GAMS BEGINNER'S NOTES FOR CGE MODELING

Rob Davies and Dirk van Seventer

CONTENTS

INTRODUCTION

These notes were originally prepared by IFPRI and have been modified for the "Economic modelling techniques" course taught at the School of Economics, University of Cape Town, and for TIPS workshops. They are **not** intended to replace the GAMS user guide which you can access under `help>docs>gams`. You can also access a tutorial there which you should read. These replicate Brooke, Kendrick, Meeraus, and Raman (1998) GAMS: A User's Guide.

There is also an Expanded GAMS User Guide developed by Bruce McCarl available under Help. It is worth getting to know your way around this document. There are also plenty of resources for learning GAMS on the internet. Agrodep has a YouTube channel you can consult: https://www.youtube.com/user/AGRODEP

Other GAMS resources are available at www.gams.com. You will find tutorials and other useful material by using the YouTube link on the GAMS homepage. If you are serious about developing your skills as a modeller it is worth joining the GAMS listserve which can be done via the GAMS website. Many GAMS users benefit from this forum to ask questions and get advice from an international community of GAMS users.

- GAMS = General Algebraic Modelling System.
- GAMS was initially developed to run in DOS, and some users continue to prefer this mode. However, most now use the windows version, the GAMS Integrated Development Environment (GAMSIDE). These notes relate mainly to GAMSIDE.
- The GAMS Development Corporation now also releases GAMS Studio. Although this is not widely used yet, it is likely that it will replace GAMSIDE as the main development environment for GAMS modellers at some point in the future. It is intended to provide a better platform for incorporting new computing developments. We do not deal with GAMS Studio in these notes. As usual, there are excellent notes and guides on the GAMS website for those who wish to begin switching to it.
- GAMS can be used to solve a wide variety of model types, not only in economics.
- GAMS permits the disciplined modeller to follow 'good coding practices' which make the code accessible to anyone. These include:
    - any piece of data is entered only once and in its most elemental form;
    - explanatory text can be included;
    - everything needed to understand the model is in one file or set of linked files;
    - a given algebraic structure can be applied to highly aggregated or a highly disaggregated model – you only need to change the data;
- GAMS includes a model library with examples of different types of models;
- GAMS permits alternative user interfaces for inputting and outputting information, including reading and printing to spreadsheets.

PRELIMINARIES

GAMS Files

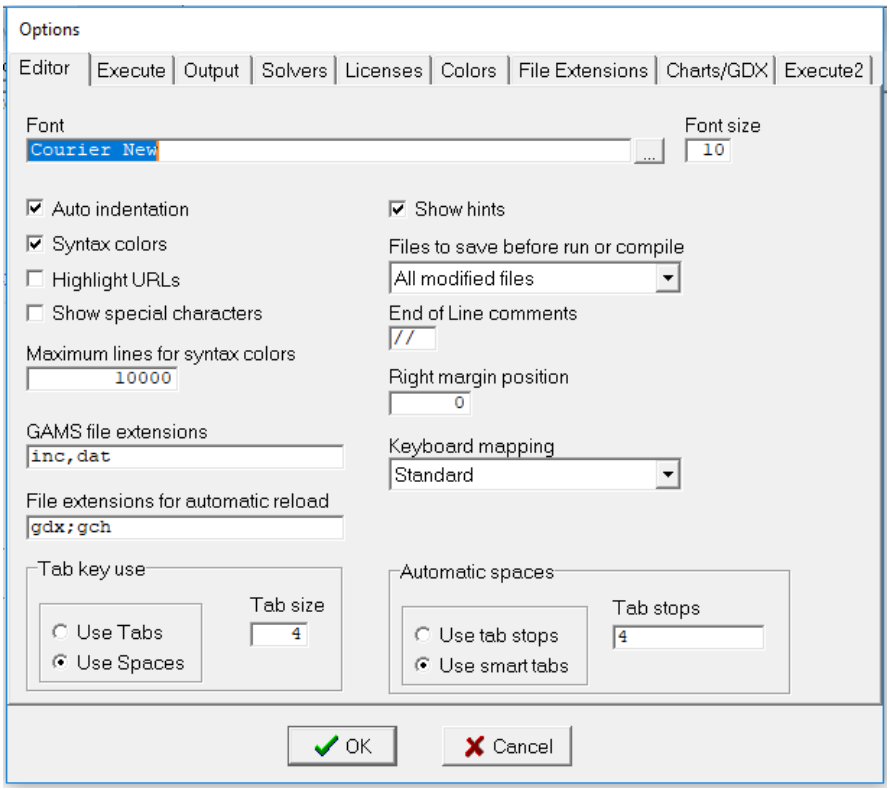
When you create a GAMS program, you will generate a number of different types of files, identified by their extensions. The common ones are:

- `.gpr` GAMS Project files
- `.gms` standard extension for GAMS codes
- `.inc` standard extension for Include files that may be called in during the running of a GAMS program
- `.lst` standard extension for Listing output
- `.log` Log files that keep the details of what went on while the program was attempting to solve.

