gamY - Documentation

Martin Kirk Bonde

January 2022

Introduction

gamY is a pre-processor for GAMS and implements additional features that are convenient for working with large models. The documentation assumes basic familiarity with GAMS.

Contents

Installation

gamY requires an installation of GAMS to be useful.
gamY comes with an executable, gamY.exe, which does not require any further installation. It can also be run as a python script which simply requires a python installation.

Running gamY

gamY can be used together with the GAMS-Python-API to run GAMS jobs directly from Python, or it can be run as a script run from a terminal, for example in a Windows .cmd file as shown in the box below.
When run as a script, gamY does 2 things:

1. gamY processes the input file, expanding all gamY commands, and outputs a file with plain GAMS code. To distinguish the input file and the expanded output file we use the extension .gmy for the expanded file and put it in a sub-directory called *'Expanded'*. The expanded .gmy file can be very verbose and should only be looked at for debugging purposes or to gain a better understanding of how gamY works behind the scenes.
2. gamY opens GAMS and tells it to execute the expanded GAMS file.

GAMS' save and restart feature can also be used with gamY (as shown in the box below). In that case, gamY automatically produces and reads a .pkl file containing metadata.

Running gamY from .cmd file

```
set gamY=Q:\gamY\gamY.exe
:: Run a .gms file using gamY
%gamY% test.gms
:: Run multiple .gms files with gamY using the GAMS save and restart feature
%gamY% a.gms s=a
%gamY% b.gms r=a s=b
```

To work offline, copy the gamY *executable "Q:\gamY*\gamY.exe" to your local project and replace "Q:\gamY\gamY.exe" with "gamY.exe" in the above.

General syntax

- Nothing is case sensitive, just like in GAMS.
- All gamY macros start with a "$".
- '#' is used for all comments instead of '*' or $ONTEXT.
- Square brackets are preferred for encasing sets, e.g. x[t] rather than x(t)
- Neither comments or macros have be at the beginning of a line (unlike GAMS dollar commands and '*' comments)
- Items can be separated by commas, line breaks, or a mix.
- gamY macros are processed before anything else in the program.

- Flow control statements (IF, LOOP, FOR, and FUNCTION) are processed before other macros.
- gamY macros can be nested.
- Nested flow control macros must be identified with a number, e.g.

```
$IF1 <condition>:
    $IF2 <condition>:
       ...
    $ENDIF2
$ENDIF1
```

Group

A gamY group is a data structure containing variables. New variables should always be defined using the $GROUP command rather than a GAMS *variables* statement. The group command bundles together variables so that they can be manipulated together more easily, using other gamY macros such as $FIX, $UNFIX, $LOOP, $DISPLAY, and even $GROUP itself. Variable elements can be selectively included in a group using dollar conditions. Note that conditions ALWAYS need to be enclosed in round brackets. Groups can be added together (union operation) or removed (complement operation) as shown in the examples below.

Syntax

```
$GROUP <group name> <groups and/or variables>;
```

Examples

Define new variables using the GROUP command

```
$GROUP group1
    var1[t] “label for variable 1”
    var2[a,t] “label for variable 2”
;
```

Existing groups or variables can be included in a group

```
$GROUP group2
    group1, var3 “Label for variable 3”
;
# Commas and line breaks can both be used to separate groups/variables
```

Groups can be re-defined (or modified by including the existing group in the definition)

```
$GROUP group2
    group2
    var4 “Remember labels for new variables!”
;
# Note that sets and labels are ignored for existing variables,
# but are often convenient to keep.
```

Variable can be conditionally included using dollar conditions

```
$GROUP group3
    var2[a,t]$(a.val >= 18) “label for variable 2”
;
# Note that conditions ALWAYS need to be enclosed in round brackets.
```

Group can be defined by ’subtracting’ a variable from the the group

```
$GROUP group4
    var2[a,t], -var2[a,t]$(a.val < 18)
;
# group4 is identical to group3
```

Dollar conditions can also be applied to entire groups

```
$GROUP group5
    group1$(t.val > 2019)
;
# Note that all variables in group1 must be defined over t in this case
```

Block

A block is a data structure containing equations. New equations should always be defined using the $BLOCK command rather than a GAMS *EQUATIONS* statement. The block command bundles together equations so that they can be manipulated together more easily, using other gamY macros such as $LOOP or $MODEL. Whenever a gamY block is defined, an equivalent GAMS model is also defined and can be used in solve statements (as shown in the example below).

