The Dynare Macro-processor

Dynare Summer School 2008

Sébastien Villemot

Bank of France - Paris School of Economics

July 3, 2008

Outline

Outline

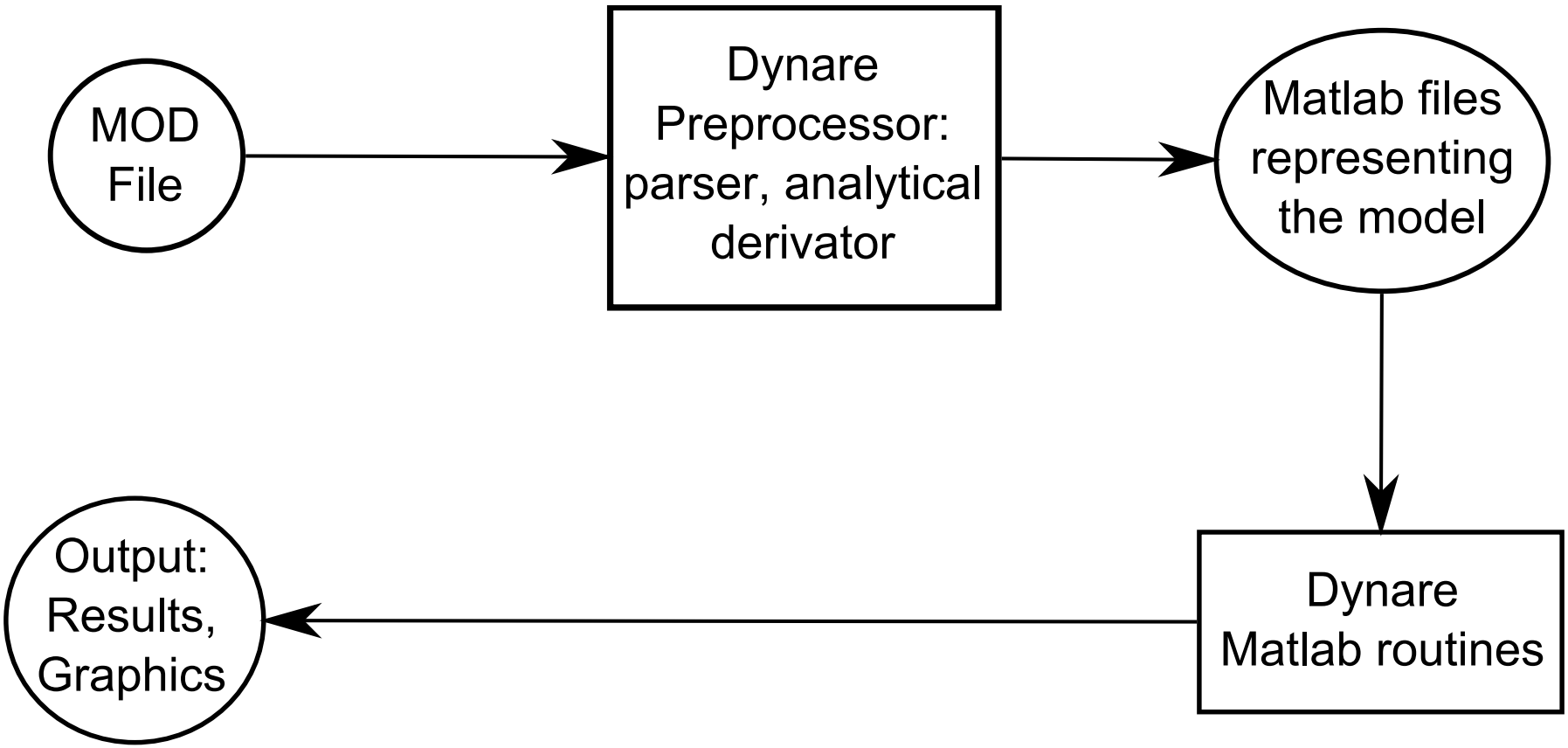
Motivation

- The **Dynare language** (used in MOD files) is well suited for describing economic models
- However, 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