Note that many of these extensions are designed to assist the user, not for GAMS itself. For example, although the core GAMS code is normally given a `.gms` extension, you could just as legitimately give a `.madiba` extension. It would still run. This is, in general, BAD modelling practice.

GAMS IDE Properties

You can modify the way GAMS looks by going to `file>options` and making the appropriate selections. You should adjust these for your own preferences, but we suggest:



- Editor
  - Font size 10

- Turn on `Auto indentation, Show Hints, Syntax colours`
- GAMS file extensions: `inc` (This ensures that GAMS syntax colours will operate in these as well as `.gms` files). If you want to add more file types here, separate them with a comma, but no space: `inc,dat`
- File extension for automatic reload: you can include extensions here for various results files so that yu do not have to open them manually each time you run your code. In this box multiple extensions have to be sparated by ";". We recommend adding `gdx`
- Tabs Options: `Smart Tabs`

- Output
  - Page height and Page Width – these depend on whether you plan printing hard copy of your code or not.
    - If you do, make sure page width is consistent with Right Margin Position and Font Size under <Editor> and that Page Height is the number of lines that you will fit on your printed page;
    - If you do not, leave Page Height at 0.
- Colours
  - You can choose the colours that GAMS uses to distinguish different types of code to suit your own preferences. Experiment with them. However, we recommend you leave them at the default setting, as it makes it easier for other GAMS programmers to read your code.

STARTING CODING

All new programs are started by specifying a project: `files>project>new project`. This creates a `.gpr` files that stores control options for the project (such as path names for data). Give your project a descriptive name (see below). We often prefix our project names with a 0 so that it is listed first in explorer.

You then open a new file: `file>new`. This will open a blank page in which you can start writing your code. It is a good idea to save this immediately, using `file>save` and giving it an approriate name. If you do not do this, you will end up with lots of untitled.gms files in the folder.

It is usual to save the first program as a `.gms` file.

If you want to run an existing program, you can rather follow the path `files> project> open project`. You can then select the appropriate `.gpr` file,

Naming GAMS files

GAMS IDE allows you to use most Windows file naming conventions. However, it can avoid problems later if you steer clear of long names with spaces in them.

Some General Tips on Coding

Compile frequently

Like many computer programs, GAMS programs run in "multiple passes" – running over the code several times. There are 3 main passes:

- **Compilation:** the code you have written is compiled—converted from the text you have written into machine code the computer can understand;
- **Execution:** all assignment statements are executed;
- **Model Generation:** all variables and equations in the model are generated.

After the third stage, GAMS sends your model to the solver you have selected, which solves it and sends the solution back to GAMS for further post-solve calculations and displaying

During compilation, the code is checked for syntax errors. These are errors relating to coding—wrong specification of functions, inconsistent spelling of variable names, etc. Code can be compiled at any stage—it does not have to be complete. This allows a very useful method of writing to be used. After writing a few lines, run the program. Any syntax errors will be discovered immediately. See the section "Fixing errors" below for further discussion.

You can compile without executing by `<shift+F9>` or holding down `<shift>` while clicking on the run button.

Note that this syntax checking facility simply checks that you have followed the language rules of GAMS. It does NOT check whether your model makes economic sense or even that it does what you want it to.

Annotate your Code

Use the comment options frequently to explain what you are doing. This will make the program more user friendly, helping not only other users, but also yourself several months later when you have forgotten why you did what you did.

Comments can be inserted in a number of ways. Two are most widely used:

a) A '`*`' in the first column of a line means that every thing in that line will be ignored. This is useful for short comments.

b) The command `$ontext` tells the compiler that everything that follows is text rather than code. It will ignore it until it comes across `$offtext`, which turns off the text command. **NB**: the $ must be in the first column of the line

There is are also ways of adding comments in the middle of or at the end of a line, but most programmers do not use this. See `$oneolcom` in the User Guide.

Do not always delete 'errors'

Sometimes you write several lines of code that do not work. Rather than deleting them all, try switching them off using the commenting options explained above. This way you can easily recover bits you need. However, it is a good idea to tidy up your code once the problems have been sorted out.

Take trouble to set our your code neatly

You can write code in pretty much any format you wish. However, if you take trouble from the outset to write it neatly, you will not only make it easier to read, but also save time later.

