The red book on CAPRI GAMS coding

By Wolfgang Britz, Bonn University

Preface to 2016 edition

This document was originally created as a deliverable in the project CAPRI-RD[1]. The document was included into the CAPRI development trunk in 2016, after minor layout revisions. I tried to modify the headings of the document so that the table of contents reads as a numbered check list. I also replaced the red and green colours in order to make black and white prints of the text readable.

(Uppsala 2016, Torbjörn Jansson)

Table of contents

---

[1] Common Agricultural Policy Regionalised Impact – The Rural Development Dimension, a small to medium-scale focused research project under the Seventh Framework Programme Project No.: 226195

The objectives

The aim of the CAPRI 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.

The Java code conventions (http://java.sun.com/docs/codeconv/html/CodeConventions.doc) give the following reasons to establish coding conventions: "*Code conventions are important to programmers for a number of reasons:*

- *80% of the lifetime cost of a piece of software goes to maintenance.*
- *Hardly any software is maintained for its whole life by the original author.*
- *Code conventions improve the readability of the software, allowing engineers to understand new code more quickly and thoroughly.*
- *If you ship your source code as a product, you need to make sure it is as well packaged and clean as any other product you create."*

As CAPRI is indeed by now also a "software package" which is distributed to different teams and clients, the arguments above are also valid for us. Using code conventions is not "l'art pour l'art". Whoever has ever tried to work on program code which was coded by somebody else knows from own experience that unfortunate naming of symbols, missing or mis-guiding comments, bad structured code, highly personal coding style etc. can cost a lot of time and provoke terrible errors. It is highly egoistic to spare a few minutes by writing sluggish, un-documented code, and let others later deal with the problem to maintain it. The set of rather simple rules compiled in our guide supports us all to save costs and time, and to ensure that we can maintain in future the code of CAPRI.

Coding conventions in GAMS

Compared to other programming languages such as FORTRAN, PASCAL, C(++), Java or C#, GAMS does not break its code into functions and/or subroutines which clearly defined inputs and outputs. Equally, GAMS does not provide scoping for symbols: all GAMS symbols are known and accessible past the point where they had been declared; they have all global scope. Whereas coding conventions for most programming languages typically have a strong focus on modularisation of the code and clear scoping, we need to solve that issue for GAMS differently. Accordingly, naming conventions and clearly structured code are even more important in GAMS where every symbol has global scope!

Naming conventions

1. 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 CAPRI 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_data(rall,cols,rows,years) "Generic data cube of CAPRI";
  PARAMETER p_popGrowthRate(rall) "Population growth rate";
```

An exception can be made if the tokens already comprise acronyms in upper case so that reading becomes cumbersome:

```
😐 PARAMETER p_CAPMTRPolicy "Policy parameters for the MTR of the CAP"
```

In that case, it is better to use:

```
☺ PARAMETER p_CAP_MTR_policy "Policy parameters for the MTR of the CAP"
```

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

```
☹ PARAMETER i,p,q;
```

Please keep in mind that the very same name could be used by somebody else for a different symbol! If you introduce a new symbol, first use "search in files" from the GAMSIDE to make sure that the symbol name is not already in use.

Always add an explanatory long text to the declaration of symbols, if possible stating physical units or other elements helping to provide a clear definition:

```
☺ PARAMETER p_minFeedSharePerc(regions,animals,feed) “Minimum feed shares per region,
  animal and feed stuff in % of dry matter intake”
```

Bad is:

```
☹ PARAMETER p_minFeed;
```

As, (1) no domains are given, (2) the name is ambiguous (could be per animal, in a region …) and (3) an explanatory text is missing.

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

Tipp: “Find in files” from the GAMS IDE can be used to find all occurrences of a string over directories and files – easing dramatically the task to rename a symbol in a project.

2. Let equation names start with “e_”

There is tradition in CAPRI program to let equations end with an underscore which at least for old code can be kept.

3. Let parameter names start with “p_” and variables names with “v_”.

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

Parameters which are endogenous during calibration in equations should start with PV_, variables which are fixed during calibration should be start with VP_. Sets do not a have a prefix. The conventions should it make easier to understand what type of GAMS symbol is used.

4. Use clear and easy to understand codes for set elements

As for any GAMS symbol, take time to create set element names that indicate the meaning of the set. In that way the code becomes more self-documenting, and the risk of misuse is reduced.

5. Always add an explanatory text to set elements

Explanatory texts for set elements track the set throughout the code and into GDX-containers, and are therefore a good way of documenting the meaning of an element. Note: an explanatory text or comment does not replace a properly selected set element code.

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, space, products or processes. Set names should be clear, but generally short as otherwise, statements become very long.

6. 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_maxFeedShare(RALL,PACT,A,FEED)   "Maximum shares for each feedingstuff, expressed
  in dry matter"
```

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.

