::Union{Real,Missing}=missing`: variance of the prior distribution

source

`Dynare.rwmh_compute!` – Function.

```
rwmh_compute!(;context::Context=context,
         back_transformation::Bool = true,
         datafile::String = "",
         data::AxisArrayTable = AxisArrayTable(AxisArrayTables.AxisArray(Matrix(undef, 0, 0))),
         diffuse_filter::Bool = false,
         display::Bool = true,
         fast_kalman_filter::Bool = true,
         first_obs::PeriodsSinceEpoch = Undated(typemin(Int)),
         initial_values = prior_mean(context.work.estimated_parameters),
         covariance::AbstractMatrix{Float64} =
         ↪  Matrix(prior_variance(context.work.estimated_parameters)),
         transformed_covariance::Matrix{Float64} = Matrix{Float64}(undef, 0,0),
         last_obs::PeriodsSinceEpoch = Undated(typemin(Int)),
         mcmc_chains::Int = 1,
         mcmc_init_scale::Float64 = 0.0,
         mcmc_jscale::Float64 = 0.0,
         mcmc_replic::Int =  0,
         mode_compute::Bool = true,
         nobs::Int = 0,
         order::Int = 1,
         plot_chain::Bool = true,
         plot_posterior_density::Bool = false,
         presample::Int = 0,
         transformed_parameters::Bool = true
```

)

runs random walk Monte Carlo simulations of the posterior

Keyword arguments

- `context::Context=context`: context of the computation
- `covariance::AbstractMatrix{Float64}`:

- `data::AxisArrayTable`: AxisArrayTable containing observed variables
- `datafile::String`: data filename
- `first_obs::PeriodsSinceEpoch`: first observation (default: first observation in the dataset)
- `initial_values`: initival parameter values for optimization algorithm (default: `estimated_params_init` block if present or prior mean)
- `last_obs::PeriodsSinceEpoch`: last period (default: last observation in the dataset)
- `mcmc_chains::Int` number of MCMC chains (default: 1)
- `mcmc_jscale::Float64`: scale factor of proposal
- `mcmc_replic::Int`: = 0,
- `nobs::Int = 0`: number of observations (default: entire dataset)
- `plot_chain::Bool`: whether to display standard MCMC chain output (default:true)
- `plot_posterior_density::Bool`: wether to display plots with prior and posterior densities (default: false)
- `transformed_covariance::Matrix{Float64}`: covariance of transformed parameters (default: empty)
- `transformed_parameters = true`: whether to transform estimated parameter so as they take their value on R

Either `data` or `datafile` must be specified.

source

Chapter 14

Forecasting

Julia functions

`Dynare.forecasting!` – Function.

```
forecasting!(; periods::Integer,
               forecast_mode::ForecastModes,
               context::Context=context,
               datafile::String="",
               first_obs::PeriodsSinceEpoch=Undated(typemin(Int)),
               first_period::PeriodsSinceEpoch=Undated(0),
               last_obs::PeriodsSinceEpoch=Undated(typemin(Int)),
               order::Integer=1)
```

computes an unconditional forecast of the variables of the model

Keyword arguments

- `periods::Integer`: number of forecasted periods [required]
- `forecast_mode::ForecastModes`: one of `histval` or `calibsmoother` [required]
- `datafile::String`: file with the observations for the smoother
- `first_obs::PeriodsSinceEpoch`: first period used by smoother (default: first observation in the file)
- `first_period::PeriodsSinceEpoch`: initial_period for the forecast (default when histval: Undated(0), default when calibsmoother: last period of the smoother)
- `last_obs::PeriodsSinceEpoch`: last period used by smoother (default: last observation in the file)
- `order::Integer`: order of local approximation

source

`Dynare.recursive_forecasting!` – Function.

```
recursive_forecasting!(; Np::Integer,
                         first_period::PeriodsSinceEpoch=Undated(typemin(Int)),
                         last_period::PeriodsSinceEpoch=Undated(typemax(Int)),
                         context::Context=context,
```

```
                      datafile::String="",
                      first_obs::PeriodsSinceEpoch=Undated(typemin(Int)),
                      last_obs::PeriodsSinceEpoch=Undated(typemax(Int)),
                      order::Integer=1)
```

computes an unconditional recursive forecast by adding one period to the sample used for the smoother before forecasting over `Np` periods.