Compare

```
EGREP(SIM)          = EG.L;
EXRREP(SIM)         = EXR.L;
FSAVREP(SIM)        = FSAV.L;
IADJREP(SIM)        = IADJ.L;
MPSREP(H,SIM)       = MPS.L(H);
PAREP(A,SIM)        = PA.L(A);
```

with

```
EGREP(sim)=EG.L;EXRREP(SIM)=EXR.L;fsavrep(SIM)=FSAV.L;IADJREP(SIM)=
IADJ.L;MPSREP(H,SIM)=MPS.L(h); PAREP(A,SIM)=PA.L(A);
```

Fixing errors

Errors can—and will—come into your coding at various stages. We have mentioned syntax errors above. These are errors in the writing of the code and are shown up at the compilation stage. When you run a program, GAMS switches into a *process window*. If you have syntax errors, they appear in red, with a hint about what the problem might be. If you double-click on the red line, it will jump to the place in your code where the syntax error is.

Often there will be numerous syntax errors. You should start fixing the first one, because one error can cause subsequent ones. Say for example you have declared a parameter "`Income`". Later you define it by assigning a value, but you accidently mis-spell it and write `Incime= 10;` Then later in the program you use it in a formula, this time spelling it correctly: `consumption = 0.8*Income;` This will generate two errors, one indicating `Incime` is not recognised and the other indicating `Income` has not been assigned a value. Correct the spelling and both errors disappear. So always start at the beginning.

Once the syntax is correct and you run the model, you can get errors in the execution. Typical errors here are attempting to divide by zero or trying to raise a negative number by a fractional power, such as taking the square root of -2. GAMS tells you which line of the code is causing the problem. It is often difficult to see why the problem arises. If you have an equation that is solving for the value of 200 variables, it is not always obvious which instance is causing the problem. We can code GAMS to avoid some problems, such as division by zero, but searching for other errors can be a frustrating experience. As you become more experienced, you will learn how best to look for the cause of these.

Then there are problems that arise during the solution. For example, it is at this point that GAMS may tell you that the number of equations does not match the number of variables. GAMS gives you some assistance in finding these, but they depend on which solver you are using.

Of course, GAMS does not ever tell you that the economics of your model is wrong. If you specify a downward sloping supply curve by accident, GAMS cannot tell you to read an elementary textbook. So, just because your model "works" in that it solves and gives you some answers, that does not mean you are right. The process of interrogating and understanding your results is the only way you can pick up such errors.

Moving from Mathematical statements to GAMS code

One of the great advantages of GAMS is that you can write equations and formulae almost as you would in algebra. However, there are a few modifications to be aware of.

GAMS is not case sensitive

There is no difference between Q and q. If they mean something different in your algebra, you need to devise different names in GAMS.

However, you should use upper and lower case to make your code easier to read: the computer does not recognize the difference, but the human reader can. You can thus use conventions to make code easier to follow. For example, the IFPRI convention is to name variables in UPPER CASE and parameters in lower. Other conventions use lower case to refer to rates of change and UPPER to levels.

(If you want the echo print to distinguish upper and lower case so, put `$offupper` in column 1 in the beginning of the file.)

GAMS cannot distinguish subscripts

In algebra we can write X_{ij} . In GAMS we have to write `X(i,j)`.

GAMS cannot distinguish superscripts

In economics, we often use complicated looking combinations of symbols, such as QD_c^h to mean "the quantity of good c demanded by household h". In GAMS we have to be simpler. Since both c and h are indices, we could write this as `QD(c, h)`.

Since GAMS cannot recognise superscripts, we cannot write power functions normally. We use "`**`" to mean "to the power of".

GAMS cannot use Greek letters

In algebra we write β. In GAMS we would have to write 'beta' if we wanted an exact translation.

This is also the case where we use Greek symbols to mean some function in algebra. Thus, $\sum_{i=1}^{100} X_i$ in algebra is written in GAMS as **`SUM`**`(i, X(i)),` where `i` has previously been declared as the index for a set comprising the numbers 1 to 100.

Notice here that there are two pairs of parentheses – the first enclosing the components of the summation process, the second enclosing the index over which X_i is defined. All functions in GAMS require the entities over which they operate to be enclosed in parentheses. A common error arises from not closing off the parentheses. The GAMS IDE toolbar has a button and function key [F8] for checking whether parentheses match.

Similarly, the product function $\prod_{i=1}^{n} X_i \left(= X_1 \cdot X_2 \cdot \ldots X_n\right)$ has to be written **`PROD`**`(i, X(i) ).`

"=" does not mean "equals"

As in many programming languages, in GAMS "=" does not mean "equals" but is an assignment operator. The English for "Y = X + 5" is "let Y take on the value of X plus 5".

