modBuilder

stephane.adjemian@univ-lemans.fr

September, 2026

Introduction

- `modBuilder` is a MATLAB class aimed at simplifying the interactive and programmatic creation of Dynare `.mod` files.
- Build incrementally a model directly from the MATLAB.
- Facilitates consistency checks, error detection, and modular model development.
- Also provides tools for computing the steady state: solving equations one at a time, or deriving the whole steady state structurally.
- Export the model to a `.mod` file ready to run, or to LaTeX.
- Code available in a public Git repository (not part of Dynare).

Instantiate an empty model

```
>> model = modBuilder()

model =

  modBuilder with properties:

       params: {0x4 cell}
       varexo: {0x4 cell}
          var: {0x4 cell}
         tags: [1x1 struct]
      symbols: {1x0 cell}
    equations: {0x2 cell}
            T: [1x1 struct]
         date: 04-Oct-2025 09:39:05
```

Instantiate a model from a .mod file – I –

```
1  var a b y c h k;
2
3  varexo e (long_name='Productivity shock innovation')
4    u (long_name='Preference shock innovation')
5    ;
6
7  parameters alpha $\alpha$
8    rho $\rho$
9    tau $\tau$
10   beta $\beta$
11   delta $\delta$
12   psi $\psi$
13   theta $\theta$
14   phi $\phi$
15   ;
16
17 alpha = 0.360000; rho = 0.950000; tau = 0.025000;
18 beta = 0.990000; delta = 0.025000; psi = 0.000000;
19 theta = 2.950000; phi = 0.100000;
```

Instantiate a model from a .mod file – II –

```
model;

[name = 'a']
a = rho*a(-1)+tau*b(-1)+e;

[name = 'b']
b = tau*a(-1)+rho*b(-1)+u;

[name = 'y']
y = exp(a)*k(-1)^alpha*h^(1-alpha);

[name = 'c']
k = exp(b)*(y-c)+k(-1)*(1-delta);

[name = 'h']
c*theta*h^(1+psi) = y*(1-alpha);

[name = 'k']
1/beta = exp(b)*c/(exp(b(1))*c(1))*(1-delta+alpha*exp(b(1))*
    y(1)/k);

end;
```

Instantiate a model from a .mod file – III –

```
>> model = modBuilder('rbc12.mod')

model =

  modBuilder with properties:

       params: {8x4 cell}
       varexo: {2x4 cell}
          var: {6x4 cell}
         tags: [1x1 struct]
      symbols: {1x0 cell}
    equations: {6x2 cell}
            T: [1x1 struct]
         date: 04-Oct-2025 09:25:42
```

- No Dynare run, no JSON file: the `.mod` file is read directly.
- The other route stays available when Dynare has already run the file: `modBuilder(M_, oo_, 'model.json')` builds the model from what the session holds.

Instantiate a model from a .mod file – III bis –

Reading a .mod file writes a modBuilder script and runs it. Keep the script and carry on from there:

```
>> model = modBuilder('rbc12.mod', Script='rbc12.m');
```

```
% modBuilder script generated from rbc12.mod. Edit freely.

m = modBuilder();

% Equations
m.add('a', 'a = rho*a(-1)+tau*b(-1)+e');
m.add('y', 'y = exp(a)*k(-1)^alpha*h^(1-alpha)');
...
% Calibration
alpha = 0.360000;
...
% Parameters
m.parameter('alpha', alpha, 'texname', '\alpha', 'declared',
    true);
```

Instantiate a model from a .mod file – macros –

The macro language is read in full; its control flow survives into the script.

```
@#define Countries = ["US", "EA"]
model;
@#for c in Countries
[name = 'y_@{c}']
y_@{c} = alpha + e_@{c};
@#endfor
@#if OpenEconomy
[name = 'nx']
nx = y_US - y_EA;
@#endif
end;

```

becomes

```
Countries = macroarray('US', 'EA');
m.add('y_$1', 'y_$1 = alpha + e_$1', Countries);
if OpenEconomy
    m.add('nx', 'nx = y_US - y_EA');
end

```

Instantiate a model from a .mod file – macros II –

- Covered: `@#define` for variables and functions, with Dynare's own scoping; `@#if`, `@#ifdef`, `@#ifndef`; `@#for` over tuples and with a `when` guard; `@#include`; comprehensions, casts, set operations and ranges; `@#echo` and `@#error`; continuation lines. Dynare's own macro-processor self-test loads end to end.
- A loop becomes one implicit-loop call when a `$N` template reproduces every iteration, and a MATLAB `for` loop otherwise. The template is guessed and then verified against the iterations, so a wrong guess is discarded rather than emitted. The loop runs over the local its set became, `Countries` above, so editing that local changes the model; a `when` guard becomes a `filter`.
- A `@#if` becomes a MATLAB `if/elseif/else` carrying every branch. A branch that is not selected produces no text, so the file is read again with the conditional forced onto it; the model itself always comes from the unforced read. Changing the flag in the script then gives the model the `.mod` file would have given with that setting.

Instantiate a model from a .mod file – macros, -D –

A flag with a default, that Dynare's -DOpen=true overrides:

```
@#ifndef Open
@#define Open = false
@#endif
...
@#if Open
c = y - e;
@#else
c = y;
@#endif

```

Defines= is -D. The flag is a local of the script, at the value the read used, and the if keeps both branches:

```
>> model = modBuilder('open.mod', Defines=struct('Open', true));

Open = true;
...
if Open
    m.add('c', 'c = y - e');
else
    m.add('c', 'c = y');
end
```

Instantiate a model from a `.mod` file – what is read –

- **Imported:** the declarations, the calibration, the `model` block with its tags and its # model-local variables, `steady_state_model`, `initval`, and the options of `steady` and `check`.

- **Skipped, with a warning:** everything computational (`stoch_simul`, `estimation`, `shocks`, ...) and every native MATLAB line, told apart by Dynare's own rule: a line whose first token is neither a keyword nor a declared symbol is MATLAB. A native assignment that a calibration or an initial value uses is kept as a local of the script. `Strict=true` turns the warnings into errors.

- **Refused**, because skipping them would silently build a different model: `predetermined_variables`, `varexo_det`, trend variables, external functions, the optimal-policy statements, a second `model` block.

Of the 280 files of Dynare's test suite that use the macro language, 222 load as they are. Most of the rest hit a refused statement or an operator the parser does not know yet ($\perp$); a few expect a macro variable that Dynare's test runner sets with `-D`, and load once `Defines=` sets it.