Keyword arguments

- `Np::Integer`: number of forecasted periods [required]
- `first_period::PeriodsSinceEpoch`: initial period of first forecast [required]
- `last_period::PeriodsSinceEpoch`: initial period of last forecast [required]
- `datafile::String`: file with the observations for the smoother
- `first_obs::PeriodsSinceEpoch`: first period used by smoother (default: first observation in the file)
- `last_obs::PeriodsSinceEpoch`: last period used by smoother (default: last observation in the file)
- `order::Integer`: order of local approximation

source

Chapter 15

Reporting

Dynare can generate PDF reports using \LaTeX

- A report is made of
  - a title
  - a subtitle (optional)
  - pages
- A page is made of sections
- A section can be
  - a text paragraph
  - a listing of the model
  - a table
  - a graphic

Julia functions

`Dynare.Report` – Type.

Report(title::String; subtitle::String = "")

initialize empty report

Keyword arguments

- `title::String`: Report title [required]
- `subtitle::String`: Report subtitle

`source`

`Dynare.add_page!` – Function.

add_page!(report::Report, page::Page)

adds a page to a report

Keyword arguments

- `report::Report`: report
- `page::Page`: page to be added

source

`Dynare.add_graph!` – Function.

add_graph!(page::Page, graph::Graph)

adds a graph to a page

Keyword arguments

- `page::Page`: page
- `graph::Graph`: graph to be added

source

`Dynare.add_model!` – Function.

add_model!(page::Page; context::Context = context, lastline::Int = 0, format = 1) adds the lines of a *.mod file to pages

Keyword arguments

- `page::Page`: page [required]
- `context::Context`: context corresponding to the *.mod file (default: context)
- `lastline::Int`: last line to be printed
- `format::Int`: how to display parameter values 1: value is written after the parameter name 2: value is written below the parameter name

source

`Dynare.add_paragraph!` – Function.

add_paragraph!(page::Page, paragraph::String)

adds a graph to a page

Keyword arguments

- `page::Page`: page
- `paragraph::String`: paragraph to be added

source

@docs add_table! ““

‘$@docsprint$

Part V

Macroprocessing language

15.1 Macro processing language

It is possible to use "macro" commands in the `.mod` file for performing tasks such as: including modular source files, replicating blocks of equations through loops, conditionally executing some code, writing indexed sums or products inside equations...

The Dynare macro-language provides a new set of macro-commands which can be used in `.mod` files. It features:

- File inclusion
- Loops (`for` structure)
- Conditional inclusion (`if/then/else` structures)
- Expression substitution

This macro-language is totally independent of the basic Dynare language, and is processed by a separate component of the Dynare pre-processor. The macro processor transforms a `.mod` file with macros into a `.mod` file without macros (doing expansions/inclusions), and then feeds it to the Dynare parser. The key point to understand is that the macro processor only does text substitution (like the C preprocessor or the PHP language). Note that it is possible to see the output of the macro processor by using the `savemacro` option of the `dynare` command (see `dyn-invoc`).

The macro processor is invoked by placing macro directives in the `.mod` file. Directives begin with an at-sign followed by a pound sign (`@#`). They produce no output, but give instructions to the macro processor. In most cases, directives occupy exactly one line of text. If needed, two backslashes (`\\`) at the end of the line indicate that the directive is continued on the next line. Macro directives following `//` are not interpreted by the macro processor. For historical reasons, directives in commented blocks, ie surrounded by `/*` and `*/`, are interpreted by the macro processor. The user should not rely on this behavior. The main directives are:

- `@#includepath`, paths to search for files that are to be included,
- `@#include`, for file inclusion,
- `@#define`, for defining a macro processor variable,
- `@#if, @#ifdef, @#ifndef, @#elseif, @#else, @#endif` for conditional statements,
- `@#for, @#endfor` for constructing loops.

The macro processor maintains its own list of variables (distinct from model variables and Julia variables). These macro-variables are assigned using the `@#define` directive and can be of the following basic types: boolean, real, string, tuple, function, and array (of any of the previous types).

Macro expressions

Macro-expressions can be used in two places:

- Inside macro directives, directly;
- In the body of the `.mod` file, between an at-sign and curly braces: the macro processor will substitute the expression with its value

It is possible to construct macro-expressions that can be assigned to macro-variables or used within a macro-directive. The expressions are constructed using literals of the basic types (boolean, real, string, tuple, array), comprehensions, macro-variables, macro-functions, and standard operators.

Note

Elsewhere in the manual, MACRO_EXPRESSION designates an expression constructed as explained in this section.