This means that you can only have one entity to the left of the "=". Also, you can write statements such as "`X = X + log(X)`".

Assignment statements ("=" on its own) are implemented **in order of appearance in the program**. Thus, if there are two lines

```
X =  100 ;
X = -200 ;
```

X will have the value –200: the **last** assignment statement will overwrite the earlier statement in which X had the value 100.

The equality operator in equations is written "=E=" for example

```
Y  =E=  X + 5.
```

With this statement GAMS will attempt to find a solution for Y given what is on the right hand side. X could be defined as above or perhaps also in an equation.

Editing options

If you are not using GAMSIDE, editing options will depend on the text editor you use. In GAMSIDE, you can use most of the standard Windows editing options. For example, <ctrl+C> = copy, <ctrl+V> = paste, <ctrl+X> = cut, <ctrl+F> = find, <ctrl+Z> = undo, etc..

Get used to using these.

A very useful option is holding down <shift+alt> while selecting a range allows you to select a block in the middle of the page. When you copy and paste such a block, it is inserted as an area on the page, rather than as new lines. Interestingly, not many people know that MSWord has the same function.

There are additional control keys in GAMSIDE that you can learn. For example <alt+U> makes the word the cursor is on uppercase, while <alt+L> makes it lower. Learning these can help you to speed up editing.

"Naming of Parts"

Although there are some restrictions on names you can use for sets, parameters, variables, equations etc, the choice is largely up to you. It pays to try to adopt a standard approach that helps you to follow your code easily – without having to look up names each time you want to use them. Thus, you might

- Distinguish common entities from each other: start quantity names with Q, commodities with C, prices with P etc
- Use acronyms. If you have 100 commodities, you can most easily name them C1, C2,…, C100. But this might make it difficult to remember what each one is, so most modellers try to make up names that remind them: CAGR, CFOOD, CMINE etc. If you do this,
    - spend some time thinking about it to make sure you do not run into problems later;
    - keep the acronyms the same length – it makes editing easier;
- GAMS names
    - must begin with a letter, but can include numbers and +, - and _ (underscore)
    - can be up to 63 characters long;

NOTES ON THE FORMAT OF GAMS INPUT FILES

GAMS programs generally comprise the following main elements or objects

- Sets
- Parameters (Parameters, Tables, Scalars)
- Variables
- Equations

- Model(s) statement
- Solve statements
- Display statements

General Comments

- Typographically, you can lay out the information in any style that you find appealing.
- It is recommended that you end each GAMS statement with a semicolon. They are not always required, but do no harm if redundant.
- The creation of GAMS objects (sets, parameters, variables, equations, models) involves two steps:

  1. Declaration: declaring the existence of something and giving it a name

  2. Assignment or definition: giving something a specific value or form

- For some of the objects, you can perform these two steps either separately or in one statement; for others, a specific format is imposed.
- You can enter information in any order as long as you do not refer to an entity before declaring its existence or use a parameter before it has been assigned a value.
- Reference is often made below to optional text – shown as `[text]`. If you use this, it will be printed in the output file, making it easier to follow.

Sets

Sets provide GAMS with its power. For example, if we have 2 commodities in our model, instead of writing a market clearing equation for each, we can write a general equation indexed on the set of commodities. If we later want to include 100 commodities, all we need to do is to expand the definition of the set, not write 98 new equations.

Each set has a **name,** an optional descriptive label and is defined over the elements comprising it.

The standard format when declaration and definition is made in one statement:

```
SET
name [text]  /element1 [text], element2 [text], …… /
```

Example:

```
C  commodities
   /AGR-C agricultural commodity, NAG-C non-agricultural commodity/
```

In this example, a set consisting of two commodities (AGR-C and NAG-C) is defined.

Instead of using a comma to signal a new element, you may move to a new line:

```
C  commodities
   /AGR-C  agricultural commodity
    NAG-C  non-agricultural commodity/
```

- The names of sets may have up to 10 alphanumeric characters and must start with a letter.
- The names of elements may have up to 10 characters, must start with a letter or a digit, and may include the characters "+" , "-" and "_".

The ALIAS statement is used to give a different name to a previously declared set:

```
ALIAS(C,CP);
```

This is useful when we want to specify an operation that involves interactions between elements of the same set. In the example, "cp" refers to "c prime" or "c' " an alias of "c".

Parameters

The distinction between parameters and variables in GAMS is not quite the same as in economics. Both refer to entities that hold values. The value of a parameter does not change during the solution of the model, while a variable's may. Thus, an exogenous variable in an economic model could be declared as a parameter in GAMS.

Parameters have dimensions – the number of indices over which they are defined.