Syntax

```
$BLOCK <block name>
   <equation definitions>
$ENDBLOCK
```

Example

```
$BLOCK MyBlock
    eq1[t].. v1[t] =E= v2[t];
    eq2[t].. v2[t] =E= 1;
$ENDBLOCK

solve MyBlock using CNS;
```

Model

The $MODEL command defines a new block data structure by combining existing blocks and/or equations. By default, blocks cannot be redefined. The GAMS options $onMultiR or $onMulti can be used to change this behavior.

Syntax

```
$MODEL <model name> <equations and/or blocks>;
```

Example

```
$MODEL MyNewModel
    MyBlock
    eq3
;

solve MyNewModel using CNS;
```

Loop

The $LOOP command is used to loop over a group or block, repeating an operation for each variable in the group or each equation in the block.
Within the loop, a number of special keywords are automaically replaced with fields from the variable or equation.
Looping over groups and blocks is described separately in the two following sections.

Looping over a group (or pgroup)

Syntax

```
$LOOP <group name>:
    <content where {name}, {sets}, and {conditions} are replaced>
$ENDLOOP
```

{name} is replaced with the name of the variable

{sets} is replaced with the sets that the variable is defined over

{conditions} is replaced with the dollar-conditions that limits which elements of a variable are included in the group being looped over. If no dollar-condition applies, it is replaced by (1) (i.e. *true*).

Example

```
$GROUP G_myGroup:
    v1[a,t]$(a.val > 15) “Variable defined over age and time”
;

$LOOP G_myGroup:
    parameter saved_{name}{sets};
    saved_{name}{sets}${conditions}) = {name}.L{sets};
$EndLoop

→

parameter saved_v1[a,t];
saved_v1[a,t]$(a.val > 15) = v1.L[a,t];
```

Looping over a block

Syntax

```
$LOOP <block name>:

   <content where {name}, {sets}, {conditions}, {LHS}, and {RHS} are replaced>
$ENDLOOP
```

{name} is replaced with the name of the equation.

{sets} is replaced with the sets that the equation is defined over.

{conditions} is replaced with the dollar-conditions that limits which elements the equation is defined for. If no dollar-condition applies, it is replaced by (1) (i.e. *true*).

{LHS} is replaced with the left hand side of the equation.

{RHS} is replaced with the right hand side of the equation.

Example

```
$BLOCK MyBlock:

    E_eq1[s,t]$(sp[s]).. v1[s,t] =E= v2[t];
    E_eq2[t].. v2[t] =E= 1;

$ENDBLOCK


$LOOP MyBlock:

    {name}_terminal{sets}$(tEnd[t] and {conditions}) {LHS} =E= {RHS};

$EndLoop


→

E_eq1_terminal[s,t]$(tEnd[t] and (sp[s])) v1[s,t] =E= v2[t];
E_eq2_terminal[t]$(tEnd[t] and (1)) v2[t] =E= 1;
```

Advanced set manipulation in loops

A number of advanced options are availiable for modifying the sets or applying conditions to sets in loops. These are useful for getting around the fact that we cannot for example apply a condition *$(t.val > 2019)* to a group where only some of the parameters are defined over t.
These features are not yet documented.

Fix

The $FIX macro is used to fix variables or groups of variables. By default, the variables are fixed to their current levels. Alternatively, a value can be supplied that the variables should be fixed to. All the same syntax for combining groups and variables that can be used with $GROUP macro can also be used with a $FIX macro.

Syntax

```
$FIX[(<optional value>)] <groups and/or variables>;
```

Examples

Fixing a group to its current level

```
$GROUP group1 var1[a,t] “Variable defined over age and time”;
$FIX group1;
→

var1.fx[a,t] = var1.l[a,t];
```

Fixing a group to a specified value

```
$FIX(100) group1;
→

var1.fx[a,t] = 100;
```

The same syntax can be used with a $FIX macro as with a $GROUP macro.

```
$GROUP group2 var2[t] "New variable";
$FIX group1$(a.val > 15), var2, -var2$(t.val > 2019);
→

var1.fx[a,t]$(a.val > 15) = var1.l[a,t];
v2.fx[t]$(not (t.val > 2019)) = v2.l[t];
```

Unfix

The $UNFIX macro is used to set the bounds of variables or groups of variables. By default, the lower and upper bounds are set to -inf and +inf respectively. Alternatively, a lower and upper bound value can be supplied which the variables' bounds should be set to. All the same syntax for combining groups and variables that can be used with $GROUP macro can also be used with a $UNFIX macro.