Boolean

The following operators can be used on booleans:

- Comparison operators: `==, !=`
- Logical operators: `&&, ||, !`

Real

The following operators can be used on reals:

- Arithmetic operators: `+, -, *, /, ^`
- Comparison operators: `<, >, <=, >=, ==, !=`
- Logical operators: `&&, ||, !`
- Ranges with an increment of 1: `REAL1:REAL2` (for example, `1:4` is equivalent to real array `[1, 2, 3, 4]`).

  4.6 Previously, putting brackets around the arguments to the colon operator (e.g. `[1:4]`) had no effect. Now, `[1:4]` will create an array containing an array (i.e. `[ [1, 2, 3, 4] ]`).
- Ranges with user-defined increment: `REAL1:REAL2:REAL3` (for example, `6:-2.1:-1` is equivalent to real array `[6, 3.9, 1.8, -0.3]`).
- Functions: `max, min, mod, exp, log, log10, sin, cos, tan, asin, acos, atan, sqrt, cbrt, sign, floor, ceil, trunc, erf, erfc, gamma, lgamma, round, normpdf, normcdf`. NB `ln` can be used instead of `log`

String

String literals have to be enclosed by **double** quotes (like `"name"`).

The following operators can be used on strings:

- Comparison operators: `<, >, <=, >=, ==, !=`
- Concatenation of two strings: `+`
- Extraction of substrings: if `s` is a string, then `s[3]` is a string containing only the third character of `s`, and `s[4:6]` contains the characters from 4th to 6th
- Function: `length`

Tuple

Tuples are enclosed by parenthesis and elements separated by commas (like `(a,b,c)` or `(1,2,3)`).

The following operators can be used on tuples:

- Comparison operators: `==, !=`
- Functions: `empty, length`

Array

Arrays are enclosed by brackets, and their elements are separated by commas (like `[1,[2,3],4]` or `["US", "FR"]`).

The following operators can be used on arrays:

- Comparison operators: `==, !=`
- Dereferencing: if v is an array, then `v[2]` is its 2nd element
- Concatenation of two arrays: `+`
- Set union of two arrays: `|`
- Set intersection of two arrays: `&`
- Difference `-`: returns the first operand from which the elements of the second operand have been removed.
- Cartesian product of two arrays: `*`
- Cartesian product of one array N times: `^N`
- Extraction of sub-arrays: e.g. `v[4:6]`
- Testing membership of an array: `in` operator (for example: `"b"` in `["a", "b", "c"]` returns 1)
- Functions: `empty, sum, length`

Comprehension

Comprehension syntax is a shorthand way to make arrays from other arrays. There are three different ways the comprehension syntax can be employed: [filtering], [mapping], and [filtering and mapping].

Filtering

Filtering allows one to choose those elements from an array for which a certain condition hold.

Example

Create a new array, choosing the even numbers from the array `1:5`:

```
[ i in 1:5 when mod(i,2) == 0 ]
```

would result in:

```
[2, 4]
```

Mapping

Mapping allows you to apply a transformation to every element of an array.

Example

Create a new array, squaring all elements of the array `1:5`:

```
[ i^2 for i in 1:5 ]
```

would result in:

```
 [1, 4, 9, 16, 25]
```

Filtering and Mapping

Combining the two preceding ideas would allow one to apply a transformation to every selected element of an array.

Example

Create a new array, squaring all even elements of the array `1:5`:

```
[ i^2 for i in 1:5 when mod(i,2) == 0]
```

would result in:

```
 [4, 16]
```

Further Examples :

```
[ (j, i+1) for (i,j) in (1:2)^2 ]
[ (j, i+1) for (i,j) in (1:2)*(1:2) when i < j ]
```

would result in:

```
  [(1, 2), (2, 2), (1, 3), (2, 3)]
  [(2, 2)]
```

Function

Functions can be defined in the macro processor using the `@#define` directive (see below). A function is evaluated at the time it is invoked, not at define time. Functions can be included in expressions and the operators that can be combined with them depend on their return type.

Checking variable type

Given a variable name or literal, you can check the type it evaluates to using the following functions: `isboolean`, `isreal`, `isstring`, `istuple`, and `isarray`.

Code	Output
`isboolean(0)`	`false`
`isboolean(true)`	`true`
`isreal("str")`	`false`

Examples

Casting between types

Variables and literals of one type can be cast into another type. Some type changes are straightforward (e.g. changing a [real]{.title-ref} to a [string]{.title-ref}) whereas others have certain requirements (e.g. to cast an [array]{.title-ref} to a [real]{.title-ref} it must be a one element array containing a type that can be cast to [real]{.title-ref}).