Instantiate a model from a .mod file – naming equations –

A `modBuilder` equation is keyed to one endogenous variable. A `name` tag that is a declared variable settles it. Without tags the reader pairs equations and variables itself (next frame), and when no pairing covers every variable, the error names what to tag. Here neither equation mentions c:

```
1  var y c;
2  varexo e;
3  parameters alpha;
4  alpha = 0.5;
5
6  model;
7  y = alpha*e;
8  y = 2*e;
9  end;
10
```

```
>> modBuilder('twoeqs.mod')
twoeqs.mod: unable to associate every equation with a unique
endogenous variable. Add a "name" tag to these:
  line 8: y = 2*e
  unmatched endogenous variables: c
```

Instantiate a model from a `.mod` file – pairing –

Every equation of a `modBuilder` model is the equation *of* one endogenous variable, and every variable has one. Without `name` tags the reader has to decide which is which: n equations, n variables, each equation to be given a distinct variable it can stand for.

An equation may stand for a variable it mentions. Two cases, decided on the static equation, leads and lags dropped and the result simplified:

1. the static equation pins the variable: it holds it outright, as `y = alpha*k(-1) + e` holds `y` and `k`, or as a multiple whose other factor cannot vanish, as `junk = 0.9*junk(+1)`, which reduces to $0.1\,\texttt{junk} = 0$ and fixes `junk` at zero;
2. the static equation leaves the variable free: an Euler equation reduces to $c^{-\sigma}(1-\beta R)=0$, which pins R and says nothing about c; and `Y/Y(-1) = g` reduces to `1 = g`, with `Y` gone. Both are still the equations of `c` and of `Y`, only weaker claims.

When an equation could stand for several variables, two preferences settle it: the variable on its left-hand side, if it is one of them; otherwise the variable that the fewest other equations could stand for, since a variable mentioned by one equation only must take that one or end up with none.

Instantiate a model from a `.mod` file – pairing, the choice –

Each possible pair is given a price: 1 when the static equation pins the variable, 2 when it leaves it free, less a discount for the left-hand side, plus a small surcharge when many equations want the same variable. Leaving an equation without a variable is priced above any pair. The cheapest complete assignment is then computed exactly; it is the problem of assigning n workers to n jobs at the least total cost, and MATLAB solves it (`matchpairs`). Good pairs are used wherever they can be, and give way only where the assignment as a whole needs it.

	y	k	i	junk
`y = alpha*k(-1) + e`	**1**	1		
`k = (1-delta)*k(-1) + i`		**1**	1	
`i = s*y`	1		**1**	
`junk = 0.9*junk(+1)`				**1**

The case of each possible pair, in bold the assignment chosen: the left-hand side settles the first three rows, and the last equation takes `junk`, which it pins at zero.

When no complete assignment exists, some variable appears in no equation left for it, the reader stops and lists the equations to tag rather than guess. A `name` tag always wins over the assignment.

Instantiate a model

- It is also possible to create a new model from a previously defined modBuilder object that has been saved to disk.
- Once a `modBuilder` object is instantiated, the model can be expanded or transformed by utilizing the methods provided by the class.
- In the end the model can be saved on disk for later usage (`save` method) or exported to a `.mod` file (`write` method).

Add an equation

```
>> model.add('y', 'y = rho*y(-1)-gamma*z+e');
```

- Each equation must be associated to an endogenous variable (`y`) that will be used to index the model object.

$\Rightarrow$ Each new equation adds a new endogenous variable to the model.

$\Rightarrow$ Removing the equation of `y` therefore says what becomes of `y` (next frame).

- All the other symbols (`rho`, `z`, and `e`), unless previously defined, are classified as unresolved.

- These symbols need to be typed before completing the model. This can be done by adding an equation for an endogenous variable (`z`) or explictly declaring a symbol as a parameter (`rho`) or an exogenous variable (`e`).

Remove an equation

An equation is the equation *of* a variable, so removing it decides the fate of that variable, and of the symbols only that equation used:

```
m.add('y', 'y = a*k(-1) + e');
m.add('k', 'k = (1-delta)*k(-1) + i');
m.add('i', 'i = s*y');
m.add('h', 'h = theta*y');
```

call	what happens	endogenous / exogenous
`m.remove('i')`	`i` is still used by the equation of `k`: it becomes exogenous, `s` goes	`y k h / e i`
`m.remove('h')`	`h` is used nowhere else: it goes, and `theta` with it	`y k i / e`
`m.rmflip('i', 'k')`	`i` stays endogenous, keyed to the former equation of `k`	`y i h / e k`

Which variable an equation belongs to is therefore not a label: it is what `remove`, `rmflip` and `exogenise` act on, and why a `.mod` file is read with its `name` tags, or paired as described earlier. To re-pair without removing anything, `reassign` cycles the association of several equations at once.

Tag equations

```
>> model.tag('h', 'kind', 'static');
>> model.tag('k', 'kind', 'euler');
```

- The first argument specifies the equation's name, the second argument denotes the tag's name, and the third argument represents the tag's value.
- An equation can have any number of tags added to it.
- A tag can be named anything, except for 'name', which is specifically reserved for linking equations to an endogenous variable.

Tag equations – an example –

Two sectors, `m` and `s`. Tags are set with implicit loops, and a tag value may use the index:

```
m = modBuilder();
m.add('Y_$1', 'Y_$1 = A_$1*K_$1(-1)^alpha*L_$1^(1-alpha)', ...
      {'m', 's'});
m.add('K_$1', 'K_$1 = (1-delta)*K_$1(-1) + I_$1', {'m', 's'});
m.add('C', 'C = Y_m + Y_s - I_m - I_s');
% parameters and exogenous variables declared as usual
m.tag('Y_$1', 'type', 'production', {'m', 's'});
m.tag('K_$1', 'type', 'accumulation', {'m', 's'});
m.tag('Y_$1', 'sector', '$1', {'m', 's'});
m.tag('K_$1', 'sector', '$1', {'m', 's'});
m.tag('C', 'type', 'aggregation');
```

- The tags of an equation are the fields of a structure: `m.tags.Y_m` holds `name 'Y_m'`, `type 'production'` and `sector 'm'`.
- The next frames use them to find equations, to extract a block, and to remove one.

Tag equations – find equations –

```
>> m.listeqbytag('type', 'production')
   % Y_m Y_s
>> m.listeqbytag('sector', 'm')
   % Y_m K_m
>> m.listeqbytag('sector', 'm', 'type', 'production')
   % Y_m
>> m.listeqbytag('type', 'acc.*')
   % K_m K_s
>> m.listeqbytag('type', 'production|accumulation')
   % Y_m Y_s K_m K_s
```

- Several criteria must all hold: the third call keeps only the production equation of sector `m`.
- A value is a regular expression matched against the whole tag value: `'acc.*'` matches `accumulation`, and `|` reads as "or".
- No match is an error, not an empty list.

Tag equations – select a block –

```
>> p = m.select('sector', 's');
>> p = m{bytag('sector', 's')};      % the same
```

- `select` returns a model made of the equations that match and of the symbols they use. Here `p` holds the equations of `Y_s` and `K_s`, the parameters `A_s`, `alpha` and `delta`, and the exogenous variables `L_s` and `I_s`.

- A variable whose equation is left out becomes exogenous in the selection: `m{bytag('type', 'production')}` keeps the equations of `Y_m` and `Y_s`, with `K_m` and `K_s` exogenous.

