The Dynare Macro-processor

Sébastien Villemot

CEPREMAP

June 29, 2010

Outline

Outline

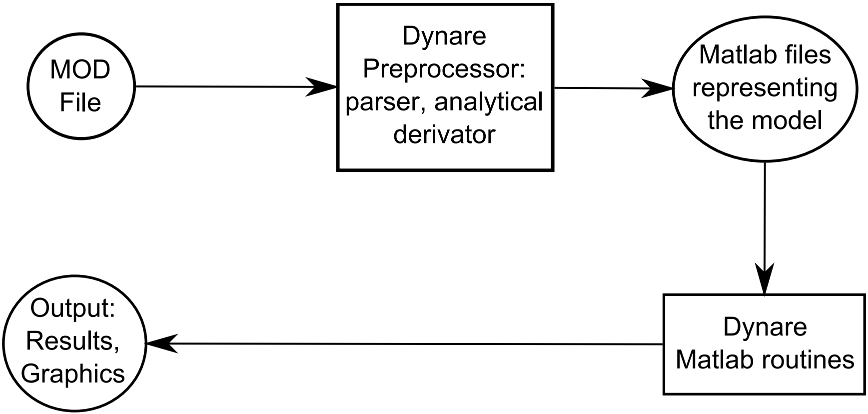
Motivation

- The **Dynare language** (used in MOD files) is well suited for many economic models
- However, as such, it lacks some useful features, such as:
    - a loop mechanism for automatically repeating similar blocks of equations (such as in multi-country models)
    - an operator for indexed sums or products inside equations
    - a mechanism for splitting large MOD files in smaller modular files
    - the possibility of conditionally including some equations or some runtime commands
- The **Dynare Macro-language** was specifically designed to address these issues
- Being flexible and fairly general, it can also be helpful in other situations

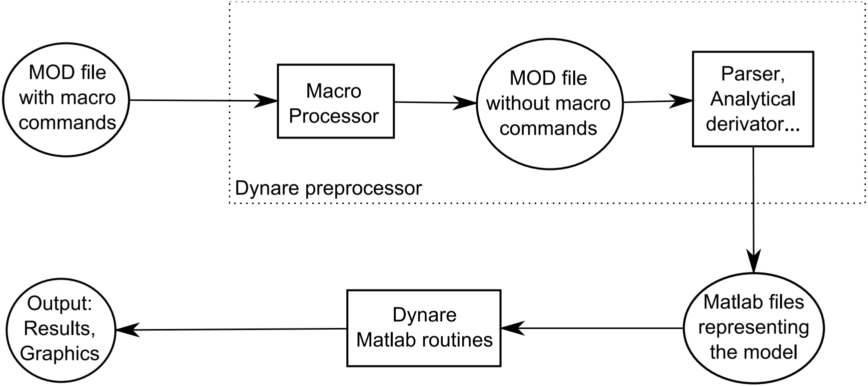
Design of the macro-language

- The Dynare Macro-language provides a new set of **macro-commands** which can be inserted inside MOD files
- Language features include:
    - file inclusion
    - loops (*for* structure)
    - conditional inclusion (*if/then/else* structures)
    - expression substitution
- Implemented in Dynare starting from version 4.0
- 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)

Old Dynare design



New Dynare design



Outline

Macro Directives

- Directives begin with an at-sign followed by a pound sign (`@#`)
- A directive produces no output, but gives instructions to the macro-processor
- Main directives are:
  - file inclusion: `@#include`
  - definition a variable of the macro-processor: `@#define`
  - conditional statements (`@#if/@#then/@#else/@#endif`)
  - loop statements (`@#for/@#endfor`)
- In most cases, directives occupy exactly one line of text. In case of need, two anti-slashes (\\) at the end of the line indicates that the directive is continued on the next line.

Inclusion directive

- This directive simply includes the content of another file at the place where it is inserted.

Syntax

@#include "*filename*"

Example

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

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

Variables

- The macro processor maintains its own list of variables (distinct of model variables and of MATLAB variables)
- Macro-variables can be of four types:
  - integer
  - character string (declared between *double* quotes)
  - array of integers
  - array of strings
- No boolean type:
  - false is represented by integer zero
  - true is any non-null integer

Macro-expressions (1/2)

It is possible to construct macro-expressions, using standard operators.