Examples

Code	Output
`(bool) -1.1`	`true`
`(bool) 0`	`false`
`(real) "2.2"`	`2.2`
`(tuple) [3.3]`	`(3.3)`
`(array) 4.4`	`[4.4]`
`(real) [5.5]`	`5.5`
`(real) [6.6, 7.7]`	`error`
`(real) "8.8 in a string"`	`error`

Casts can be used in expressions:

Examples

Code	Output
`(bool) 0 && true`	`false`
`(real) "1" + 2`	`3`
`(string) (3 + 4)`	`"7"`
`(array) 5 + (array) 6`	`[5, 6]`

Macro directives

Macro Directive: `@#includepath "PATH"`

Macro Directive `@#includepath MACRO_EXPRESSION`

This directive adds the path contained in PATH to the list of those to search when looking for a `.mod` file specified by `@#include`. If provided with a MACRO_EXPRESSION argument, the argument must evaluate to a string. Note that these paths are added after any paths passed using `-I <-I\<\<path\>\>>`{.interpreted-text role="opt"}.

Example

```
@#includepath "/path/to/folder/containing/modfiles"
@#includepath folders_containing_mod_files
```

Macro Directive: `@#include "FILENAME"`

Macro Directive: `@#include MACRO_EXPRESSION`

This directive simply includes the content of another file in its place; it is exactly equivalent to a copy/paste of the content of the included file. If provided with a MACRO_EXPRESSION argument, the argument must evaluate to a string. Note that it is possible to nest includes (i.e. to include a file from an included file). The file will be searched for in the current directory. If it is not found, the file will be searched for in the folders provided by `-I <-I\<\<path\>\>>`{.interpreted-text role="opt"} and `@#includepath`.

Example

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

Macro Directive: `@#define MACRO_VARIABLE`

Macro Directive: `@#define MACRO_VARIABLE = MACRO_EXPRESSION`

Macro Directive: `@#define MACRO_FUNCTION = MACRO_EXPRESSION`

Defines a macro-variable or macro function.

Example

```
@#define var                      // Equals true
@#define x = 5                    // Real
@#define y = "US"                 // String
@#define v = [ 1, 2, 4 ]          // Real array
@#define w = [ "US", "EA" ]       // String array
@#define u = [ 1, ["EA"] ]        // Mixed array
@#define z = 3 + v[2]             // Equals 5
@#define t = ("US" in w)          // Equals true
@#define f(x) = " " + x + y       // Function `f` with argument `x`
                                  // returns the string ' ' + x + 'US'
```

Example

```
@#define x = 1
@#define y = [ "B", "C" ]
@#define i = 2
@#define f(x) = x + " + " + y[i]
@#define i = 1

model;
  A = @{y[i] + f("D")};
end;
```

The latter is strictly equivalent to:

```
model;
  A = BD + B;
end;
```

Macro Directive: `@#if MACRO_EXPRESSION`

Macro Directive: `@#ifdef MACRO_VARIABLE`

Macro Directive: `@#ifndef MACRO_VARIABLE`

Macro Directive: `@#elseif MACRO_EXPRESSION`

Macro Directive: `@#else @#endif`

Conditional inclusion of some part of the `.mod` file. The lines between `@#if`, `@#ifdef`, or `@#ifndef` and the next `@#elseif`, `@#else` or `@#endif` is executed only if the condition evaluates to `true`. Following the `@#if` body, you can zero or more `@#elseif` branches. An `@#elseif` condition is only evaluated if the preceding `@#if` or `@#elseif` condition evaluated to `false`. The `@#else` branch is optional and is only evaluated if all `@#if` and `@#elseif` statements evaluate to false.

Note that when using `@#ifdef`, the condition will evaluate to `true` if the MACROVARIABLE has been previously defined, regardless of its value. Conversely, `@#ifndef` will evaluate to true if the MACROVARIABLE has not yet been defined.

Note that when using `@#elseif` you can check whether or not a variable has been defined by using the `defined` operator. Hence, to enter the body of an `@#elseif` branch if the variable `X` has not been defined, you would write: `@#elseif !defined(X)`.