- Common formats for entering data:
    1. lists (any dimension with declaration and assignment in one statement)
    2. tables (two or higher dimensions with declaration and assignment in one statement)
    3. direct assignments (any dimension, declaration and assignment are separate)
    4. using MS Excel: many modellers use MS Excel to enter data. We will do that once we have the basics done. In large models this method has largely replaced the use of TABLE as a format.
- The same parameter can be assigned a value more than once. (In most experiments, you change the values of one or more parameters.)
- Example of (1) with one dimension:

  ```
  PARAMETER

  name(set) [text]  /element1 value, element2 value, ……/  ;
  ```

  (Set refers to the name of the set in the domain. GAMS checks the domain.)
- Example of (1) with no dimension:

  ```
  SCALAR

  name [text] /value/
  ```

Example:

```
PARAMETER

CONSVAL(C) base-year consumption values

 /AGR-C 125, NAG-C 150/ ;
```

- Example of (2) with two dimensions:

```
TABLE
name(set1,set2) [text]
                        element1_in_set2     …….
element1_in_set1              value
```

Example:

```
TABLE SAM(ac,acp)
          AGR_A     NAGR_A      AGR_C     NAGR_C     LAB     CAP     U_HHD   RHHD   TOTAL
 AGR_A                          125                                              125
 NAGR_A                                    150                                   150
 AGR_C                                                                 50     75   125
 NAGR_C                                                                100    50   150
 LAB         62         55                                                       117
 CAP         63         95                                                       158
 U_HHD                                                  60     90                150
 RHHD                                                   57     68                125
 TOTAL       125        150       125       150       117    158       150    125
;
```

For interpretation:

```
SAM('AGR_C','U_HHD') refers to value 50
SAM('LAB','NAGR_A') refers to value 55
```

- Example of (3) with a previously declared one-dimensional parameter for the case of a direct entry of a value (an algebraic operation is another option):

```
name('element1') = value;
```

Example:

```
P0(C)  = 1;
P(C)   = P0(C);
```

To assign a value to a particular element in the set, replace the set name with the name of the element enclosed in single quotation marks, i.e. 'AGR-C'. Example the first command above can be replaced by:

```
P0('AGR-C')  = 1;
P0('NAG-C') = 1;
```

Variables

Variables are solved endogenously in the model. The variable has to be ***declared*** with a `Variables` statement. Each variable is given a name, a domain if appropriate and text (a description).

- Standard format for a one-dimensional variable:

```
VARIABLE

name(set)  [text]
```

Example:

```
VARIABLE

QH(C)      quantity of household consumption of c ;
```

This statement declares a variable for household consumption over all elements in the set C. In the current model the set includes `AGR-C` and `NAG-C`.

- GAMS has a database for variables that includes lower and upper limits (controlled by the user) and a level (initialized or changed by the user and changed by the solver every time a model that includes the specific variable is solved). By default, they are –INF, +INF and 0, respectively.
- An important difference between variables and parameters is that a parameter has only one "attribute" – its value – whereas a variable has up to seven. The attributes relevant for most modellers are:
    - Its level or current value `.L`
    - Its lower bound `.LO`
    - Its upper bound `.UP`
    - Its fixed value (sets all three above equal) `.FX`
    - Its marginal or dual value `.M`

  When you want to refer to the value of a variable, you have to specify which attribute you refer to. This is indicated by adding a suffix to the variable name.

Example

When displaying a parameter, you can simply give the name:

```
Display X;
```

However, when displaying a variable, you have to specify the attribute

```
Display X.L ;
```

Similarly, when assigning a value to a variable, you have to specify which attribute you are assigning the value to. Thus, if you want to give a value to the level, you have to write

```
X.L(C) = p0(C) * q0(C) ;
```

This applies in many situations. Basically, whenever you want to refer to the specific value of a variable, you have to tell it the specific attribute to which you refer.

- Depending on the problem type, it may or may not be important to assign non-default limits and levels for variables before solving. When solving non-linear simultaneous equation models, it is typically necessary to specify initial variable levels.
- Format for assigning upper and lower limits and levels for a one-dimensional variable:

  ```
  name.LO(set_name) = value;

  name.UP(set_name) = value;

  name.L(set_name)  = value;
  ```

  (Instead of a value, an algebraic operation may be specified.)
- To fix the level of a one-dimensional variable (i.e., to impose identical upper and lower limits):

  ```
  name.FX(set_name) = value;
  ```

Variable Initialisation

GAMS solves models using numerical techniques. Very generally, this involves guessing a solution and then using an algorithm to improve the guess. There are numerous solution algorithms available, but all of them require numbers to work with. If your variables do not have initial values assigned to them, most algorithms will assume default values as a starting point and work from there. It is more efficient if they have initial values that are close to a solution.

