FARMDYN - a dynamic mixed integer bio-economic farm scale model

Coding and testing conventions[1]

Version 1.0, 13/09/16

Table of Contents

[1] Coding conventions are adapted from the CAPRI Coding conventions (ref.) for the use in FARMDYN

1. The Objectives

The aim of the FARMDYN GAMS coding convention is to motivate a coding style generating GAMS program code which:

- can be easily *understood* by another GAMS programmer
- can be successfully *maintained* and updated;
- and can source an *automated code documentation* system.

2. Naming conventions

Use clear and easy to understand names for symbols and files

A good name is self-explanatory, but short. Please keep in mind that the code basis of FARMDYN is very large, a name such as "*p_emissionFactor*" is still rather general (but clearly better than *"p_factor"* and much better than "*p_f*"). In doubt, ask a colleague not familiar with the problem you are working on if she or he is able to understand the chosen symbol names.

If a symbol name consists logically of several words, each new word except for the first one should start with upper case (we save space compared to using underscores). That so-called "**camelCase**" is a standard e.g. proposed in Java coding conventions:

```
parameter  p_manQuantMonth("cows25") "Manure quantitiy for cows25 in qubic metre";
parameter  p_price(inputs,t,s)       "Prices for inputs in €/unit";
```

Discouraged is the use of very short symbols where the meaning is not clear in the context, such as:

```
parameter p,i,s;
```

When introducing a new symbol, be sure that this symbol is not defined yet by checking through "search in file" in the GAMSIDE. Always be sure to add an explanatory long text to the declaration of a symbol including potential physical or monetary units if applicable to the symbol.

Note that vowels often can be dropped to shorten names, e.g. "*p_cnsQunt*" is almost as easy to read as *"p_consQuant"*. The use of "scientific" names such as *"p_alpha", "v_gamma*" etc. is discouraged for two obvious reasons. First, their meaning is far from clearly defined and highly context depending. Second, there is a huge danger that the very same symbol name is introduced somewhere else in the code, leading to possible conflicts.

Let equation names end with "_"

There is tradition in FARMDYN program to let equations end with an underscore.

Let parameter names start with "p_" and variables names with "v_"

That eases it considerably to read equations in a model or other GAMS statements as the GAMS notations is ambiguous in the sense that one cannot see what a parameter is and what a variable.

3. Usage of Sets

Sets are a central element of the GAMS language. They structure logically the code by spanning the "problem dimensions", such as time, products or processes. Set names should be clear, but generally short as otherwise, statements become very long.

Let Use domain checking wherever possible

Domain checking means that a symbol declaration in GAMS includes the information which sets are allowed on a specific dimension of a symbol, e.g.

```
p_syntAppLossShare(syntFertilizer,soil,till,intens,nutLosses)
```

Domain checking might be cumbersome to implement and might require the use of SAMEAS, but it can avoid terrible errors which are otherwise very hard to detect.

Use sub-sets wherever possible

Sub-sets are derived from other sets. They hence structure a domain clearly.

Don't declare the same collection of set members a second time

GAMS offers the so-called alias for that, the so far mostly used notation in FARMDYN in alias statements is to add a 1, 2 .., e.g.

```
set curBhkw(bhkw)                /
                                  set.selBhkws
                                 /;

alias(curBhkw,curBhkw1);
```

If you need the same collection in another set do allow for domain checking, use the possibility to introduce a complete set in a GAMS set declaration. It is proposed to use for sets which only used for that purpose the "SET_" notation is seen below, e.g.

```
set set_cows         / cows25*cows55,motherCow /;
set set_slgtCows     / slgtCows25*slgtCows55,slgtMotherCows /;
set set_mCalvs       / mCalvs,mCalvsSold,mCalvsRais,mCalvsRaisH /;
set set_fCalvs       / fCalvs/;
set set_heif         / heifsShort,heifsMiddle,heifsLonger,heifsLong  /;
```

4. Coding style and structuring

Declare symbols used in one file only at the top of that file

If the file is used in a loop or if statement, so that declaration in that file is not allowed, put the declarations into a separated file with "_decl" appended to the file name, and store it in the same sub-directory.

Generate Files with a clearly defined purpose

Each file should have clearly defined inputs and outputs, and especially the latter should form a logical unit. To give an example: a file which defines animal requirements should not as a kind of by-product correct herd sizes.

Avoid unnecessary deep include structures (<3)

Deep include structures require to open many files at the same time in the editor.

Statements

One declaration per line is recommended since it encourages commenting. In other words,

```
parameter p_level(domain1, domain2);
          p_size(domain3);
```

Is preferred over

```
parameter p_level(domain1, domain2), p_size(domain3);
```

Each line should contain at most one statement. Example:

```
example = example + 1;
```

And not

```
example = example + 1; p_thing("element") = NO;
```

Avoid lines longer than 80 characters, since they are not handled well by many terminals and tools.

Parameters used in the model should be defined in the model\templ_decl file.