7. Use sub-sets wherever possible.

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

8. 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 CAPRI in alias statements is to add a 1, 2 .., e.g.

```
☺ ALIAS (regions, regions1, regions2)
```

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_FUELS / gasoline, diesel /;
  SET fuelRows(Rows) / set.SET_FUELS/;
  SET fuelCols(Cols) / set.SET_FUELS/;
```

That notation can also to be used to avoid repeating collections of set elements in sub-sets, e.g.

```
☺ SET SET_FINFUELS / gasoline, diesel /;
  SET SET_RAWFUELS / natGas, crudeOil/;
```

```
SET fuels / set.SET_FINFUELS, set.SET_RAWFUELS/;
SET finFuels(fuels) / set.SET_FINFUELS/;
```

Coding style and structuring

9. 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.

10. Separate processing code from data

Put the numerical data entering the code if possible in the relevant directory under "dat", and beyond a certain size, generate a GDX file from tables so that the GAMS code does not comprise an unnecessary high amount of code lines.

11. 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.

12. Avoid unnecessary deep include structures (> 3).

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

13. Use at most one statement per line

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(domain2,domain2), p_size(domain3);
```

Each line should contain at most one statement. Example:

```
☺ iTry = iTry + 1;

☹ iTry = iTry +1; RUNR(MS) = NO;
```

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

Indentation and program flow structures

14. Use indention to make code readable

When an expression will not fit on a single line, break it according to these general principles (from the Java coding conventions):

- Break after a 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.

15. Loop and other program structures should be clearly visible by 3 spaces indentation:

```
LOOP(RU,
        Statements in here must be identated to show the structure of the program
);
```

16. $ operators are generally preferred over IF statements:

```
☺ p_myParam(RU) $ (p_otherParam) = 10;
```

is preferred over:

```
☹ IF ( p_otherParam,
       p_myParam(RU) = 10;
  );
```

And certainly one should not use a loop as shown below – it is not only harder to read, but also slows down program execution:

```
☹ LOOP(RU $ otherParam(RU),
     p_myParam(RU) = 10;
  );
```

However, that is bad style to code as follows, as is not immediately visible that several assignments all depend on the same condition:

```
☹ p_myParam(RU) $ (p_otherParam) = 10;
  p_myParam1(RU) $ (p_otherParam) = 20;
  p_myParam(RU) $ (p_otherParam) = 30;
```

- Avoid unnecessary complex if and loop structures, or $-controls in statements.

17. Remove duplicate code by moving it to an include files.

18. Use $BATINCLUDE transparently

“Batcinlude” statements allow passing arguments to an included file. Inside the included file, the passed arguments are referred to with “%1, %2 etc.” according to the order they are handed over. It is extremely cumbersome to read such a program as “%6” is simply meaningless. That problem can be circumvented with the following coding trick which works as a rename:

```
☺ $setlocal regions %1

☺ p_myParam(%regions%) = p_someOtherParem(%regions%) …;
```

19. $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.

Use of $IF

$IF is a compile time command, i.e. it is defining what pieces of the code are executed.

20. $IF should always be replaced by $IFI – the not case sensitive version.

21. $IFI should only be used for single line statements:

```
☺ $IFI %MODE%==CAPREG $INCLUDE “capreg\someFile.gms”.
```

22. If several lines refer to the same $IFI statements, $IFHTENI … $ENDIF should be used.

Accordingly, avoid constructions such as:

```
☹ $IF %MODE%==CAPREG p_x(RS) = p_y(RS)
  $IF %MODE%==CAPREG   * p_o(RS)
  $IF %MODE%==CAPREG   * p_z(RS);
```

GAMS might treat the second line as a comment (it starts with a “*”)! There, according to the rule above, use:

```
☺ $IFTHENI %MODE%==CAPREG
     p_x(RS) = p_y(RS)
     * p_o(RS)
     * p_z(RS);
  $ENDIF
```

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

Files should whenever possible not be longer then 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.

Error trapping

Error trapping means that the code itself comprises tests which throw an error, instead of doing bad calculation due to missing or erroneous data or provoking run time errors. Imagine e.g. a program which works on market balances. Besides stock changes, all elements of the market balance are defined to be non-negative. Continuing with the code while trapping with $ and “if” statements negative market balance elements is probably the wrong tactic, as the results will anyway not make sense. It is hence good to test first if such logically nonsense data are present and then to stop execution and warn the program user about such errors.

24. Include tests of whether an include file does its job properly

All include files should (according to this red book) have just one well defined task. Try hard to include a test at the top of the file which raises an exception if necessary data is missing or does not satisfy some lowest standard.

