The Dynare Macro Processor

Sébastien Villemot   Houtan Bastani

C E P R E M A P
CENTRE POUR LA RECHERCHE ECONOMIQUE ET SES APPLICATIONS

21 June 2021

Outline

1. **Overview**

2. **Syntax**

3. **Common uses**

Outline

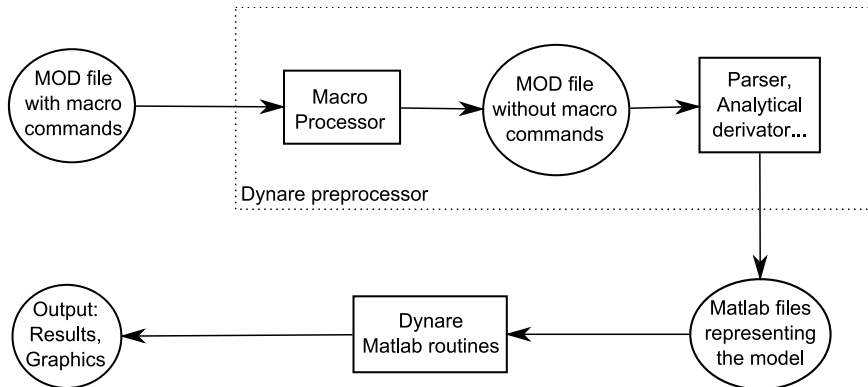1. **Overview**

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
  - ▶ Lacks a programmatic element
- The **Dynare macro language** adds a programmatic element to Dynare
  - ▶ Introduces conditionals, loops, and other simple programmatic directives
  - ▶ Used to speed 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
- The key point to understand is that the macro processor only does **text substitution** (like the C preprocessor or the PHP language)

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. If needed however, two backslashes (i.e. `\\`) at the end of a line indicate that the directive is continued on the next line.

Values

- The macro processor can handle values of 5 different types:
  - boolean
  - real
  - string
  - tuple
  - 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 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 is required
  - in the body of the `.mod` file, between an at sign and curly braces (like `@{expr}`); the macro processor will substitute the expression with its value

Macro-expressions (2/8): Boolean

Boolean literals are `true` and `false`.

Operators on booleans
- comparison operators: `==` `!=`
- logical operators: `&&` `||` `!`

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
- sin, cos, tan, asin, acos, atan
- sqrt, cbrt, sign, floor, ceil, trunc, mod
- erf, erfc, normpdf, normcdf, gamma, lgamma, round

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 parenthesis and elements separated by commas (like (a,b,c) or (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 (like [1,[2,3],4] or ["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 can be seen as representing a set of elements (assuming no element appears twice in the array). Several set operations can thus be performed on arrays: union, intersection, difference, cartesian product and power.

Set operations on arrays

- set union: |
- set intersection: &
- set difference: −
- Cartesian product of two arrays: *
- Cartesian power of an array: ^

For example, if `A` and `B` are arrays, then the following set operations are valid: `A|B`, `A&B`, `A−B`, `A*B`, `A^3`.

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

Comprehensions are a shorthand way of creating arrays from other arrays. This is done by filtering, mapping, or both.

Filtering

- Allows one to choose those elements from an array for which a condition holds
- 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 one to apply a transformation to every element of an array
- 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 one to apply a transformation to the elements selected from an array
- 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, 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

The value of a macro-variable can be defined with the @#define directive.
The macro processor has its own list of variables which are different from
model variables 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

Dummy example

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 (1/2)

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

Syntax

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

Example

`@#include "modelcomponent.mod"`

- Equivalent to a copy/paste of the content of the included file
- Note that it is possible to nest includes (*i.e.* to include a file with an included file)

Include directive (2/2)

- 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, or the [paths] section in config file.

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 in array_expr
    loop_body
@#endfor
```

Syntax 3: Iteration with some values excluded

```
@#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` (*i.e.* it tests whether a variable is *not* defined). NB: There is *no* `@#elseifdef` or `@#elseifndef` directive; use `elseif defined(variable_name)` to achieve the desired objective.

Echo and error directives

- The echo directive will simply display a message on standard output
- The error directive will display the message and make Dynare stop (only makes sense inside a conditional directive)
- The echomacrovars directive will display all of the macro variables (or those specified) and their values, optionally saving them

Syntax

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

Examples

```
@#echo "Information message."
@#error "Error message!"
```

Macro-related command line options

- `savemacro`: Useful for debugging or learning purposes, saves the output of the macro processor. If your `.mod` file is called `file.mod`, the output is saved to `file-macroexp.mod`.
- `nolinemacro`: In the output of `savemacro`, don't print line numbers where the macro directives were placed.
- `noemptylinemacro`: Remove empty lines in the output of `savemacro`.
- `onlymacro`: Stops processing after the macro processing step.

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 shocks 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 it makes no sense to run it on modeldesc.mod
- Advantage: no need to manually copy/paste the whole model (at the beginning) or changes to the model (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 skeleton example

```
@#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 (1/4)

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

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

$$lab\_rat = \frac{w\ell}{py}$$

- In the model, α is a (share) parameter, and lab_rat is an endogenous variable
- We observe that:
    - calibrating α is not straigthforward!
    - on the contrary, we have real world data for lab_rat
    - it is clear that these two variables are economically linked

Endogeneizing parameters (2/4)

- Therefore, when computing the steady state:
  - ▸ we make α an endogenous variable and *lab_rat* a parameter
  - ▸ we impose an economically relevant value for *lab_rat*
  - ▸ the solution algorithm deduces the implied value for α
- We call this method "variable flipping"

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`
  - then with `@#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 endogenous at steady-state in a file, using the `save_params_and_steady_state` command
- File `simulate.mod`:
  - begins with `@#define steady = false`
  - then with `@#include "modeqs.mod"`
  - loads values of parameters and endogenous 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)
  rho = rhos(i);
  stoch_simul(order=1);
end
```

- The loop is not unrolled
- MATLAB/Octave manages the 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
  rho = rhos(@{i});
  stoch_simul(order=1);
@#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]
  rho = @{rho_val};
  stoch_simul(order=1);
@#endfor
```

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

Thanks for your attention!

Questions?