Operators on integers

- arithmetic operators: `+ - * /`
- comparison operators: `< > <= >= == !=`
- logical operators: `&& || !`
- integer ranges: `1:4` is equivalent to integer array `[1,2,3,4]`

Operators on character strings

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

Macro-expressions (2/2)

Operators on arrays

- dereferencing: if `v` is an array, then `v[2]` is its 2^{nd} element
- concatenation: `+`
- difference `-`: returns the first operand from which the elements of the second operand have been removed
- extraction of sub-arrays: *e.g.* `v[4:6]`
- testing membership of an array: `in` operator (example: `"b" in ["a", "b", "c"]` returns `1`)

Macro-expressions can be used at two places:

- inside macro directives, directly
- in the body of the MOD file, between an at-sign and curly braces (like `@{expr}`): the macro processor will substitute the expression with its value

Define directive

The value of a macro-variable can be defined with the `@#define` directive.

Syntax

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

Examples

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

Expression substitution

Dummy example

Before macro-processing

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

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

After macro-processing

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

Loop directive

Syntax

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

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;
```

Conditional inclusion directive

Syntax 1

@#if *integer_expr*
    body included if expr != 0
@#endif

Syntax 2

@#if *integer_expr*
    body included if expr != 0
@#else
    body included if expr == 0
@#endif

Example: alternative monetary policy rules

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

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 inclusion directive)

Syntax

`@#echo` *string_expr*

`@#error` *string_expr*

Examples

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

Saving the macro-expanded MOD file

- For **debugging or learning** purposes, it is possible to save the output of the macro-processor
- This output is a valid MOD file, obtained after processing the macro-commands of the original MOD file
- Just add the `savemacro` option on the Dynare command line (after the name of your MOD file)
- If MOD file is `filename.mod`, then the macro-expanded version will be saved in `filename-macroexp.mod`
- You can specify the filename for the macro-expanded version with the syntax `savemacro=mymacroexp.mod`

Outline

Modularization

- The `@#include` directive can be used to split MOD files into several modular components
- Example setup:

  `modeldesc.mod`: contains variable declarations, model equations and shocks declarations

  `simul.mod`: includes `modeldesc.mod`, calibrates parameters and runs stochastic simulations

  `estim.mod`: includes `modeldesc.mod`, declares priors on parameters and runs bayesian estimation
- Dynare can be called on `simul.mod` and `estim.mod`
- But it makes no sense to run it on `modeldesc.mod`
- 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 doing the steady-state calibration of 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 `steady.mod`:
  - begins with `@#define steady = 1`
  - 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 `simul.mod`:
  - begins with `@#define steady = 0`
  - 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 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 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 manages the iterations
- Interesting when there are a lot of iterations

MATLAB 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 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
```

- Advantage: shorter syntax, since list of values directly given in the loop construct
- Note that values are given as character strings (the macro-processor does not know floating point values)
- Inconvenient: can not reuse an array stored in a MATLAB variable

Outline

Possible future developments

- Find a nicer syntax for indexed sums/products
- Implement other control structures: `elsif`, `switch/case`, `while/until` loops
- Implement macro-functions (or templates), with a syntax like:
  `@#define QUADRATIC_COST(x, x_ss, phi) = phi/2*(x/x_ss-1)^2`

Dynare for Octave (1/2)

- GNU Octave (or simply Octave) is a high-level language, primarily intended for numerical computations
- Basically, it is a free clone of MATLAB: same syntax, almost same set of functions
- Runs on Windows, Linux and Mac OS X
- Advantages:
  - free software, no license fee to pay
  - source code available
  - dynamic and responsive community of users and developers
- Inconvenients:
  - slower than MATLAB
  - less user friendly (however note that there is a graphical user interface to Octave called "qtoctave")

Dynare for Octave (2/2)

- Since version 4.0, Dynare works on top of Octave
- This makes Dynare 100% free software
- All features of Dynare work with Octave, except:
  - loading of Excel files for estimation
  - diffuse Kalman filter (used in models with unit roots)
  - some graphics automatically generated look bad, it may be necessary to recreate them manually
- For more information:
  `http://www.dynare.org/DynareWiki/DynareOctave`