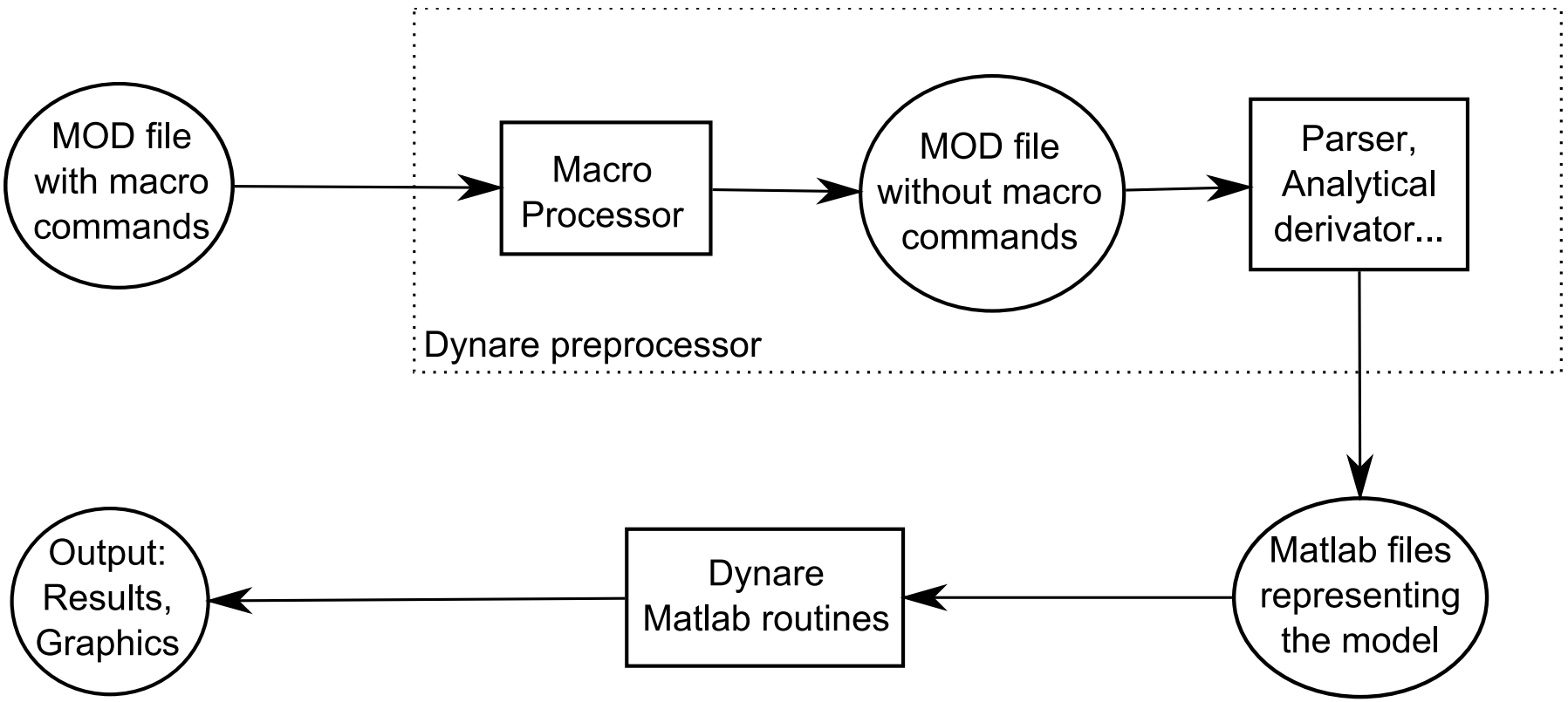
Design of the macro-language

- The Dynare Macro-language provides a new set of **macro-commands** which can be inserted inside MOD-files
- Language features include:
  - file inclusion
  - loops
  - conditional inclusion (if/then/else structures)
  - expression substitution
- 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 (@#) and occupy exactly one line
- However, a directive can be continued on next line by adding two anti-slashes (\\) at the end of the line to be continued
- A directive produces no output, but serves to give instructions to the macro processor

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"
```

Note that it is possible to include a file from an included file (nested includes).

Variables

- The macro processor maintains its own list of variables (distinct of model variables and of Matlab variables)
- 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]`

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
@#define y = "foo"
@#define v = [ 1, 2, 4 ]
@#define w = [ "foo", "bar" ]
@#define z = 3+v[2]
```

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} = K_@{country}^a * L_@{country}^(1-a)
@#endfor
end;
```

Example: after macro-processing

```
model;
  GDP_home = K_home^a * L_home^(1-a);
  GDP_foreign = 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 = ...
...
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`

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`)

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", "EU", "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 (1/3)

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

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

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

- During simulation or estimation, the share parameter α is a parameter, and *lab_rat* is an endogenous variable
- But for steady-state calibration, we may want to impose an economically relevant value for *lab_rat*, and deduce the implied value for α

  $\Rightarrow$ during calibration, α is endogenous and *lab_rat* is a parameter

Endogeneizing parameters (2/3)

- Create `modeqs.mod` with 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
```

- Create `steady.mod`:
  - begins with `@#define steady = 1`
  - then with `@#include "modeqs.mod"`
  - initializes parameters (including `lab_rat`, excluding `alpha`)
  - computes steady state (using hints for endogenous, including `alpha`)
  - saves values of parameters and endogenous at steady-state to a file

Endogeneizing parameters (3/3)

- Create `simul.mod`:
  - begins with `@#define steady = 0`
  - then with `@#include "modeqs.mod"`
  - loads values of parameters and endogenous at steady-state from file
  - computes simulations
- *Note:* functions for saving and loading parameters and endogenous are not yet in Dynare distribution (they should be soon, ask me if you're interested)

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
- Runs on MS Windows, Linux and MacOS
- Advantages:
  - mostly compatible with Matlab: same syntax, almost same set of functions
  - free software, no license needed
  - source code available
  - software under constant development
  - dynamic and responsive community of developers
- Inconvenients:
  - slower than Matlab
  - less user-friendly (no fancy graphical user interface)

Dynare for Octave (2/2)

- Small adjustments have been made in recent versions of Dynare to make it run on Octave
- This makes Dynare 100% free software
- If you're interested in using Dynare for Octave, go to: `http://www.cepremap.cnrs.fr/DynareWiki/DynareOctave`
- Adapting Dynare for Octave is still a work in progress
  $\Rightarrow$ feedback is welcome