Most programs thus have a section that initialises the variables, using the original data. One can do this by simply assigning values to the variables directly. However, this means that these initial (or base) values will be over-written by the new solution values when the model solves. Since the modeller often wants to compare solution values with original values, it is generally better to use what initially appears to be a more cumbersome approach:

- Declare the existence of new parameters, one for each of the variables. An efficient naming convention is simply to add a 0 to the name of each variable;
- Define the values of each parameter equal to the initial variable values
- Assign each of the variables the value of its matching parameter.

Example

If we have a variable for the price of each commodity C, we would have

```
Variables
      …
      P(C) price of commodity C
      …;
Parameters
      …
      P0(C) initial price of commodity C
      …;
…
P0(C) = 1 ;
…
P.L(C) = P0(C) ;
```

In this way we can always refer back to the initial value of the variable.

Equations

- Equations must be ***declared*** and ***defined*** in separate statements.
- "Equations" refers to equations (with equality sign) or to inequalities.
- Standard format for the ***declaration*** of a one-dimensional equation:

```
EQUATION
name[(set)] [text]
```

Example:

```
EQUATION
      HHDEM(C)  consumption demand for commodity c  ;
```

- Standard format for the ***definition*** of a one-dimensional equation (assuming that it already has been declared):

```
name(set)..  left-hand-side relational_operator  right-hand-
             side;
```

Example:

```
HHDEM(C)..    QH(C) =E= beta(C)*yh/p(C);
```

Note that this equation calculates consumption demand for all elements in set C, i.e. AGR-C and NAG-C. There are thus 2 equations specified here.

- The "relational operators" are: =L=, =E=, and =G= (in standard algebra: ≤, =, ≥)
- If an equation is defined over a domain, the equation is defined in the same way irrespective of whether the set in the domain has one element or 10,000 elements.
- The symbol =E= is used in equation definitions; the symbol = is used in assignments.
- In any given equation, variables can appear on the left- or right-hand side, and the same variable can appear more than once.
- An equation definition can appear anywhere (as long as all sets, variables and parameters in the equation, and the equation itself, have been declared). The parameters in the equation can be assigned values later.

GAMS Summation and Product Notation

- The same notation is used in equations and in assignments for variables and parameters.
- Examples of summation over one and two sets:

```
SUM(set,  parameter_name(set) )

SUM((set1, set2),  parameter_name(set1, set2) )
```

Example:

```
INCOME = SUM(C, CONSVAL(C));
```

Note that this is NOT an equation. `INCOME` is a parameter and this command assigns a value to the parameter.

- Products are defined using the same format. For the product over one set:

```
PROD(set,  parameter_name(set) )
```

Example:

The Cobb-Douglas utility function can be represented as

```
Utility = PROD(C,QH(C)**beta(C))
```

The double ** represents 'to the power of'.

Models

- A MODEL is a collection of equations.

- You have to declare a Model, giving it a name and specifying which equations should be included in it.
- The standard formats are:

```
MODEL  name [text]  /equation1, equation2, …../
MODEL  name [text]  /all/
```

("equation1" refers to the name of the equation, i.e., not including its domain.)

- The second format works if you want to include all previously declared equations in the model.

Example:

```
MODEL
 CPE2C Cobb-Douglas consumer model
 /HHDEM/
```

- You can declare many different models in the same program.

Solve Statement

- If the model has been declared and defined (without error messages), and if you have defined the sets, the parameters, and the limits and levels for the variables (if needed), then you are ready to solve the model.
- The format:

```
SOLVE model_name USING solution_procedure
```

- If your solution procedure involves optimisation, then you have to add:

```
MINIMIZING|MAXIMIZING objective_variable_name
```

- Common solution procedures include:

```
LP    (= linear programming)
NLP   (= nonlinear programming)
MCP   (= mixed complementarity problem)
```

Displaying and Transforming Results

Most modellers nowadays dsiplay results in GDX file, which we explain below. This section is therefore largely redundant, but we retain it as it may be useful especially when searching for errors in the program.

You will normally end your program by telling GAMS which results to display and in what format. If there is no Display statement, GAMS will not show you anything.

The simplest display statement is a list of all variables and parameters in the model. For ease of reference in the listing file it is useful to have these in an order that makes sense – alphabetic, or by price, quantity, value and so on. GAMS displays in the order specified in the display statement.

When displaying variables you have to specify the attribute to display. However, you do not need to specify the domain of the variable. Thus

```
Display P.L, Q.L ;
```

Not

```
Display P.L(C) , Q.L(C) ;
```
or
```
Display P, Q ;
```

You may also want to transform some of the results. For example, you may wish to calculate percentage deviations from the base. This can be done using the .L attributes of the variables. You will also have to display the transformed results.