Syntax

```
$UNFIX[(<lower bound>, <upper bound>)] <groups and/or variables>;
```

Examples

Unfixing a group with no bounds

```
$GROUP group1 var1[a,t] "Variable defined over age and time";
$UNFIX group1;
→

var1.lo[a,t] = -inf;
var1.up[a,t] = inf;
```

Unfix a group with specified bounds

```
$UNFIX(0, inf) group1;
→

var1.lo[a,t] = 0;
var1.up[a,t] = inf;
```

Display

The $DISPLAY macro is used to display a group of variables in the listing fil (.lst file). All the same syntax for combining groups and variables that can be used with $GROUP macro can also be used with a $DISPLAY macro.

Syntax

```
$DISPLAY <groups and/or variables>;
```

Example

```
$DISPLAY group1$(2010 < t.val and t.val < 2020);
```

If

The $IF macro control which parts of a file is executed. Remember that gamY macros are executed before any GAMS code and gamY $IF statements therefore cannot use any parameters etc. in the condition (regular GAMS if statements can do that). The $IF macro is most useful in conjunction with the $SET or $SETGLOBAL macros. The condition can be be (almost) any statement that can be evaluated by Python. Nested if statements must be matched by adding a number to the matching $IF and $ENDIF macros.

Syntax

```
$IF[<id>] <condition>:
   <content>
$ENDIF[<id>];
```

Examples

```
$IF %i% = 1:
    display '1';
    ...
$ENDIF
```

Nested $IF statements

```
$IF1 %i% = 1:
    $IF2
       ...;
    $ENDIF2
$ENDIF1
# Nested if statements must be matched
# by adddding an identifying number to the matching $IF and $ENDIF macros.
```

Macro variables

The $SET, $SETLOCAL, and $SETGLOBAL commands from GAMS are also parsed by gamY and are used to define macro variables. Local macro variables take precedent over global variables, but are not remembered between files (using the save/restart feature). $SET and $SETLOCAL have identical behavior. Any names encased in '%' characters are replaced by gamY if the name is defined.
The $EVAL macros are identical to the $SET macros, except the assigned expression is evaluated before assignment.

Syntax

```
$SET[GLOBAL/LOCAL] <variable name> <value>

$EVAL[GLOBAL/LOCAL] <variable name> <expression>
```

Example

Setting and using a macro variable

```
$SETGLOBAL terminal_age 100
sets a "Age groups" /0*%terminal_age%/;

→

sets a "Age groups" /0*100/;
```

Using $EVAL to set a macro variable

```
$EVAL terminal_age_plus_1 %terminal_age%+1
set a "Age groups" /0*%terminal_age_plus_1%/;

→

set a "Age groups" /0*101/;
```

For loops

The $FOR macro is used to loop over arbitrary iterables that can be evaluated by Pyhon. Nested for loops must be matched by adding a number to the matching $FOR and $ENDFOR macros.

Syntax

```
$FOR[<id>] <iterators> in <iterable>:
    <expression>
$ENDFOR[<id>]
```

Examples

```
$FOR {parameter}, {value} in [('a', 1), ('b', 2)]:
    {parameter} = {value};
$ENDFOR

→

a = 1;
b = 2;
```

Import

The $IMPORT macro is equivalent to the the GAMS $INCLUDE command, except that the included file is also pre-processed by gamY.

Syntax

```
$IMPORT <file path>
```

Example

```
$IMPORT production.gms
```

User defined functions

Users can define their own functions by using the $FUNCTION macro. GAMS already allows users to define their own macros, but gamY user functions are useful as they allow for functions which utilize other gamY features. A function can be used to define other functions, however nested function definitions must be matched by adding a number to the matching $FUNCTION and $ENDFUNCTION macros.

Syntax

Function definition:

```
$FUNCTION[<id>] <function name>([<argument name>, <...>]):
    <expression>
$ENDFUNCTION[<id>]
```

Function call:

```
@<function name>([<argument>, <...>]);
```

Examples

Simple user function

```
# A regular GAMS macro could have been used instead
$FUNCTION abs({x}):
    sqrt(sqr({x}))
$ENDFOR

a[t] = @abs(b[t]);

→

a[t] = sqrt(sqr(b[t]));
```

User function using other gamY macros

```
# Set lower bounds to zero if the variable is not fixed
$FUNCTION zero_bound({group}):
    $LOOP {group}:

       {name}.lo{sets}$({conditions} and {name}.up{sets} <> {name}.lo{sets}) = 0;
    $ENDLOOP
$ENDFUNCTION

@zero_bound(group1);
```

Search and replace

The $REPLACE and $REGEX macros are used to do arbitrary text search and replace procedures. Optionally, a maximum number of replacements can be supplied. The $REGEX macro interprets the <old> string as a regular expression and allows for captured groups in the replacement string. See https://docs.python.org/3/library/re.html for details on the regular expression syntax.
Nested replacements must be matched by adding a number to the matching $REPLACE and $REPLACE macros.