Note that if a real appears as the result of the MACROEXPRESSION, it will be interpreted as a boolean; a value of `0` is interpreted as `false`, otherwise it is interpreted as `true`. Further note that because of the imprecision of reals, extra care must be taken when testing them in the MACROEXPRESSION. For example, `exp(log(5)) == 5` will evaluate to `false`. Hence, when comparing real values, you should generally use a zero tolerance around the value desired, e.g. `exp(log(5)) > 5-1e-14 && exp(log(5)) < 5+1e-14`

Example

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

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

This would result in:

```
...
model;
  i = i(-1)^w * i_ss^(1-w) * (pie/piestar)^w2;
...
end;
```

Example

Choose between two alternative monetary policy rules using a macro-variable. The only difference between this example and the previous one is the use of `@#ifdef` instead of `@#if`. Even though `linear_mon_pol` contains the value `false` because `@#ifdef` only checks that the variable has been defined, the linear monetary policy is output:

```
@#define linear_mon_pol = false // 0 would be treated the same
...
model;
@#ifdef linear_mon_pol
  i = w*i(-1) + (1-w)*i_ss + w2*(pie-piestar);
@#else
  i = i(-1)^w * i_ss^(1-w) * (pie/piestar)^w2;
@#endif
...
end;
```

This would result in:

```
...
model;
  i = w*i(-1) + (1-w)*i_ss + w2*(pie-piestar);
...
end;
```

Macro Directive: `@#for MACRO_VARIABLE in MACRO_EXPRESSION`

Macro Directive: `@#for MACRO_VARIABLE in MACRO_EXPRESSION when MACRO_EXPRESSION`

Macro Directive: `@#for MACRO_TUPLE in MACRO_EXPRESSION`

Macro Directive: `@#for MACRO_TUPLE in MACRO_EXPRESSION when MACRO\_EXPRESSION`

Macro Directive: `@#endfor`

Loop construction for replicating portions of the `.mod` file. Note that this construct can enclose variable/parameters declaration, computational tasks, but not a model declaration.

Example

```
model;
@#for country in [ "home", "foreign" ]
  GDP_@{country} = A * K_@{country}^a * L_@{country}^(1-a);
@#endfor
end;
```

The latter is equivalent to:

```
model;
  GDP_home = A * K_home^a * L_home^(1-a);
  GDP_foreign = A * K_foreign^a * L_foreign^(1-a);
end;
```

Example

```
model;
@#for (i, j) in ["GDP"] * ["home", "foreign"]
  @{i}_@{j} = A * K_@{j}^a * L_@{j}^(1-a);
@#endfor
end;
```

The latter is equivalent to:

```
model;
  GDP_home = A * K_home^a * L_home^(1-a);
  GDP_foreign = A * K_foreign^a * L_foreign^(1-a);
end;
```

Example

```
@#define countries = ["US", "FR", "JA"]
@#define nth_co = "US"
model;
@#for co in countries when co != nth_co
  (1+i_@{co}) = (1+i_@{nth_co}) * E_@{co}(+1) / E_@{co};
@#endfor
  E_@{nth_co} = 1;
end;
```

The latter is equivalent to:

```
model;
  (1+i_FR) = (1+i_US) * E_FR(+1) / E_FR;
  (1+i_JA) = (1+i_US) * E_JA(+1) / E_JA;
  E_US = 1;
end;
```

Macro Directive: `@#echo MACRO_EXPRESSION`

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

Macro Directive: `@#error MACRO_EXPRESSION`

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

Macro Directive: `@#echomacrovars`

Macro Directive: `@#echomacrovars MACRO_VARIABLE_LIST`

Macro Directive: `@#echomacrovars(save) MACRO_VARIABLE_LIST`

Asks the preprocessor to display the value of all macro variables up until this point. If the `save` option is passed, then values of the macro variables are saved to `options_.macrovars_line_<<line_numbers>>`. If `NAME_LIST` is passed, only display/save variables and functions with that name.

Example

```
@#define A = 1
@#define B = 2
@#define C(x) = x*2
@#echomacrovars A C D
```

The output of the command above is:

```
Macro Variables:
  A = 1
Macro Functions:
  C(x) = (x * 2)
```

Typical usages

Modularization

The `@#include` directive can be used to split `.mod` files into several modular components.

Example setup:

`modeldesc.mod`

Contains variable declarations, model equations, and shocks declarations.

`simul.mod`

Includes `modeldesc.mod`, calibrates parameter,s and runs stochastic simulations.

`estim.mod`

Includes `modeldesc.mod`, declares priors on parameters, and runs Bayesian estimation.

Dynare can be called on `simul.mod` and `estim.mod` but it makes no sense to run it on `modeldesc.mod`.