Tag equations – remove a block, write the model –

`rm` and `remove` take the same selector:

```
>> m.rm(bytag('sector', 'm'));        % equations left: Y_s K_s C
```

As on the frame on removal, `Y_m`, still used by `C`, becomes exogenous, while `K_m` and `A_m`, used only by the removed equations, go.

Tags are written to the `.mod` file, where Dynare accepts any tag, and read back with it:

```
[name = 'Y_m', sector = 'm', type = 'production']
Y_m = A_m*K_m(-1)^alpha*L_m^(1-alpha);

[name = 'C', type = 'aggregation']
C = Y_m + Y_s - I_m - I_s;
```

So `modBuilder('sectors.mod')` gives a model on which the same selections work, and tags written by hand in a `.mod` file serve the same purpose.

Type symbols

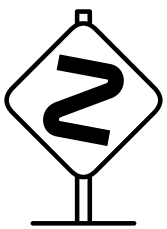
- `model.symbols` is the list of unclassified symbols.
- This property must be empty to finalize the model.

```
>> model.parameter('rho', .9, 'texname', '\rho');
>> model.exogenous('epsilon', .0);
```

- Calibration occurs upon declaring a parameter but can be modified subsequently.
- The value assigned to an exogenous variable determines its long-run level.

Converting symbols

- Methods `parameter` and `exogenous` can also be used to convert a symbol.
- These methods cannot be applied to an endogenous variable.
- To exogenize an endogenous variable one can:
  - Remove an equation (methods `rm` or `remove`)
  - Flip an exogenous variable and and endogenous variable (`flip` method).

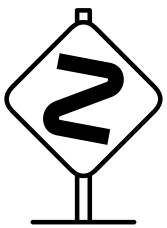


The method `endogenous` does not change the type of a variable, but only serves to provide the steady state level of an endogenous variable.

```
>> model.endogenous('y', .0);
```

Change an equation

- `change` method: replace a whole equation.
- `subs` method: replace an expression by another expression. Both arguments are parsed into AST trees, so the replacement is precedence-safe and lag-aware: substituting `mc` by `w/mpl` correctly rewrites `mc(-1)` as `w(-1)/mpl(-1)`.
- `substitute` method: regex-based rewrite on the equation text. Use when the target is a textual pattern (e.g. an indexed family) rather than a parsable expression. Warns when the target is a single symbol that `subs` would handle better.
- `rename` method: rename a symbol in all equations (preserves lag and `STEADY_STATE` wrapping).



If new symbols are introduced with the changes, they must be typed after.

```
>> model.change('y', 'y=rho*y(-1)+phi*y(-2)+e');
>> model.subs('y(-2)', 'y(-3)', 'y');
```

Search for symbols within a model

- The `lookfor` method searches for any symbol, such as a parameter or variable, within a model and displays the equations in which the symbol appears.

```
>> model.lookfor('k')

Endogenous variable k appears in 3 equations:

1/beta = ((exp(b)*c)/(exp(b(+1))*c(+1)))*(exp(b
(+1))*alpha*y(+1)/k+(1-delta))

y = exp(a)*(k(-1)^alpha)*(h^(1-alpha))

k = exp(b)*(y-c)+(1-delta)*k(-1)
```

Inspect a model

```
>> model.summary()

Model Summary
=============
Created: 16-Sep-2026 08:03:34

Parameters: 7 (7 calibrated, 0 uncalibrated)
Endogenous: 6
Exogenous: 2
Equations: 6

>> [type, id] = model.typeof('k')
type =
    'endogenous'
id =
     6
```

- `table('parameters')` returns the same content as a matlab table, with the long names and the TEX names; `equationmap` prints which equation each variable is keyed to, `size` counts, `getallsymbols` lists.
- `isparameter`, `isexogenous`, `isendogenous` and `issymbol` answer the same question as `typeof` with a logical. Reading a symbol is `model.alpha`, writing one is `model.beta = 0.95`.

Compose and compare models

```
>> sub = model{'y', 'c'}                          % same as model.extract('y', 'c')
sub =
  modBuilder: 2 endogenous, 4 exogenous, 2 parameters (2 calibrated), 2 equations

>> m2 = model.copy(); model == m2
ans =
  logical
   1

>> m2.beta = 0.98; model == m2
ans =
  logical
   0
```

- `extract` takes equations by name, `select` by tag – the two submodel constructors behind `m{...}` and `m{bytag(...)}`.
- `copy` is not decoration: `modBuilder` is a handle class, so `m2 = m` aliases the same model – every trial in `suggest_calibrations` runs on a copy.
- `==` compares structure and calibration, ignoring long names and TEX names: a refactor can be checked against a reference model.
- `merge` builds one model out of two, as the next frames do with two agents.

Use matlab to write an equation – I –

- The features of the MATLAB language, such as loops and conditional statements, can be employed to programmatically generate equations.

- Suppose we want to add the following equation:

$$y_t = \rho_1 y_{t-1} + \rho_2 y_{t-2} + \cdots + \rho_{12} y_{t-12} + \varepsilon_t$$

- Define a general MATLAB routine generating AR equations:

```
1  function eq = ar(maxlag, ename, pname, xname)
2      eq = sprintf('%s =', ename);
3      for l=1:maxlag
4          eq = sprintf('%s %s%u*%s(-%u) +', eq,
       pname, l, ename, l);
5      end
6      eq = sprintf('%s %s', eq, xname);
7  end
```

Use matlab to write an equation – II –

```
1  m = modBuilder();
2  m.add('y', ar(12, 'y', 'rho', 'e'));
3
4  for lag = 1:12;
5      m.parameter(sprintf('rho%u', lag), 2*rand-1)
6  end
7
8  m.exogenous('e', 0);
```

```
>> m.y.equations{2}

ans =

'y = rho1*y(-1) + rho2*y(-2) + rho3*y(-3) + rho4
*y(-4) + rho5*y(-5) + rho6*y(-6) + rho7*y(-7) +
rho8*y(-8) + rho9*y(-9) + rho10*y(-10) + rho11*y
(-11) + rho12*y(-12) + e'
```

Use matlab to write equations – III –

```
m = modBuilder();

COUNTRIES = ["FRA" "ITA" "GER"];

for c = COUNTRIES
    m.add(sprintf('y_%s', c), ...
          ar(12, sprintf('y_%s', c), ...
             sprintf('rho_%s_', c), ...
             sprintf('e_%s', c)));
end
```

Use matlab to write equations – IV –

Same with an implicit loop:

```
1  m = modBuilder();
2
3  COUNTRIES = {'FR' 'IT' 'DE' 'BE'};
4
5  m.add('y_$1', ar(12, 'y_$1', 'rho_$1_', 'e_$1'),
       COUNTRIES);
```

