

Advanced Dynare

Chapter III

The Dynare macroprocessor

Johannes Pfeifer

Banco de la República Colombia

November 2023



Copyright © 2008-2023 Dynare Team
License: [Creative Commons Attribution-ShareAlike 4.0](https://creativecommons.org/licenses/by-sa/4.0/)

Outline

1. Overview

2. Syntax

3. Common uses

Outline

1. Overview

2. Syntax

3. Common uses

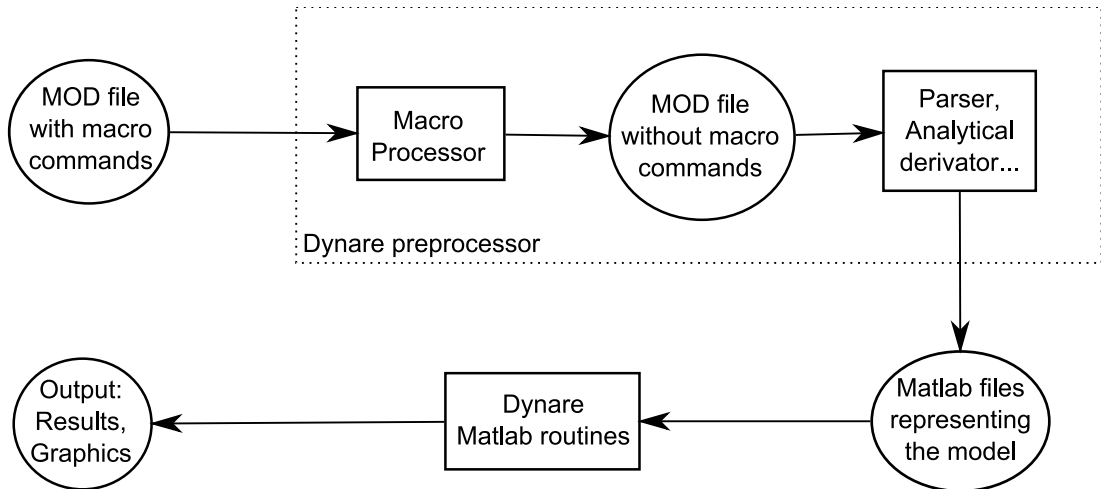
Motivation

- The **Dynare language** (used in `.mod` files) is well suited for many economic models
 - It's a markup language that defines models
 - but lacks programmatic element
- The **Dynare macro language** adds programmatic element
 - Introduces conditionals, loops, and other simple programmatic directives
 - Used to speed up model development
 - Useful in various situations
 - Multi-country models
 - Creation of modular `.mod` files
 - Variable flipping
 - Conditional inclusion of equations
 - ... among others

Design of the macro language

- The Dynare macro language provides a set of **macro commands** that can be used in `.mod` files
- The macro processor transforms a `.mod` file with macro commands into a `.mod` file without macro commands (doing text expansions/inclusions) and then feeds it to the Dynare parser
- Key point: the macro processor only performs **text substitution**
→ objects computed in later steps are not available yet and cannot be conditioned on

Dynare Flowchart



Outline

1. Overview

2. Syntax

3. Common uses

Macro Directives

- Directives begin with: `@#`
- A directive gives instructions to the macro processor
- Main directives are:
 - file inclusion: `@#include`
 - definition of a macro processor variable or function: `@#define`
 - conditional statements: `@#if/@#ifdef/@#ifndef/@#else/@#elseif/@#endif`
 - loop statements: `@#for/@#endfor`
- Most directives fit on one line
→ two backslashes (*i.e.* `\\`) at end of line indicate directive is continued
- Directives are not terminated with a semicolon

Values and types

- The macro processor can handle values of 5 different types:
 1. boolean (logical value, true or false)
 2. real (double precision floating point number)
 3. string (of characters)
 4. tuple
 5. array
- Values of the types listed above can be cast to other types
 - (real) "3.1" \rightarrow 3.1
 - (string) 3.1 \rightarrow "3.1"
 - (array) 4 \rightarrow [4]
 - (real) [5] \rightarrow 5
 - (real) [6, 7] \rightarrow error
 - (bool) -1 && (bool) 2 \rightarrow true

Macro-expressions (1/8)

- Macro-expressions are constructed using literals (*i.e.* fixed values) of the 5 basic types described above, macro-variables, standard operators, function calls, and comprehensions
- Macro-expressions can be used in two places:
 - inside macro directives
 - no special markup required
 - in the body of the `.mod` file, between an `@`-sign and curly braces (like `@{expr}`)
 - the macro processor will substitute the expression with its value

Macro-expressions (2/8): Boolean

- Boolean literals are `true` and `false`.