Indentation and program flow structures

When an expression will not fit on a single line, break it according to these general principles

- Break after comma

- Break before an operator
- Prefer higher-level breaks to lower-level breaks
- Align the new line with the beginning of the expression at the same level on the previous line
- If the above rules lead to confusing code or to code that's squished up against the right margin, just indent 6 spaces instead

Loops and other program structures should be clearly visible by 3 spaces indentation:

```
loop(scen,
          statements in here must be indented to show the structure of this
          program
    );
```

$operators are preferred over IF statements

Good example:

```
p_parameter(herd)$ (p_otherparameter) = 10;
```

Find a compromise between the number of files included and their length.

Files should whenever possible not be longer than 1000 lines, but also should consists of more than 10 statements or so. A top level module should reveal its structure in the GAMS code.

$ONMULTI may be used only locally for well-motivated cases, followed by $OFFMULTI.

$ONMULTI allows for several declaration of the same symbol. That is really dangerous, as conflicting use of the same symbol might not be detected.

5. Use of $IF

For single lines $IFI and for multiple lines $IFTHENI should be used.

$IF is a compile time command, i.e. it is defining what pieces of the code are executed. The modular structure of FARMDYN allows to exclude/include certain modular elements depending on the initial GUI setup. These modular elements can be activated or deactivated via *iftheni* or *ifi* statements*:*

```
$iftheni.d %dairyherd%==true
                -sum((actHerds(slgtCows,breeds,feedRegime,t,m),cows)
                        v_herdStart(slgtCows,breeds,feedRegime,t,nCur,m))
$endif.d
```

Nested if statements should be indented by using a double dollar sign $$

```
$iftheni.herd %herd%==true
         v_herds(herds)
         $$iftheni.biogas %biogas%==true
         + v_biogas(biogas)
         $$endif.biogas
$endif.herd
```

***Iftheni and Ifi* statements should be tagged** by a corresponding modular element name of FARMDYN as seen in the above examples *.herd* or *.biogas*

6. Comments

GAMS code is computer code – it is not preliminary designed to provide easy to read text to humans. Indeed, it is often necessary to write of e.g. equations differently as they are documented in a paper to allow for an efficient use of GAMS. The meaning of the GAMS code is therefore often not immediately evident. Misinterpretation of the code however can provoke bad errors – somebody might change a statement as she or he has not clearly understood what the purpose is.

Comments, on the other hand, are directed towards our colleagues who want to understand the code – often, because there is the need to change or debug it. Comments should especially explain those things which are not easy to deduct from the code itself – they should not repeat the obvious, but motivate why a certain task is coded in a specific way. Comments also help us to quickly locate a statement or block of statements related to a specific task. Generally, comments are at least as important as the GAMS code itself.

Add clear and easy to understand comments to any GAMS code.

Try hard to write self-explaining code, but assume that it is not possible – hence add comments! Motivate and explain statements and code structure, instead of repeating what the code does again in plain English. Good code is like a good paper: it is structured such that the reader can easily follow the flow; comments support that. A typical example of a completely useless comment which does not add information is shown below:

```
*  --- Parameter p_herd is equal to p_herdSize
   p_herd(herd) = p_herdSize(herd);
```

Include *references* wherever possible to comments, e.g. to the methodological documentation or project deliverables. If the GAMS code is developed from a reference (e.g. the IPCC guidelines to structure GHG emissions), note the full reference and the page, so that the code can be verified quickly.

Comments are introduced in a separate line *above* the code to comment as seen in the upper example. The same indentation as the code commented upon should be used.

In order to structure a file in a logical manner block comments should be used

```
*---------------------------------------------------------------
*
*  Here comes the description of the block
*
*---------------------------------------------------------------
```

It is good coding style to insert a descriptive comment above an include statement

Introduce yourself and mark your code with your initials

Those who contribute a bit of code should label it with their name. We use pre-defined file headers (see next) where the name of the author(s) is one of the fields.

Generate a file header explaining the purpose of the file

Use the predefined template for doing so, so that the HTML based documentation can collect that information automatically. The following standard pieces of information should be included:

- Name of the author
- Name of the file
- Purpose of the file
- In case of a file used with "$batinclude": descriptions of the arguments

The screenshot below shows an example

```
*******************************************************************************
$ontext

   FARMDYN project

   GAMS file : GENERAL_HERD_MODULE.GMS

   @purpose  : Equations which are active if a herd is in the model
               (cattle, pigs), but are not specific to cattle or pigs

   @author   : Wolfgang Britz, building on revision 472
   @date     : 11.12.14
   @since    :
   @refDoc   :
   @seeAlso  :
   @calledBy :

$offtext
*******************************************************************************
```

7. SVN and Testing

The software versioning system SVN allows us to work efficiently as a distributed team of developers, especially to synchronize easily to the common established code base and to document changes to the code from version to version. Information on TortoiseSVN, the plug-in for Windows, can be found at http://tortoisesvn.tigris.org/.