Use matlab to write equations – V –

Implicit loops can be nested:

```
1  m = modBuilder();
2
3  Countries = {'FR' 'IT' 'DE' 'BE'};
4  Sectors = num2cell(1:10);
5
6  m.add('Y_$1_$2', 'Y_$1_$2 = A_$1*K_$1_$2^alpha*
       L_$1_$2^(1-alpha)', Countries, Sectors);
```

$\Rightarrow$ The model now includes 40 additional equations! However, users still need to manually input the symbols introduced in the loops.

```
>> m.Y_FR_1.equations{2}
ans =
'Y_FR_1 = A_FR*K_FR_1^alpha*L_FR_1^(1-alpha)'
```

Steady state – the model –

```
% Instantiate an empty model
model = modBuilder();

% Set equations
model.add('a', 'a = rho*a(-1)+tau*b(-1) + e');
model.add('b', 'b = tau*a(-1)+rho*b(-1) + u');
model.add('y', 'y = exp(a)*(k(-1)^alpha)*(h^(1-
    alpha))');
model.add('c', 'k = exp(b)*(y-c)+(1-delta)*k(-1)
    ');
model.add('h', 'c*theta*h^(1+psi)=(1-alpha)*exp(
    a)*(k(-1)^alpha)*(h^(1-alpha))');
model.add('k', '1/beta = ((exp(b)*c)/(exp(b(+1))
    *c(+1)))*(exp(b(+1))*alpha*exp(a(1))*(k^(
    alpha-1))*(h(1)^(1-alpha))+(1-delta))');

% Define parameters and provide calibration
model.parameter('alpha', 0.36);
model.parameter('rho', 0.95);
model.parameter('tau', 0.025);
model.parameter('beta', 0.99);
model.parameter('delta', 0.025);
model.parameter('psi', 0);
model.parameter('theta', 2.95);

% Set default values for the exogenous variables
model.exogenous('e', 0);
model.exogenous('u', 0);
```

Steady state – declare it –

When the steady state has a closed form, declare it:

- The `steady` method attaches a steady-state expression to an endogenous variable. The expression may reference parameters and the steady state of other variables.
- `steady_aux` declares an intermediate quantity that is not a model variable (a ratio, a scale), `steady_call` registers a call assigning several variables at once.
- `checksteady` validates the declarations and returns them in dependency order, refusing unknown symbols and circular definitions.
- `write` emits them as a `steady_state_model` block, so that Dynare runs no numerical solve at all.

Steady state – declare it, the whole block –

For the model above, capital, output and consumption per hour follow from the Euler equation and the resource constraint, and hours from labour supply:

```
model.steady('a', '0');
model.steady('b', '0');
model.steady_aux('kh', '(alpha/(1/beta-1+delta))^(1/(1-alpha))');
model.steady_aux('yh', 'kh^alpha');
model.steady_aux('ch', 'yh - delta*kh');
model.steady('h', '((1-alpha)*yh/(theta*ch))^(1/(1+psi))');
model.steady('k', 'kh*h');
model.steady('y', 'yh*h');
model.steady('c', 'ch*h');
model.write('rbc', steady_state_model=true, steady=true);
```

- `checksteady` orders them `a b kh yh ch h k y c`; the ratios stay local to the `steady_state_model` block.
- Dynare runs the written file with no numerical solve: $h^\star = 0.2918$, $k^\star = 11.08$, $y^\star = 1.081$, $c^\star = 0.8036$.
- Two more `write` options: `check=true` adds Dynare's `check` after `steady`, and `precision=` sets how many digits the written calibration carries.

Steady state – compute it numerically –

When no closed form is at hand, two routes:

- Let Dynare solve the whole static system: give initial guesses, and `write` emits an `initval` block and the `steady` command (next frame).

- Or solve within `modBuilder`:
  - `evaluate` evaluates a static equation at the current values, and returns the left-hand side (`lhs`), the right-hand side (`rhs`) and the residual `resid=lhs-rhs`;
  - `solve` solves the static version of an equation for one symbol, an endogenous variable, an exogenous variable or a parameter;
  - `solve_system` solves several equations for as many symbols at once.

Steady state – let Dynare solve it –

```
% initial guesses, one per endogenous variable
model.endogenous('a', 0);
model.endogenous('b', 0);
model.endogenous('h', 1/3);
model.endogenous('k', 10);
model.endogenous('y', 1);
model.endogenous('c', 0.8);
model.write('rbc', initval=true, steady=true);
```

writes, after the model block (blank lines dropped):

```
initval;
  a = 0.000000;
  b = 0.000000;
  y = 1.000000;
  c = 0.800000;
  h = 0.333333;
  k = 10.000000;
end; // initval block
steady;
```

Dynare's `steady` solves the whole static system from these guesses, and here finds the point declared on the previous frames, to within $4 \cdot 10^{-7}$.
`steady_options` passes options to its solver, e.g. `{'maxit', 100}`.

Steady state – equation by equation –

```
model.endogenous('a', 0);
model.endogenous('b', 0);
model.endogenous('h', 1/3);

model.solve('k', 'k', 0.5);
model.solve('y', 'y', 1.0);
model.solve('c', 'c', 1.0);  % Correct steady state for consumption

model.solve('h', 'psi', 0); % Find the parameterization consistent with h=1/3
```

- Each `solve` handles one static equation for one symbol, simplified first; the order of the calls does the rest, as by hand.
- The equation of `k` needs `a` and `b`, which enter it through `exp(a)*exp(b)`, and their own two equations each need the other: one equation at a time cannot untangle them, so their level, zero, is given (`solve_system` can, next frame). `c` needs no value: it cancels from the static equation of `k`.
- The last call calibrates: with hours fixed at 1/3, the equation of `h` is solved for `psi`.

Steady state – a system at once –

```
% rough starting values; h holds its target, 1/3
model.endogenous('a', 0.1);
model.endogenous('b', -0.1);
model.endogenous('h', 1/3);
model.endogenous('y', 1);
model.endogenous('c', 0.8);
model.endogenous('k', 10);

model.solve_system({'a', 'b', 'y', 'c', 'k', 'h'}, ...
                   {'a', 'b', 'y', 'c', 'k', 'psi'});
```

- `solve_system` solves n static equations for n symbols jointly, by Newton's method from their current values; the symbols can mix endogenous variables, exogenous variables and parameters, here `psi`.
- Any subset of the equations will do, the other symbols held at their values: `solve_system({'a', 'b'}, {'a', 'b'})` solves the block that one equation at a time could not untangle, $a = b = 0$.
- Same point as the equation-by-equation route, $\psi = 0.1213$ and $k = 12.66$, within the default tolerance 10^{-6} (option `tol`).
- `Method` chooses how the Jacobian is obtained: automatic differentiation by default, or analytical partials, or finite differences.

Steady state plan – the idea –

- Writing the steady state by hand is the tedious part of building a model. The `steady_plan` method derives it instead.