Operators on booleans

- comparison operators: `==` `!=`
- logical operators:
 - conjunction (“and”): `&&`
 - disjunction (“or”): `||`
 - negation (“not”): `!`

Macro-expressions (3/8): Real

Operators on reals

- arithmetic operators: `+` `-` `*` `/` `^`
- comparison operators: `<` `>` `<=` `>=` `==` `!=`
- logical operators: `&&` `||` `!`
- range with unit increment: `1:4` is equivalent to real array `[1, 2, 3, 4]`. (NB: `[1:4]` is equivalent to an array containing an array of reals, *i.e.* `[[1, 2, 3, 4]]`)
- range with user-defined increment: `4:-1.1:-1` is equivalent to real array `[4, 2.9, 1.8, 0.7, -0.4]`.

Functions for reals

- `min`, `max`, `exp`, `ln` (or `log`), `log10`
- `sign`, `floor`, `ceil`, `trunc`, `round`, `mod`
- `sin`, `cos`, `tan`, `asin`, `acos`, `atan`
- `sqrt`, `cbirt`, `erf`, `erfc`, `normpdf`, `normcdf`, `gamma`, `lgamma`

Macro-expressions (4/8): String

- String literals have to be declared between *double* quotes, e.g. "string"

Operators on character strings

- comparison operators: < > <= >= == !=
- concatenation: +
- string length: length()
- string emptiness: isempty()
- extraction of substrings: if `s` is a string, then one can write `s[3]` or `s[4:6]`

Macro-expressions (5/8): Tuple

- Tuples are enclosed by parentheses and elements are separated by commas
 - (a,b,c)
 - (1,2.2,c)

Operators on tuples

- comparison operators: == !=
- functions: length(), isempty()
- testing membership in tuple: in operator
(example: "b" in ("a", "b", "c") returns true)

Macro-expressions (6/8): Array (1/2)

- Arrays are enclosed by brackets, and their elements are separated by commas
 - `[1, [2,3], 4]`
 - `["US", "EA"]`

Operators on arrays

- comparison operators: `==` `!=`
- dereferencing: if `v` is an array, then `v[2]` is its 2nd element
- concatenation: `+`
- functions: `sum()`, `length()`, `isempty()`
- extraction of sub-arrays: e.g. `v[4:6]`
- testing membership of an array: `in` operator
(example: `"b" in ["a", "b", "c"]` returns `true`)

Macro-expressions (6/8): Array (2/2)

- Arrays represent a set of elements (assuming unique elements)

Set operations on arrays

- set union: $|$
 - set intersection: $\&$
 - set difference: $-$
 - Cartesian product of two arrays: $*$
 - Cartesian power of an array: \wedge
- if A and B are arrays, the following set operations are valid: $A|B$, $A\&B$, $A-B$, $A*B$, $A\wedge 3$
 - array resulting from Cartesian product or power has tuples as its elements

Macro-expressions (7/8): Comprehension (1/3)

- Comprehensions are shorthand way of creating arrays from other arrays
- Done by filtering, mapping, or both

Filtering

- Allows choosing all array elements satisfying a condition
- Syntax: [*variable/tuple* in *array* when *condition*]
- Example: Choose even numbers from array
 - Code: [i in 1:5 when mod(i,2) == 0]
 - Result: [2, 4]

Macro-expressions (7/8): Comprehension (2/3)

Mapping

- Allows applying transformations to all array elements
- Syntax: [*expr* for *variable/tuple* in *array*]
- Example: Square elements in array
 - Code: [i^2 for i in 1:5]
 - Result: [1, 4, 9, 16, 25]
- Example: Swap pairs of an array
 - Code: [(j,i) for (i,j) in (1:2)^2]
 - Result: [(1, 1), (2, 1), (1, 2), (2, 2)]

Macro-expressions (7/8): Comprehension (3/3)

Mapping and Filtering

- Allows apply a transformation to selected array elements
- Syntax: [*expr* for *variable/tuple* in *array* when *condition*]
- Example: Square of odd numbers between 1 and 5
 - Code: [`i^2 for i in 1:5 when mod(i,2) == 1`]
 - Result: [1, 9, 25]

Macro-expressions (8/8): Functions

- Can take any number of arguments
- Dynamic binding: is evaluated when invoked during the macroprocessing stage, not when defined
- Can be included in expressions; valid operators depend on return type

Declaration syntax

```
@#define function_signature= expression
```

Example

If we declare the following function:

```
@#define distance(x, y) = sqrt(x^2 + y^2)
```

Then `distance(3, 4)` will be equivalent to 5.