Only commit fully functioning and tested code to SVN

Any exemptions must be made public beforehand and are subject to agreement of all others involved. That holds especially for the trunk. Any major changes, especially those leading to different results, should also be announced via the FARMDYN mailing list.

Accompany your commit with a clear description what was changed and why. If a whole block of files is subject to your change, commit them if possible together. Avoid committing whole bundles of unrelated changes with one commit.

If you introduce complex new features or refactor substantially existing code, provide a separate short technical note to be uploaded on the FARMDYN web site which describes the changes. Such a short note should comprise (1) a short *motivation* including references to project deliverables etc., (2) which *files* had been added (or changed), (3) a clear description of inputs and outputs, and (4) any unusual technical solution.

Update before committing

Make sure that you have updated the files you plan to commit, and do so before any tests, to make sure that you are testing the latest available version.

Compilation testing before every upload

Ensuring that the model compiles in all relevant settings (such as different modules switched on/off, different combination of farm branches) is crucial to prevent that colleagues run into errors due to changes by others. Hence:

- Run compilation batch file before every upload! It takes 2 min 30 sec and can be found in the GUI menu bar following GUI -> batch execution. Make sure that the batch file *test_batch_compilation.txt* is selected in the line *batch file to execute*. Every setting has to compile!
  If you only made minor changes and time is short – please run at least the execution batch in the compilation mode. It takes only 25 sec. To do so, you have to select the batch file *test_batch_execution.txt*.

Continuous Quality Management

There is an ongoing quality management carried out by student assistants in intervals of approximately two weeks. Student assistants will run the execution batch for relevant revisions and compare results. Generally, the aim is to identify unexpected changes in core variables, summarized in *p_sumres* that hint at errors occurring during the model development. p_sumres includes the variables like profit, number of animals or crop cultivation. Furthermore, student assistants make sure that the batch files are up-to-date.

To enable this quality check, developers have to stick to the following rules

- Mark in the *log* entry when uploading if a revision should be tested! When you upload changes from your working copy using SVN version control, mark if the revision should be tested. Write in the last line "[execution test: changes/no changes expected]", depending on if you expect your changes to impact on the core variables *p_sumres*.
- Check findings from the student assistants, summarized in *Tracking_Changes_Execution.xlsx*! The file can be found at SHK-Seaflie. Generally, you need to analyse if the changes in *p_sumres* are as expected from your changes in FARMDYN. If not, please look for possible mistakes. Student assistants will also contact you if they find any irregularities.
- Expand batch files! Relevant settings, that need to be included in the testing procedure after changes in the GUI, need to be added by the developer to *GUI**test_batch_execution.txt* and, particularly, to *GUI**test_batch_compilation.txt*. The batch files are constantly updated from the student assistants with regard to including new elements in existing listings of settings.

Good testing practices

The continuous quality management cannot replace testing procedures that should be done by every developer. It is difficult to define general testing procedures because they are highly depending on the changes you made and your level of understanding FARMDYN. The following steps could help you to test the changes you made.

- Make yourself familiar with dimensions and changes of core variables. The file *Tracking_Changes_Execution.xlsx* documents the values of *p_sumres* for all tested revisions. p_sumres is constantly displayed in the gams listing when running the model.
- Run the execution batch frequently and compare the results to results from older revisions, following guidance in Appendix 1 and 2. In the folder *XXYYY*, gdx files from older revisions are constantly saved. When you copy the file in the relevant \<*FarmDynFolderName*>*results**expFarms* folder, you can compare your working copy to older (and stable) revisions of FARMDYN.

Appendix 1: Running the execution batch for the first time:

Generate new .gdx instances for each scenario from the execution batch file by running the execution batch file via the Graphical User Interface -> GUI -> Batch execution. The generated results for each scenario can be found under ..\<*FarmDynFolderName*>*results**expFarms*. The batch execution can be found under ..*GUI**batch_execution.txt*.

Appendix 2: Compare the .gdx instances between the previous/first batch execution run and the batch execution after changes in the model:

The procedure to get differences between current and previous batch executions works as following:

1. When starting the execution batch after changes were made in the model, a new folder in *D:**temp*\<*date_time*> will be created. The name of the folder is created by using the date and the time of creation of the folder, e.g. *13.09.2016_12.31.29*. The folder starts with the

incgen_runIn_0_1.gms file, the listing file of the solve of the first scenario of the current batch run *0_1.lst* and the first scenario result .gdx file of the **previous** execution batch run.

2. After the first scenario is solved the *0_1.lst* file is used to create a new .gdx result scenario file which is stored in ..\<*FarmDynFolderName*>*results**expFarms*. The stored file overrides the scenario .gdx file from the previous run.
3. In the *D:**temp*\<*date_time*> a new file *diff_0_1* is created, which compares the differences between the new results .gdx file, stored in \<*FarmDynFolderName*>*results**expFarms*, and the old result .gdx file, which is in the *D:**temp* folder.

Steps 1.-3. are executed, within *D:**temp*\<*date_time*>, for each scenario which is defined in the execution batch file.