- Three steps:
  - pair each equation with the variable it pins, by bipartite matching on the **static** residuals (option `Match`);
  - order the variables: an equation depends on the variables it mentions; variables whose equations need one another, in a loop, form a **block** solved together, and the blocks are taken in an order where each uses only what the previous ones gave;
  - solve each block by algebra: a single equation is rearranged for its variable; a small block is reduced by substituting its equations into one another. A linear block always yields, unless its determinant is zero; a nonlinear one does when the rearrangements succeed (listed later), and otherwise resists.
- What comes out is a **cascade of closed forms**, in evaluation order, and the list of variables (if any) that resisted.

Steady state plan – a small model –

```
% A small RBC model, without steady state.
model = modBuilder();

model.add('a', 'a = rho*a(-1) + e');
model.add('y', 'y = exp(a)*k(-1)^alpha*h^(1-alpha)');
model.add('k', 'k = (1-delta)*k(-1) + i');
model.add('i', 'y = c + i');
model.add('c', '1/beta = c/c(+1)*(alpha*y(+1)/k + 1 - delta)');
model.add('h', 'theta*c*h^psi = (1-alpha)*y/h');

model.parameter('alpha', 0.36, 'texname', '\alpha');
model.parameter('beta',  0.99, 'texname', '\beta');
model.parameter('delta', 0.025, 'texname', '\delta');
model.parameter('rho',   0.95, 'texname', '\rho');
model.parameter('theta', 2.95, 'texname', '\theta');
model.parameter('psi',   1.00, 'texname', '\psi');

model.exogenous('e', 0, 'texname', '\varepsilon');
```

Steady state plan – deriving it –

```
% Derive the steady state of the small RBC model instead of writing it.
rbc14                                       % the model, without steady state

model.calibrate('h', 1/3, 'theta');         % fix hours, free theta

b = model.steady_plan(Match=true, PropagateKnown=true);
model.print_steady_plan(b, Match=true, PropagateKnown=true);

model = model.apply_steady_plan(b);
```

- The level of `a` is not declared: the plan closes it from its own process, `a = rho*a(-1) + e`, as $a = e/(1-\rho)$.
- `PropagateKnown` inlines what is already known – here the calibrated $e = 0$ – before the recognisers run: $a = 0$, and `exp(a)` drops out of the closed forms of k and y.
- `calibrate` swaps a role: hours are fixed at $1/3$, θ becomes an unknown, and gets a closed form where `solve('h', 'psi', 0)` searched numerically.

Steady state plan – the plan –

```
Steady-state plan: 5 block(s)

  Block 1 [anchor]
    vars: a
    closed form: a = 0

  Block 2 [simultaneous, 2 vars]
    vars: y, k
    depends on (already solved): a
    external constants: h, alpha, delta, beta
    closed form: k = ((-1 + delta + beta ^ -1) / alpha * h ^ (-(1 -
     alpha))) ^ ((-1 + alpha) ^ -1)
    closed form: y = h ^ (1 - alpha) * k ^ alpha

  Block 3 [trivial]
    vars: i
    closed form: i = -(-k + k * (1 - delta))

  Block 5 [trivial]
    vars: theta
    closed form: theta = y * (1 - alpha) / c * h ^ (-1 - psi)
```

$\Rightarrow$ The calibration of θ consistent with $h^\star = 1/3$ is derived, not searched for.

Steady state plan – reading the plan –

The five blocks of the previous frame's plan, in the order they are solved:

1. `a`, from its own process, `a = rho*a(-1) + e`: $a = 0$.
2. `y` and `k` together. Once $c/c(+1)$ is one, the Euler equation reads $\alpha y/k = 1/\beta - 1 + \delta$, and the production function also links y and k: each equation needs the other. Substituting the production function into the Euler equation leaves one equation in k; y follows.
3. `i` from capital accumulation, $i = \delta k$.
4. `c` from the resource constraint, $c = y - i$.
5. θ from labour supply, hours being fixed at 1/3.

Blocks 3 to 5 are single equations, each rearranged for its variable.

A linear block always yields a closed form, unless its determinant is zero. The `a`, `b` process of the earlier model gives

$$b = \frac{\tau e + (1-\rho)\,u}{(1-\rho)^2 - \tau^2}, \qquad a = \frac{e + \tau b}{1-\rho},$$

whose denominator is the determinant of the block: the plan reports a unit root when it is zero at the calibration.

Steady state plan – the rearrangements –

A single equation is rearranged for its unknown x when, once simplified, x enters it in one of four ways, its leads and lags counting as x:

- linearly, $ax + b = 0$, so $x = -b/a$: $i = \delta k$ from $k = (1 - \delta)k + i$;
- as one power, $ax^d + b = 0$, so $x = (-b/a)^{1/d}$: k from $\alpha k^{\alpha-1} h^{1-\alpha} = 1/\beta - 1 + \delta$;
- as two powers, $c_p x^p + c_q x^q = 0$, so $x = (-c_q/c_p)^{1/(p-q)}$;
- inside `exp`, `log` or `sqrt`: the function is inverted first, `sqrt` by squaring, so $\log Z = \rho \log Z + e$ gives $Z = \exp(e/(1 - \rho))$.

When x sits in a denominator, the equation is first multiplied through by it. Anything else, $\log y + y^3 = x$ for instance, has no closed form: the block resists, and is left to a numerical solve (later frame).

Each case returns **one** closed form, never a set of roots: nothing branches, and no root is tested afterwards. Where several solutions exist the branch is fixed by convention: $(-b/a)^{1/d}$ is the principal root, so $x^2 = 4$ gives 2 and not -2; the two-power case drops the $x = 0$ that x^q factors out; `sqrt`, inverted by squaring, may return a root of the squared equation only. An assumption, then, not a verified selection: the residuals are the check.

A block of several equations is reduced one unknown at a time, each step using the same rearrangements. A linear block is solved outright, by triangulation and back-substitution, which fails only when its determinant is zero.

Steady state plan – the rearrangements, toy examples –

The rearrangement is `ast(residual).isolate(x)`, the equation being written as `residual = 0`. On toy residuals it returns:

case	residual	closed form returned
linear	`2*x - 6`	`x = 3`
	`a*x + b`	`x = -(b / a)`
one power	`a*x^d - b`	`x = (b / a) ^ (d ^ -1)`
two powers	`a*x^p - b*x^q`	`x = (a / b) ^ ((q - p) ^ -1)`
inside exp	`exp(x) - y`	`x = log(y)`
inside log	`log(x) - a`	`x = exp(a)`
inside sqrt	`a*sqrt(x) - b`	`x = (b / a) ^ 2`
own lag	`y - rho*y(-1) - e`	`y = e / (1 - rho)`
AR in logs	`log(Z) - rho*log(Z) - e`	`Z = exp(e / (1 - rho))`
denominator	`a/(x + 1) - b`	`x = (a - b) / b`
none	`log(y) + y^3 - x`	`[]`, no closed form

The AR process in logs is shown in its static form: the plan makes every equation static before rearranging it, so that `log(Z(-1))` reads as `log(Z)`.