Defining macro-variables

- value of a macro-variable can be defined with `@#define` directive
- list of macro variables is distinct from model and MATLAB/Octave variables

Syntax

```
@#define variable_name = expression
```

Examples

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

NB: You can define macro variables on the Dynare command line by using the `-D` option

Expression substitution

Before macro processing

```
@#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;
```

After macro processing

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

Include directive (1/3)

- This directive simply inserts the text of another file in its place

Syntax

```
@#include "filename"
```

Example

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

- Equivalent to copy/paste of content of included file
- Nesting include is allowed (*i.e.* to include a file with an included file)

Include directive (2/3)

- The filename can be given by a macro-variable (useful in loops):

Example with variable

```
@#define fname = "modelcomponent.mod"  
@#include fname
```

- Files to include are searched for in the current directory. Other directories can be added with the
 - `@#includepath` directive
 - the `-I` command line option
 - `[paths]` section in config files

Include directive (3/3): A dirty hack

- Include directives allow to hack your way around macro processor limitations like not being able to access and condition on model objects like parameter values
- Often feasible to use outer layer of MATLAB code that loops over cases and then write Dynare code to a text file included in the main mod-file
- A crude example is

```
equations={'k=0.9*k(-1)+invest;'
           'k=0.8*k(-1)+invest;'};
for ii=1:size(equations,1)
    fid=fopen('my_equation.mod','w');
    fprintf(fid,'%s \n',equations{ii,1});
    fclose(fid);
    dynare my_modfile
end
```

Loop directive (1/4)

- Syntax 1: Simple iteration over one variable

```
@#for variable_name in array_expr  
    loop_body  
@#endfor
```

- Syntax 2: Iteration over several variables at the same time

```
@#for tuple_name in array_expr  
    loop_body  
@#endfor
```

- Syntax 3: Iteration with some values excluded (filtering)

```
for tuple_or_variable in array_expr when expr  
    loop_body  
@#endfor
```

Loop directive (2/4)

Example: before macro processing

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

Example: after macro processing

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

Loop directive (3/4)

Example: loop over several variables

```
@#define A = [ "X", "Y", "Z"]
@#define B = [ 1, 2, 3]

model;
@#for (i,j) in A*B
    e_{i}_{j} = ...
@#endfor
end;
```

This will loop over e_{X_1} , e_{X_2} , ..., e_{Z_3} (9 variables in total)

Loop directive (4/4)

Example: loop over several variables with filtering

```
model;  
@#for (i,j,k) in (1:10)^3 when i^2+j^2==k^2  
  e_{i}_{j}_{k} = ...  
@#endfor  
end;
```

- This loop will iterate over only 4 triplets:
 - (3,4,5)
 - (4,3,5)
 - (6,8,10)
 - (8,6,10)

Conditional directives (1/3)

Syntax 1

```
@#if bool_or_real_expr  
    body included if expr is true (or != 0)  
@#endif
```

Syntax 2

```
@#if bool_or_real_expr  
    body included if expr is true (or != 0)  
@#else  
    body included if expr is false (or 0)  
@#endif
```

Conditional directives (2/3)

Syntax 3

```
@#if bool_or_real_expr1
    body included if expr1 is true (or != 0)
@#elseif bool_or_real_expr2
    body included if expr2 is true (or !=0)
@#else
    body included if expr1 and expr2 are false (or 0)
@#endif
```

Example: alternative monetary policy rules

```
@#define linear_mon_pol = false // or 0
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;
```

Conditional directives (3/3)

Syntax 1

```
@#ifdef variable_name
    body included if variable defined
@#endif
```

Syntax 2

```
@#ifdef variable_name
    body included if variable defined
@#else
    body included if variable not defined
@#endif
```

- There is also `@#ifndef`, which is the opposite of `@#ifdef`
→ tests whether a variable is *not* defined
- There is *no* `@#elseifdef` or `@#elseifndef` directive
→ use `elseif defined(variable_name)`

Echo directives

- The echo directive will display a message on standard output
- echomacrovars directive will display all (or specified) macro variables and their values
- The save option allows saving this information to options_ .macrovars_line_x, where x denotes the line number where the statement was encountered

Syntax

```
@#echo string_expr  
@#echomacrovars  
@#echomacrovars list_of_variables  
@#echomacrovars(save)  
@#echomacrovars(save) list_of_variables
```

Examples

```
@#echo "Information"
```

Error directive

- The error directive will display the message and make Dynare stop
→ only makes sense inside a conditional directive

Syntax

```
@#error string_expr
```

Example

```
@#error "Error message!"
```

Macro-related command line options

- `savemacro`: saves the output of the macro processor
→ if `.mod` file is called `file.mod`, output is saved to `file-macroexp.mod`
- `savemacro=filename` allows user-defined file name
- `linemacro`: In the output of `savemacro`, print line numbers where macro directives were placed
- `onlymacro`: stops processing after the macro processing step

```
dynare FILENAME[.mod] onlymacro savemacro=macroresult linemacro
```