Syntax

```
$REPLACE[<id>](<old>, <new> [,count])
   <expression>
$ENDREPLACE[<id>]


$REGEX[<id>](<old>, <new> [,count])
   <expression>
$ENDREGEX[<id>]
```

Examples

Simple replace

```
$REPLCE(“x”, “y”)
    x[t] = xy[t]
$ENDREPLACE

→

y[t] = yy[t];
```

Replace the first occurance only

```
$REPLCE(“x”, “y”, 1)
    x[t] = xy[t]
$ENDREPLACE

→

y[t] = xy[t];
```

Replacement with regular expressions

```
$REGEX(“(\w)\[”, “\g<1>.l[”)
    x[t] = y[t]
$ENDREGEX

→

x.l[t] = y.l[t];
```

Parameter groups

The $PGROUP macro is used to define and manipulate groups of parameters. p-groups are equivalent groups, except they contain parameters instead of variables. P-groups cannot be mixed with variable groups and they are not supported by other gamY macros, except LOOP.

Syntax

```
$PGROUP <pgroup name> <pgroups and/or parameters>;
```

Examples

Define new parameters using the PGROUP command

```
$PGROUP pgroup1
    par1[t] “label for parameter 1”
    par2[a,t] “label for parameter 2”
;
```

Implementation notes

The pre-processor is a prototype and uses 'brute force' regular expressions to find and replace the new macros. Macros can be nested and are processed inside out (using recursive descent parsing) from top to bottom. The program should be rewritten using a tokenizer, lexer, and parser to allow for unlimited nesting of flow control statements and to make the syntax more robust. Unittesting is not yet written.

Exercise 1 - A small Reform Model

The Model

This exercise shows how to setup and solve a small multisector model for a closed economy.[1] There are 5 sectors indexed by $j \in \{1, 2, ..., 5\}$. The production of sector j is given by $Y_j = F_j(M_j, L_j)$ where $M_j$is a material aggregate and $L_j$is labor. The production function $F_j(\cdot)$ is of the CES type. The aggregate M_j is made using CES technology with inputs being output from all other sectors $x_{i,j}$, where i denotes the supplying sector.

The demand for labor and materials in sector j is given by:

$$\theta_j L_j = \mu_j^{YL} \left(\frac{w}{\theta_j p_j^Y} \right)^{-E_j^Y} Y_j \tag{1}$$

$$M_j = \mu_j^{YM} \left(\frac{p_j^M}{p_j^Y} \right)^{-E_j^Y} Y_j \tag{2}$$

The demand for intermediate goods $x_{i,j}$ is:

$$x_{i,j} = \mu_{i,j}^{x} \left(\frac{p_i^Y}{p_j^M} \right)^{-E_j^M} M_j \tag{3}$$

Zero-profit conditions are:

$$p_j^M M_j + wL_j = p_j Y_j \tag{4}$$

$$\sum_i p_i^Y x_{i,j} = p_j^M M_j \tag{5}$$

The demand for goods and the consumer's budget constraint are:

$$C_j = \gamma_j \left(\frac{p_j^Y}{p_C} \right)^{-E_j^C} \frac{Y^D}{p_C} \tag{6}$$

$$\sum_j p_j^Y C_j = Y^D \tag{7}$$

Disposable income is:

$$Y^D = wN \tag{8}$$

Finally, there are two equilibrium conditions - one for the goods market and one for the labor market - but due to Walras's law we need only one. As such, we add the equilibrium condition for the goods market:

[1] The model is taken from an assignment for the course "Anvendt økonomisk modellering" (formerly "Anvendte Generelle Ligevægtsmodeller").

$$Y_j = \sum_i x_{i,j} + C_j \tag{9}$$

The model consists of equations (1)-(9) with the endogenous variables being $L_j, M_j Y_j, C_j, x_{i,j}, p_j^Y, p_j^M, w, Y^D$.

Initially, use the following parameter values: $\theta_j = 1, E_j^Y = 0.7, E_j^C = 0.5, E_j^M = 0.5$. Data for certain variables ($x_{i,j}, L_j, C_j$) is found in the input-output table IOdata4_2a.gms.