Steady state plan – options –

- `Match` recomputes the equation/variable pairing instead of using the declaration order. A large simultaneous block usually falls apart into singletons.
- `Anchors` lists the variables whose level the modeller fixes – a normalisation, a scale, a calibration constant. Each anchor frees one equation, which becomes a consistency condition.
- `PropagateKnown` inlines every value already known, including the calibrated levels of the exogenous processes.
- `MaxBlockSize` caps the size of the blocks the symbolic machinery attempts. Elimination cost grows quickly with block size.
- `suggest_anchors` and `suggest_calibrations` propose the anchors, or the role swaps, that would close what resisted.

Steady state plan – writing it out –

`apply_steady_plan` turns the plan into steady-state declarations, which `write` emits as an analytical block:

```
steady_state_model;

  h = 0.33333333333333;
  a = 0;
  k = ((-1 + delta + beta ^ -1) / alpha * h ^ (-(1 - alpha))
    ) ^ ((-1 + alpha) ^ -1);
  y = h ^ (1 - alpha) * k ^ alpha;
  i = -(-k + k * (1 - delta));
  c = -(i - y);
  theta = y * (1 - alpha) / c * h ^ (-1 - psi);

end;
```

Steady state plan – blocks that resist –

- Not every steady state closes in closed form. When an elimination stalls, the plan reports the **reduced system** – the open variables and their residuals, with every closed form substituted in.

- `apply_steady_plan(GenerateHelpers=true)` then writes that system to a self-contained MATLAB routine, called from the `steady_state_model` block: analytic prologue, small numerical solve, analytic epilogue.

- The generated routine differentiates by **complex step**, $f'(x) = \Im\left(f(x + \imath h)\right)/h$ with $h = 10^{-20}$, exact to machine precision. Option `Jacobian` selects `auto`, `complex-step` or `symbolic`.

- Option `Solver` selects the damped Newton written into the file (`newton`) or the solvers shipped with Dynare (`dynare`).

Steady state plan – the complex step –

Take an analytic f, a real point x, and step in the **imaginary** direction:

$$f(x + \imath h) = f(x) + \imath h f'(x) - \frac{h^2}{2} f''(x) - \imath \frac{h^3}{6} f'''(x) + \dots$$

The imaginary part holds the derivative alone, the even-order terms being real:

$$\boxed{f'(x) = \frac{\Im\left(f(x + \imath h)\right)}{h} + O(h^2)}$$

Nothing is **subtracted**. A finite difference $(f(x + h) - f(x))/h$ differences two nearby numbers, so the digits they share are lost and h cannot go below the floor where that cancellation overwhelms the $O(h)$ truncation – around $\sqrt{\varepsilon} \simeq 10^{-8}$, leaving eight digits at best. Here there is no cancellation, so h can be taken far below that floor: at $h = 10^{-20}$ the truncation term is $O(10^{-40})$ and the derivative comes out exact to machine precision. On the residual of the next frame the complex step matches the symbolic derivative to 2 ulps, where the best forward difference gives $2 \cdot 10^{-8}$.

Steady state plan – the complex step, in the helper –

One residual evaluation per column, no derivative written to the file, and no dependency on modBuilder at run time – here the stalled block of the running example, whose single unknown is `i`:

```
function [r, J] = block_system(x, alpha, delta, beta, theta, phi, psi)
r = block_residual(x, alpha, delta, beta, theta, phi, psi);
if nargout > 1
    h = 1e-20;
    J = zeros(1, 1);
    for j = 1:1
        xp = x;
        xp(j) = xp(j) + 1i*h;
        J(:, j) = imag(block_residual(xp, alpha, delta, beta, theta, phi, psi))/h;
    end
end
end
```

The step needs f analytic **and** its MATLAB implementation to carry the imaginary part. It fails on `abs`, which returns the modulus and silently destroys it, on `sign`, `min`, `max`, not complex-differentiable, and on `normcdf`, `normpdf`, `erf`, `erfc`, `cbrt`, whose implementations reject a complex argument. Powers, `exp`, `log` and the trigonometric families are safe. `Jacobian=auto` tests the block and falls back to the symbolic Jacobian; asking for `complex-step` on a block that fails the test is an error, not a silent downgrade.

Steady state plan – when the plan stalls –

A fixed cost of production breaks the proportionality of output and consumption to hours:

```
rbc14                                          % the small RBC model
model.change('y', 'y = exp(a)*k(-1)^alpha*h^(1-alpha) - phi');
model.change('c', ...
    '1/beta = c/c(+1)*(alpha*(y(+1)+phi)/k + 1 - delta)');
model.change('h', 'theta*c*h^psi = (1-alpha)*(y+phi)/h');
model.parameter('phi', 0.1);
b = model.steady_plan(Match=true, PropagateKnown=true);
```

- The plan stalls: `h`, `k`, `i`, `y` and `c` form one block, four of them are expressed through `i`, and `i` is left to a numerical solve. The equation left for it combines several powers of `i` with the fixed cost, and none of the four rearrangements applies.
- Two ways out that stay symbolic: swap a role, pinning a variable and freeing a parameter (`suggest_calibrations`); or declare a value the modeller already knows, and let `suggest_anchors` say which ones the plan needs.

Steady state plan – suggest_calibrations –

First way out: swap a role, pinning a variable and freeing a parameter. On a stalled plan, `print_steady_plan` calls `suggest_calibrations` and prints the menu:

```
>> model.print_steady_plan(b, Match=true, PropagateKnown=true);
...
    residual (still open): i
    -- numerical solver required for the residual --
Suggested calibration swaps:
  m.calibrate('h', <target>, 'phi')    [closes fully]
  m.calibrate('h', <target>, 'theta')    [closes fully]
  m.calibrate('c', <target>, 'phi')    [closes fully]
```

- Each swap is tried on a copy of the model; only those that shrink what is left open are listed, the ones that close the plan first.
- The variable left open, `i`, has no parameter in its equation, the resource constraint: the scan then widens to every variable of the block.
- The menu is algebraic, not economic. Pinning hours and freeing θ is the textbook calibration: `model.calibrate('h', 1/3, 'theta')` closes the plan.

Steady state plan – suggest_anchors –

Second way out: declare values the modeller already knows, here from a numerical solve, and let the plan say which it needs:

```
model.steady('h', '0.5586029261');
model.steady('k', '21.2209081886');
model.steady('y', '1.9690980908');
out = model.suggest_anchors();      % anchors y, dropped h and k
b = model.steady_plan(Match=true, Anchors=out.anchors, ...
                      PropagateKnown=true);
```

- `suggest_anchors` tries dropping each declared value in turn, keeps only those the plan cannot do without, here output alone, and removes the declarations of the others (`Apply=false` only reports).
- With `y` as the only anchor, the plan gives `i`, `h`, `c` and `k` in closed form.
- An anchor frees one equation, which becomes a consistency condition the declared value must meet; it does here, the value coming from a numerical solve.