Outline

1. Overview

2. Syntax

3. Common uses

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 shock declarations
 - `simulate.mod`: includes `modeldesc.mod`, calibrates parameters, and runs stochastic simulations
 - `estim.mod`: includes `modeldesc.mod`, declares priors on parameters, and runs Bayesian estimation
- Dynare can be called on `simulate.mod` and `estim.mod`, but not on `modeldesc.mod`
- Advantage: no need to manually copy/paste the whole model (during initial development) or port model changes (during development)

Indexed sums or products

Example: moving average

Before macro processing

```
@#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

```
var x MA_x;
...
model;
...
MA_x = 1/5*(
  +x(-2)
  +x(-1)
  +x(0)
  +x(1)
  +x(2)
);
end;
```

Multi-country models

.mod file bare bones example

```

#define countries = [ "US", "EA", "AS", "JP", "RC" ]
#define nth_co = "US"

#define for co in countries
var Y_{co} K_{co} L_{co} i_{co} E_{co} ...;
parameters a_{co} ...;
varexo ...;
#define endfor

model;
#define 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
#define endfor
end;
```

Endogeneizing parameters (1/4)

- When calibrating the model, it may be useful to pin down parameters by targeting endogenous objects
- Example:

$$y_t = \left(\alpha^{\frac{1}{\xi}} l_t^{1-\frac{1}{\xi}} + (1-\alpha)^{\frac{1}{\xi}} k_t^{1-\frac{1}{\xi}} \right)^{\frac{\xi}{\xi-1}} \quad (1)$$

$$lab_rat_t = \frac{w_t l_t}{p_t y_t} \quad (2)$$

- In the model, α is a (share) parameter, and lab_rat_t is an endogenous variable
- We observe that:
 - setting a value for α is not straightforward!
 - but we have real world data for lab_rat_t
 - it is clear that these two objects are economically linked

Endogeneizing parameters (2/4)

- Therefore, when computing the steady state by solving the static model:
 - we make α a variable and the steady state value lab_rat of the dynamic variable lab_rat_t a parameter
 - we impose an economically sensible value for lab_rat
 - the solution algorithm deduces the implied value for α
- We call this method “variable flipping”, because it treats α as a variable and lab_rat as a parameter for the purpose of the static model

Endogeneizing parameters (3/4)

Example implementation

- File `modeqs.mod`:
 - contains variable declarations and model equations
 - For declaration of `alpha` and `lab_rat`:

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

Endogeneizing parameters (4/4)

Example implementation

- File `steadystate.mod`:
 - begins with `@#define steady = true`
 - followed by `@#include "modeqs.mod"`
 - initializes parameters (including `lab_rat`, excluding `alpha`)
 - computes steady state (using guess values for endogenous, including `alpha`)
 - saves values of parameters and variables at steady-state in a file, using the `save_params_and_steady_state` command
- File `simulate.mod`:
 - begins with `@#define steady = false`
 - followed by `@#include "modeqs.mod"`
 - loads values of parameters and variables at steady-state from file, using the `load_params_and_steady_state` command
 - computes simulations

MATLAB/Octave loops vs macro processor loops (1/3)

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

With a MATLAB/Octave loop

```
rhos = [ 0.8, 0.9, 1];  
for i = 1:length(rhos)  
    set_param_value('rho',rhos(i));  
    stoch_simul(order=1);  
    if info(1)~=0  
        error('Simulation failed for parameter draw')  
    end  
end  
end
```

- The loop is not unrolled, MATLAB/Octave manages iterations
- NB: always check whether the error flag `info(1)==0` to prevent erroneously relying on stale results from previous iterations

MATLAB/Octave loops vs macro processor loops (2/3)

With a macro processor loop (case 1)

```
rhos = [ 0.8, 0.9, 1];  
@#for i in 1:3  
    set_param_value('rho',rhos(@{i}));  
    stoch_simul(order=1);  
    if info(1)~=0  
        error('Simulation failed for parameter draw')  
    end  
@#endfor
```

- Very similar to previous example
- Loop is unrolled
- Dynare macro processor manages the loop index but not the data array (rhos)

MATLAB/Octave loops vs macro processor loops (3/3)

With a macro processor loop (case 2)

```
@#for rho_val in [ 0.8, 0.9, 1]
  set_param_value('rho',@{rho_val});
  stoch_simul(order=1);
  if info(1)~=0
    error('Simulation failed for parameter draw')
  end
#@endfor
```

- Shorter syntax, since list of values directly given in the loop construct
- Downside: array not stored as MATLAB/Octave variable
→ cannot be used in MATLAB/Octave