The main advantage is that you don't have to copy/paste the whole model (at the beginning) or changes to the model (during development).

Indexed sums of products

The following example shows how to construct a moving average:

```
@#define window = 2

var x MA_x;
...
model;
...
MA_x = @{1/(2*window+1)}*(
@#for i in -window:window
        +x(@{i})
@#endfor
       );
...
end;
```

After macro processing, this is equivalent to:

```
var x MA_x;
...
model;
...
MA_x = 0.2*(
        +x(-2)
        +x(-1)
```

```
            +x(0)
            +x(1)
            +x(2)
           );
...
end;
```

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;
```

Endogeneizing parameters

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

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

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

and the labor share in GDP is defined as:

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

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

It is clear that calibrating α is not straightforward; on the contrary, we have real world data for `lab_rat` and it is clear that these two variables are economically linked.

The solution is to use a method called variable flipping, which consists in changing the way of computing the steady state. During this computation, α will be made an endogenous variable and `lab_rat` will be made a parameter. An economically relevant value will be calibrated for `lab_rat`, and the solution algorithm will deduce the implied value for α.

An implementation could consist of the following files:

`modeqs.mod`

This file contains variable declarations and model equations. The code for the declaration of α and `lab_rat` would look like:

```
@#if steady
  var alpha;
  parameter lab_rat;
@#else
  parameter alpha;
  var lab_rat;
@#endif
```

`steady.mod`

This file computes the steady state. It begins with:

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

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

`simul.mod`

This file computes the simulation. It begins with:

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

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

Julia loops versus macro processor loops

Suppose you have a model with a parameter ρ and you want to run simulations for three values: $\rho = 0.8, 0.9, 1$. There are several ways of doing this:

With a Julia 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, Julia manages the iterations. This is interesting when there are a lot of iterations.

With a macro processor loop (case 1)

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

This is very similar to the previous example, except that the loop is unrolled. The macro processor manages the loop index but not the data array (`rhos`).

With a macro processor loop (case 2)

```
@#for rho_val in [ 0.8, 0.9, 1]
  rho = @{rho_val};
  stoch_simul(order=1);
@#endfor
```

The advantage of this method is that it uses a shorter syntax, since the list of values is directly given in the loop construct. The inconvenience is that you can not reuse the macro array in Julia.

15.2 Verbatim inclusion

Pass everything contained within the verbatim block to the `<mod_file>.m` file.

Block: `verbatim ;`

By default, whenever Dynare encounters code that is not understood by the parser, it is directly passed to the preprocessor output.

In order to force this behavior you can use the `verbatim` block. This is useful when the code you want passed to the `<mod_file>.m` file contains tokens recognized by the Dynare preprocessor.

Example

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

15.3 Misc commands

Command: ‘saveparamsandsteadystate (FILENAME);

For all parameters, endogenous and exogenous variables, stores their value in a text file, using a simple name/value associative table.

- for parameters, the value is taken from the last parameter initialization.
- for exogenous, the value is taken from the last `initval` block.

- for endogenous, the value is taken from the last steady state computation (or, if no steady state has been computed, from the last `initval` block).

Note that no variable type is stored in the file, so that the values can be reloaded with `load_params_and_steady_state` in a setup where the variable types are different.

The typical usage of this function is to compute the steady-state of a model by calibrating the steady-state value of some endogenous variables (which implies that some parameters must be endogeneized during the steady-state computation).

You would then write a first `.mod` file which computes the steady state and saves the result of the computation at the end of the file, using `save_params_and_steady_state`.

In a second file designed to perform the actual simulations, you would use `load_params_and_steady_state` just after your variable declarations, in order to load the steady state previously computed (including the parameters which had been endogeneized during the steady state computation).

The need for two separate `.mod` files arises from the fact that the variable declarations differ between the files for steady state calibration and for simulation (the set of endogenous and parameters differ between the two); this leads to different `var` and `parameters` statements.

Also note that you can take advantage of the `@#include` directive to share the model equations between the two files (see `macro-proc-lang`).

- `load_params_and_steady_state (FILENAME);`

For all parameters, endogenous and exogenous variables, loads their value from a file created with `save_params_and_steady_sta`

- for parameters, their value will be initialized as if they had been calibrated in the `.mod` file.
- for endogenous and exogenous variables, their value will be initialized as they would have been from an `initval` block .

This function is used in conjunction with `save_params_and_steady_state`; see the documentation of that function for more information.