Steady state plan – homogeneous blocks –

- A block can pin the **ratios** of its variables without pinning their level: the Calvo price and wage recursions are the textbook case.
- Solving it as it stands would land on the trivial all-zero point. Instead the block is re-parametrised in one gauge, and the gauge is isolated from a consistency equation left over by the anchors.
- This is the hand technique of solving in ratios and backing the level out of a discarded equation – aggregate production for output, labour-market clearing for the wage recursions.
- Reported in the plan as `backout`, with the gauge and the equation that pinned it.
- `scaling_symmetries` finds these scale freedoms on its own, by homogeneity of the static system; the steady-state note covers it.

Symbolic derivatives

`partial` differentiates the residual of one equation; here the equation keyed to `c`, the capital accumulation, where `k` enters at two dates:

```
>> model.partial('c', 'k')              % static: all dates of k
ans =
  ast: delta
>> model.partial('c', 'k', Lag=0)       % the contemporaneous block
ans =
  ast: 1
>> model.partial('c', 'k', Lag=-1)      % the one-lag block
ans =
  ast: -1 + delta
```

- Omitting `Lag` makes the equation static first, so the leads and lags of the variable aggregate – the convention of `isolate` and of the AD Jacobian. Here $\delta = 1 + (\delta - 1)$.
- `symbolic_jacobian(eqnames, varnames)` returns the matrix of these partials as a cell of `ast` trees, sparse: `isempty(J{i,j})` is the test for a structural zero.
- Built on `ast.diff_ast` (see the AST note). This is the derivative of an equation, not the linearisation of the model – that is `tex_linearise`, later.

Optimal policy – the household –

The household chooses c, h and k under its budget, taking prices as given:

```
hh = modBuilder();
hh.add('W', 'W = log(c) - theta*h^(1+psi)/(1+psi) + beta*W(+1)');
hh.add('k', 'w*h + (r + 1 - delta)*k(-1) = c + k');
hh.exogenous('w', 1); hh.exogenous('r', 0.03);
foc = hh.lagrangian_foc('W', {'k'}, {'c', 'h', 'k'});
hh.augment(foc);                % declares mult_1, adds the FOCs
```

```
>> char(foc.foc)
    '-mult_1 + c ^ -1 = 0'
    '-(theta * h ^ psi) + mult_1 * w = 0'
    '-mult_1 + beta * mult_1(1) * (1 + r(1) - delta) = 0'
```

Marginal utility, labour supply, Euler. The derivation is written over **two periods**, as a Bellman problem, not as an infinite-sum Lagrangian: nothing assumes time-additivity, and that is why the multiplier carries a lead. `augment` declares `mult_1` and keys the three equations, leaving a square block of five.

Optimal policy – the firm –

The firm maximises profit over its two inputs. There is no constraint to name, and the empty list is accepted:

```
ff = modBuilder();
ff.add('Pi', 'Pi = A*kd^alpha*hd^(1-alpha) - w*hd - r*kd');
ff.exogenous('w', 1); ff.exogenous('r', 0.03);
ffoc = ff.lagrangian_foc('Pi', {}, {'kd', 'hd'});   % no constraint
ff.augment(ffoc);
```

```
>> char(ffoc.foc)
    '-r + A * alpha * hd ^ (1 - alpha) * kd ^ (-1 + alpha) = 0'
    '-w + A * (1 - alpha) * hd ^ (-alpha) * kd ^ alpha = 0'
```

The two factor demands: each marginal product equals its factor price. Nothing so far ties the firm's *kd* and *hd* to the household's *k* and *h* – that is what the equilibrium does.

Optimal policy – the equilibrium –

Four calls turn two problems into one model:

```
ff.flip('kd','r'); ff.flip('hd','w');    % each demand pins its price
ff.subs('kd','k(-1)'); ff.subs('hd','h');  % the markets clear
m = hh.merge(ff);          % 8 equations, 8 endogenous, 0 exogenous
m.eliminate('mult_1');  % substitute the multiplier out
```

```
>> m.equationmap()
    Endogenous                              Equation
    ----------    ------------------------------------------------------------
        W         W = log(c) - theta*h^(1+psi)/(1+psi) + beta*W(+1)
        k         w*h + (r + 1 - delta)*k(-1) = c + k
        c         -c ^ (-1) + beta * c(1) ^ (-1) * (1 + r(1) - delta) = 0
        h         -(theta * h ^ psi) + c ^ (-1) * w = 0
        Pi        Pi = A * k(-1) ^ alpha * h ^ (1 - alpha) - w * h - r * k(-1)
        r         -r + A * alpha * h ^ (1 - alpha) * k(-1) ^ (-1 + alpha) = 0
        w         -w + A * (1 - alpha) * h ^ (-alpha) * k(-1) ^ alpha = 0
```

The textbook decentralised model, derived rather than typed. No resource constraint was added: it follows from the budget and zero profits. `merge` returns a new model, refuses two models sharing an endogenous variable, and converts `h` and `k` – exogenous to the firm – into the household's endogenous variables.

Optimal policy – Ramsey –

`ramsey_foc` is the other route: every equation but the value equation is a constraint, and the planner sets the instrument.

```
nk.add('pi', 'pi = beta*pi(+1) + kappa*y');                % NKPC
nk.add('y',  'y = y(+1) - sigma*(i - pi(+1))');            % IS
nk.add('W',  'W = -(pi^2+lambda*y^2)/2 + beta*W(+1)');     % welfare
nk.exogenous('i', 0);                                      % instrument
nk.augment(nk.ramsey_foc('W', {'i'}));
```

```
>> nk.equationmap()
    Endogenous                              Equation
    ----------    ------------------------------------------------------------------
      pi          pi = beta*pi(+1) + kappa*y
      y           y = y(+1) - sigma*(i - pi(+1))
      W           W = -(pi^2 + lambda*y^2)/2 + beta*W(+1)
      mult_1      mult_1 - pi + (-(beta * mult_1(-1)) - mult_2(-1) * sigma) / beta = 0
      mult_2      mult_2 - kappa * mult_1 - lambda * y - mult_2(-1) / beta = 0
      i           mult_2 * sigma = 0
```

Six equations, no exogenous variable left: the instrument was promoted, and its own FOC pins $mult_2 = 0$. The FOCs are model equations like any other – write them, render them with `tex_model`, plan their steady state. Derive on a copy, `m.copy().ramsey_foc(...)`, to leave the nonlinear model untouched.