You can also control the format in which results are displayed. The main ways of doing this are:

Local Display Controls

Syntax: `option "identifier" :d:r:c`

This controls the display of the specific entity "identifier", 'd' is the number of decimal places you want (0-8); 'r' is the number of labels in the row and 'c' is the number of labels in the columns (for a multidimensional entity).

Global Display Controls

Syntax: `option decimals = "value"`

This controls the display of all entities displayed after it, unless they are controlled by a local control.

NOTES ON GAMSIDE OUTPUT FILES

When you run your program, the output appears in a listing file. This is shown in a listing window that has two panels, showing the actual listing file and an index to that file. This index panel provides a convenient way of moving around the listing file. Inserting `$STITLE [text]` at various key points in your code makes it easier to move around the listing file.

Contents of the Listing File

Outputs from the run of a program generally contain

- Echo print – a printout of the coding, with each line numbered for ease of reference
- Reference maps – to help you locate entities
- Equation and column listings
- Status reports
- Results

Most of these are useful when developing a program, but can be turned off once it is running properly

Echo Prints

The echo print is a copy of your input file with line numbers inserted in the beginning of each line. (The numbers are inserted for later reference.) You can prevent this by including the statement `$OFFLISTING` near the start of your program.

Reference Maps

Reference Maps are useful for reporting your model. There are two types of maps:

1. An alphabetical list of all "entities" (sets, parameters, variables, equations and models) showing the line numbers and types of appearance;
2. A list of model entities grouped alphabetically under each type with the documentary text.

These can be turned off using `$offsymlist` and `$offuellist` (Userguide p209)

Equation and Column Listings

The **equation listing** shows a selected number of specific equations for each generic equation. For each specific equation, it shows:

- the current values of sets and parameters plugged into the general algebraic form of each equation
- the total value of all constants (after =E=)
- the total value of the left-hand side (using the current values of sets, parameters and variables) and the difference RHS-LHS if it is non-zero (LHS = #, INFES = #)

The **column listing** shows a selected number of specific variables for each generic variable. For each specific variable, it shows:

- its current values (.LO, .L, .UP);
- the equations in which it appears;
- its coefficients in each equation.

Model Statistics and Status Reports

In the MODEL STATISTICS, BLOCK refers to the number of generic equations/ variables while SINGLE refers to the number of specific equations/variables.

If you solve an MCP model, everything has worked as intended if SOLVER STATUS shows "normal completion", and MODEL STATUS is "optimal".

Solution Reports

The model solution includes lists with equations and variables grouped by block and with the post-solution values of their databases.

Result Displays

Provided everything is working, this is the part of the listing file that modellers are most interested in.

Results for Parameters are preceded by a line

```
----  line_number PARAMETER parameter_name
```

Results for variables are preceded by a line

```
---- line_number VARIABLE variable_name
```

You can search through the listing file to find the results, but it is easier to use the index panel in the listing window. Expand the heading Display and you get a list of the entities you asked to be displayed. Double-click on any symbol and it will shift to that part of the listing file.

Remember – only the parameters and variables that have been specified in display statements will be listed.

Error Messages

Error messages appear on the line immediately following the error.

The format is

```
**** $ error_number
```

(The $ appears exactly where the first error appears.)

The interpretation of the error code appears at the bottom of the echo print (the default) or on the next following line (if ERRMSG=1 has been inserted at the end of the file GMSPRM95.TXT).

If your program fails to solve, search for the **** in the .lst file to find where the error is. (Under GAMS IDE, the log file shows the error in red. If you double click on the error, you will jump to the place in the code where it occurred.

Common errors include:

- using reserved words when one was not expected
- omission of semicolon before direct assignments
- misspelling the names of set elements
- using the wrong names for entities (sets, variables, parameters, equations, models)
- over- or under-controlled indices (over-controlled: controlled both in domain and in some other operation, for example a summation; under-controlled: controlled neither in the domain nor in any other operation – not known which value should be used)

A small error may produce a large number of error messages caused by the presence of the first error. **Lesson: fix the first error, ignore the others (unless they obviously reflect another later mistake on your part).**

SOME USEFUL GAMS TOOLS

We cannot go through all the powerful functions and programming tools that GAMS provides: you should learn these as you use GAMS by referring to the user guide and by looking at code written by other, more experienced GAM users. The GAMS website is very useful (http://www.gams.com/). Here we point you in the direction of some of the more useful aspects of GAMS and hint at how they can be used in CGE modelling.