Setting up and solving Model

1. **Sets.** Before defining any variables or equations we need to define the sets which the variables/equations are defined over. In this simple model we need only one set as the content of the sets j and i are the same. Having created the set j the set i may be constructed using *alias(j,i)*.

2. **Variables.** Create 2 groups: One for the endogenous variables and one for the exogenous variables and define the relevant variables within the relevant group. That is:

```
$GROUP group_endo
    L[j]      "Labor input"
    w         "wage rate"
    .
    .
;
$GROUP group_exo
    theta[j]  "productivity"
    pC        "Price index"
    .
    .
;
```

3. **Equations.** Write up the equations of the model (i.e. equations (1)-(9)). These are are contained within a model block - for example:

```
$BLOCK mSector_model
    eq1[j].. theta[j] * L[j] =E= muYL[j] ... ;
    .
    .
$ENDBLOCK
```

4. **Data and initialization.** Set the values for exogenous parameters and variables regularly as in GAMS. Values for endogenous variables which are calibrated should also be set here. In the current model this involves $x_{i,j}, L_j, C_j$. In order to match these variables with the correct data from the IO-table in IOdata4_2a.gms, use a $FOR loop to loop over the correct set for each variable. For instance, for C_j we need to extract data from the IO-table for sets $j = 1, 2, ..., 5$ while keeping the second dimension (the output dimension of the table) constant at "private consumption". Having loaded the data only initialization remains. Set reasonable starting values for the endogenous variables (in the calibration - see the next step) to both improve the starting point of the solver (thus increasing the likelyhood that the model is solved) and to avoid "Division-by-zero" errors.

5. **Calibration.** Calibration is done a bit differently with gamY. Instead of calibrating "by hand", we use the numerical solver to calibrate for us. For example, to calibrate the model to produce certain consumption levels $C_{j=1} = 200, C_{j=2} = 100$ we would usually insert these values into eq. (6), and this would exactly provide us with the parameters $\gamma_{j=1}, \gamma_{j=2}$ that makes the model produce the desired, observed consumption levels. A similar procedure in gamY is done by setting $C_{j=1} = 200, C_{j=2} = 100$ in the "Data and initialization"-section and endogenizing $\gamma_{j=1}, \gamma_{j=2}$ and solving the corresponding model using GAMS (i.e. and endo-exo procedure). In practice this involves using the equation and variable blocks to produce a calibration model:

```
$GROUP group_calib_endo

    group_endo
    -C[j]
    gamma[j]

;
```

This procedure takes the existing group of endogenous variables, subtracts C_j (making it exogenous when solving), and adds γ_j (making it endogenous when solving).[2] Note that it might also be necessary to add new equations in the calibration-model. For instance, it is standard to assume that the parameters γ_j sum to one (thus making them shares). To account for this when solving the model using GAMS we add the following equation:

```
$BLOCK calibration_block

    eq1_calibration[j].. sum(j,gamma[j]) =E= 1 ;

$ENDBLOCK
```

This new calibration-block is then combined with the existing block of equations by using the $MODEL command:

```
$MODEL calibration_model

    mSector_model
    calibration_block

;
$FIX All; $UNFIX group_calib_endo;
solve Calib_model using CNS;
```

The last two lines first fixes all variables, endogenizes the variables of the calibration model, and solves the calibration model.

6. **Solving.** If the calibration was solved successfully, proceed and solve the actual model, composed of block of equations **mSector_model** and endogenous variables **group_endo**. Check that the results of solving the calibration model and the actual model are the same.

Shocks

1. Increase the productivity (θ) of sector 1 and 2 by 10%.
2. Change the elasticity of substitution of consumption from 0.5 to 2 (i.e. set $E_j^C = 2$). Increase again the productivity of sector 1 and 2 by 10%.

[2] When during this kind of calibration you will often run into pivot-errors ("Pivot too small"). A common source for this error is that all sides of an equation are entirely exogenized, thus rendering gams unable to solve the equation (even if it is satisfied by default). Example: Image a model equation stating that aggregate consumption equals the sum of components $C = \sum_j C_j$. Assume that aggregate consumption is always exogenous at a level of 1000. In the calibration we then proceed to exogenise C_j and endogenise γ_j. This is likely to generate a pivot error as both sides of the equation $C = \sum_j C_j$ are fixed. The solution usually is to leave one group endogenous (i.e. exogenise C_j for $j \neq 1$) so as to leave the equation solvable. Of course another model variable must then be endogenized to leave the model square.

3. Let the productivity of sector 2 decline by 10%. How much does the productivity in sector 1 need to increase to cause an increase in wages of 3%?