Use %system%.fn and %system.incline% so that errors trapped provide information where the problem happens. Example:

```
☺ ABORT $ exceptionFilenameRegions "Error in %system.fn%, line %system.incline%:
  Population data missing for the following regions:", problemRegions;
```

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. Mis-interpretation 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.

25. Introduce yourself!

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.

26. 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 screen shot below shows an example

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

   CAPRI project

   GAMS file : FEDTRM.GMS

   @purpose  : Top level program of the feed distribution in CAPREG

   @author   : W.Britz, Institute for Agricultural Poilicy, University of Bonn
               M. Setti, G. Palladino, DIPROVAL Economics Unit, University of Bologna
               (requirement functions)

   @date     : 03.02.11
   @since    : 1999
   @refDoc   :
   @seeAlso  : feed\reqfnc.gms
   @calledBy : capreg.gms

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

27. Add clear and easy to understand comments to any not self-explaining 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:

```
☹ *     Set P_myParam to P_otherParam
  p_myParam(Domain) = p_otherParam(Domain);
```

Save others the time to deal which such useless comments.

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 (see also the section on meta data), so that the code can be verified quickly.

Comments are introduced in a separate line *above* the code to comment. The preferred standard style of a comment referring to a statement is shown in the following. The same indentation as the code commented upon should be used (i.e. if the code start in column 10, the "---" starts also in column 10):

```
☺ * --- Here comes the comment
```

Block comments should be used to structure a file logically into different sections:

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

It is good style to insert a comment above an include statements which briefly explains the purpose of the included file.

Meta data

Meta data in CAPRI follow an industry standard and are as far as possibly pushed automatically along the production chain, as well as integrated in the GUI. They allow inspecting of different types of information regarding the input data which had been use to produce some final or intermediate data or results. They are technically based on long texts stored with elements of a multi-dimensional set called META. Part of the meta data are automatically generated by the GUI when starting GAMS programs. It is therefore necessary to manually also change the content of the META set when programs are started outside the GUI!

28. Add meta data information to data and parameters.

If new data sources are included in the program, add meta information, and in case of updates, update both the numerical values and the meta information. The meta data should include the following standardized fields if possible:

```
SET META_ITEMS /
                    "Title of data set",
                    "Date of version",
                    "Abstract",
                    "Topic category"
                    "Key words",
                    "Temporal coverage",
                    "Language within the data set",
                    "Name of exchange format",
                    "Geographic coverage by name",
                    "Name of originator organisation",
                    "Name of owner organisation",
                    "Name of processor organisation",
                    "Description of process step"
                    BASEYEAR
                    SIMYEAR
                    MODEL_SWITCHES
                    WORKSTEP
                    KEY
                     /;
```

The meta data are stored as texts to set elements so that it can be passed along the production chain, as in:

```
SET META  /
 (SET.R_EU15).'Build regional database'.'REGIO database'.'NAME OF EXCHANGE FORMAT'          GDX
 (SET.R_EU15).'Build regional database'.'REGIO database'.'TITLE OF DATA SET'               'REGIO domain'
 (SET.R_EU15).'Build regional database'.'REGIO database'.'ABSTRACT'                        'NUTS II data on land use, crop areas and yields, herd sizes
 (SET.R_EU15).'Build regional database'.'REGIO database'.'TOPIC CATEGORY'                  'AGRICULTURE'
 (SET.R_EU15).'Build regional database'.'REGIO database'.'KEY WORDS'                       'Land use, crop areas, herd sizes, yields'
 (SET.R_EU15).'Build regional database'.'REGIO database'.'NAME OF ORIGINATOR ORGANISATION' EUROSTAT
 (SET.R_EU15).'Build regional database'.'REGIO database'.'LANGUAGE WITHIN THE DATA SET'    ENGLISH
 (SET.R_EU15).'Build regional database'.'REGIO database'.'GEOGRAPHIC COVERAGE BY NAME'      'EU at NUTS II level'
 (SET.R_EU15).'Build regional database'.'REGIO database'.'TEMPORAL COVERAGE'               1975 - 2004
 (SET.R_EU15).'Build regional database'.'REGIO database'.'DESCRIPTION OF PROCESS STEP'     'RAW STATISTICAL DATA'
 (SET.R_EU15).'Build regional database'.'REGIO database'.'NAME OF PROCESSOR ORGANISATION'  'ZINTL, EUROCARE BONN'
 (SET.R_EU15).'Build regional database'.'REGIO database'.'DATE OF VERSION'                 2006-12-05 11:05:59
/;
```

29. Load data and parameters wherever possible as GDX with META information included in the META set which is passed along the production line.

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/.

30. 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 CAPRI 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 CAPRI 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.

31. 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.