LaTeX – the model –

```
>> model.tex_model('rbcmodel.tex');
```

Uses the `texname` metadata declared with each symbol:

$$
\begin{aligned}
a_t &= \rho\, a_{t-1} + \varepsilon_t \\
y_t &= e^{a_t}\, k_{t-1}^{\alpha}\, h_t^{1-\alpha} \\
k_t &= (1-\delta)\, k_{t-1} + i_t \\
y_t &= c_t + i_t \\
\frac{1}{\beta} &= \frac{c_t}{c_{t+1}} \left(\frac{\alpha\, y_{t+1}}{k_t} + 1 - \delta \right) \\
\theta\, c_t\, h_t^{\psi} &= \frac{(1-\alpha)\, y_t}{h_t}
\end{aligned}
$$

LaTeX – the steady state –

- `tex_steady_state_system` renders the static system, every level starred.
- `tex_steady_state_plan` renders the derivation itself: the blocks, in order, with what each one closes.
- `tex_linearise` renders the model linearised around the steady state.

$$y^\star - e^{a^\star}\ {h^\star}^{1-\alpha}\ {k^\star}^{\alpha} = 0$$

$$\delta - 1 - \frac{\alpha\, y^\star}{k^\star} + \frac{1}{\beta} = 0$$

$$c^\star\, \theta\ {h^\star}^{\psi} - \frac{y^\star\ (1-\alpha)}{h^\star} = 0$$

LaTeX – the linearised model –

```
>> vars = {'a','y','k','i','c','h'};
>> model.tex_linearise(vars, LevelVars={'a'},
Evaluate=true);
```

$$
\begin{aligned}
(a_t - a^\star) &= 0.95\,(a_{t-1} - a^\star) \\
\hat{y}_t &= (a_t - a^\star) + 0.36\,\hat{k}_{t-1} + 0.64\,\hat{h}_t \\
\hat{k}_t &= 0.975\,\hat{k}_{t-1} + 0.025\,\hat{i}_t \\
\hat{i}_t &= 3.9001\,\hat{y}_t - 2.9001\,\hat{c}_t \\
\hat{c}_t &= -0.03475\,\hat{y}_{t+1} + 0.03475\,\hat{k}_t + \hat{c}_{t+1} \\
\hat{h}_t &= 0.5\,\hat{y}_t - 0.5\,\hat{c}_t
\end{aligned}
$$

- Each row is normalised on the variable its equation pins, so the coefficients read as elasticities.

- `LevelVars` marks the variables entering as $x_t - x^\star$ rather than $\hat{x}_t$, for a zero or negative steady state. `Evaluate=false` keeps the coefficients symbolic: $\hat{k}_t = (1-\delta)\,\hat{k}_{t-1} + \frac{i^\star}{k^\star}\,\hat{i}_t$.

LaTeX – the linearised model, symbolically –

```
>> model.tex_linearise(vars, LevelVars={'a'});
```

$$(a_t - a^\star) = \rho\,(a_{t-1} - a^\star)$$

$$\hat{y}_t = \frac{e^{a^\star}\;{h^\star}^{1-\alpha}\;{k^\star}^{\alpha}}{y^\star}\,(a_t - a^\star) + \frac{\alpha\, e^{a^\star}\;{h^\star}^{1-\alpha}\;{k^\star}^{\alpha}}{y^\star}\,\hat{k}_{t-1} + \frac{e^{a^\star}\;(1-\alpha)\;{h^\star}^{1-\alpha}\;{k^\star}^{\alpha}}{y^\star}\,\hat{h}_t$$

$$\hat{k}_t = (1-\delta)\,\hat{k}_{t-1} + \frac{i^\star}{k^\star}\,\hat{i}_t \qquad \hat{i}_t = \frac{y^\star}{i^\star}\,\hat{y}_t - \frac{c^\star}{i^\star}\,\hat{c}_t$$

$$\hat{c}_t = -\frac{\alpha\, y^\star}{k^\star\,\left(1-\delta+\frac{\alpha\, y^\star}{k^\star}\right)}\,\hat{y}_{t+1} + \frac{\alpha\, y^\star}{k^\star\,\left(1-\delta+\frac{\alpha\, y^\star}{k^\star}\right)}\,\hat{k}_t + \hat{c}_{t+1}$$

$$\hat{h}_t = \frac{y^\star\;(1-\alpha)\;{h^\star}^{-2}}{y^\star\;(1-\alpha)\;{h^\star}^{-2} + c^\star\,\psi\,\theta\;{h^\star}^{\psi-1}}\,\hat{y}_t - \frac{c^\star\,\theta\;{h^\star}^{\psi-1}}{y^\star\;(1-\alpha)\;{h^\star}^{-2} + c^\star\,\psi\,\theta\;{h^\star}^{\psi-1}}\,\hat{c}_t$$

$\Rightarrow$ Exact, but written in steady-state levels: the coefficients of the $\hat{y}_t$ row are 1, α and $1-\alpha$, and nothing says so.

LaTeX – elasticities in the deep parameters –

`Substitute=true` inlines the declared steady state into each coefficient and reduces it:

```
>> model.tex_linearise(vars, LevelVars={'a'},
Substitute=true);
```

$$
\begin{aligned}
(a_t - a^\star) &= \rho\,(a_{t-1} - a^\star) \\
\hat{y}_t &= (a_t - a^\star) + \alpha\,\hat{k}_{t-1} + (1-\alpha)\,\hat{h}_t \\
\hat{k}_t &= (1-\delta)\,\hat{k}_{t-1} + \delta\,\hat{i}_t \\
\hat{i}_t &= \frac{1-\beta+\beta\,\delta}{\alpha\,\beta\,\delta}\,\hat{y}_t - \frac{1-\beta-\alpha\,\beta\,\delta+\beta\,\delta}{\alpha\,\beta\,\delta}\,\hat{c}_t \\
\hat{c}_t &= -(1-\beta+\beta\,\delta)\,\hat{y}_{t+1} + (1-\beta+\beta\,\delta)\,\hat{k}_t + \hat{c}_{t+1} \\
\hat{h}_t &= \frac{1}{1+\psi}\,\hat{y}_t - \frac{1}{1+\psi}\,\hat{c}_t
\end{aligned}
$$

$\Rightarrow$ The investment share is δ, the Euler coefficient $1-\beta\,(1-\delta)$, the labour elasticity $1/(1+\psi)$: the textbook coefficients, derived. Among the algebraically equal forms of each coefficient, the one printed is the one `ast.complexity` scores as the simplest to read.

Examples – Smets-Wouters –

- `examples/sw` builds the nonlinear Smets-Wouters model: 78 equations, 78 endogenous variables, 17 shock processes generated rather than typed.
- `sw.m` writes the steady state by hand. Its companion `sw_from_anchors.m` derives it instead, from **seven** declared anchors – two normalisations, the efficient Tobin's Q, capacity utilisation and its efficient twin, the inverse markup, the hours scale.
- The other 54 expressions are derived, gauge backout included; the 17 shock levels are identified without being declared.
- The script then executes the emitted cascade and checks every equation at the recovered point.

Examples – two countries –

- `examples/two-country` is the other end: two RBC economies with perfect risk sharing, whose steady state is only **conditionally** analytical.
- The technology processes close, the coupled world core does not: the world resource constraint reduces to a sum of powers whose composite exponents defeat the recognisers.
- One variable is left open and handed to a generated routine; the other nine are closed conditional on it. Dynare then computes the steady state without a global numerical solve.
- Both examples carry a `README`, and neither commits a generated artefact: running them regenerates everything.