Loop Statements

As in all programming languages, GAMS can perform repetitive operations most efficiently in LOOPs. A loop is simply a portion of code that the program will run again and again for a specified number of times or until a specified condition is met. It then proceeds with the rest of the program.

In GAMS, the syntax of the loop statement is,

```
LOOP  (controlling_domain[$(condition)],
            statement {; statement}
```

```
) ;
```

The controlling domain determines whether the loop will be repeated or not. It will normally refer to a set, so that the loop will be repeated for each element in the set. The $condition is an optional way of restricting the loop to those elements of the set that meet a specified condition. The statements are the code that you wish to have performed repeatedly. Note the final parenthesis that shows where the loop ends.

An obvious use of LOOPs is when you want to run a number of simulations. We could do this by running the program over and over again, making the relevant changes each time. But this involves a lot of repetitive coding and the results will be buried in the listing files. Instead we can

- Create a set consisting of the names of the different simulations;
- Create parameters that hold the values of the variables for each simulation
- Run a loop that assigns the simulation values to the model variables and solves.

Example

Say we have a model with two goods, AGR-C and NAG-C, each with a customs tariff, tm('AGR-C') and tm('NAG-C'). We want to examine the effects of removing the tariff on each good, i.e., set the tariff from its existing rate to zero. We can use a loop, as follows

Name each simulation and create set comprising the simulation names

```
SIM simulation set

      /AGTAR removing the tariff on agric goods

       NAGTAR removing the tariff on NAG goods/ ;
```

Create a new tariff parameter

```
      tmsim(C,SIM)
```

Assign the appropriate values to this:

```
      tmsim(C,SIM) = tm(C);     {This makes sure that the tariffs we do not want to
                                 change are correct}

      tmsim('AGR-C', 'AGTAR')    = 0 ;

      tmsim('NAG-C', 'NAGTAR') = 0 ;
```

Run a loop over the simulations:

```
      LOOP(SIM,

            tm(C) = tmsim(C,SIM) ;

            SOLVE model;

            …save the results for reporting …

           );
```

Display results

Note that we have to save the results in a reporting file, otherwise we will only have the results of the last simulation run. To save them, we normally create new parameters, one for each variable we want to report on but declared over all simulations, and then assign the solution values to these (in the loop). Thus, if one of our variables is QD(C,H), we define a parameter, QDREP(C,H,SIM). In the loop above, we would write

```
      QDREP(C,H,SIM) = QD.L(C,H) ;
```

After we have completed the loop we can display `QDREP`.

Include and Batinclude

Good computer programs have a clear structure. One way of adding structure to GAMS programs is to keep different parts of the program in separate files. For example, we might keep all the data in a data file and the commands for displaying results in a results file.

These can then be 'called' into the main program by using the $INCLUDE or $BATINCLUDE commands.

$INCLUDE

Syntax: `$INCLUDE <file_name>` where the $ is in the first column;

When the program reaches this line, it goes to the include file and does everything commanded in it and then returns to the main program and continues. It is exactly the same as if the code in the include file had been in the main program. The only advantage is for the user—it is often easier to work through and to see the structure of the code in short files.

It is standard to give files to be included the extension `.inc`

$BATINCLUDE

Syntax: `$BATINCLUDE <file_name> argument1 argument2 …` where the $ is in the first column;

This is similar to but more powerful than the `$include` command

It calls the named file and passes the arguments to it as strings to be substituted in the batfile.

Thus, in the main file you can have

```
$BATINCLUDE LOOPREP.INC SIM
```

and in the `LOOPREP`.INC file you could refer to

```
QDREP(C,H,%1)
```

When `LOOPREP.INC` is called, the current value for SIM will be substituted for %1.

The name 'batinclude' is a hangover from DOS, which worked with BATch files.

Save-Restart

Once programs get big, they may take a long time to solve. It is tedious solving the whole program every time, even when you are interested in only one particular simulation.

To avoid this, use the save and restart options.

- With the file whose output you want saved in the GAMS IDE edit window, type `S=<name>` in the command prompt window (top right of GAMS IDE edit screen), where `<name>` is a convenient short name.
- Run

This will save the output of the run in an already compiled form

- Then move to file containing the code you want to use this output
- Type `R=<name>` in the command prompt window

The program will pick up the already compiled output of the first run and continue.

All this is doing is breaking a long program into two, running the first part and then running the second. The advantage is that you do not need to re-run the first part every time you want to run the second.

This is particularly useful when you are performing a number of different simulations based on the same base run. You do not have to re-run the base each time you do a new simulation.

Another useful application of this is to save all your results in this way and then write code that calls up specific results with a restart command. This gets around one of Murphy's laws of modelling – you will have forgotten to print out a result you need in the write-up.

You can save the output of the second run so that it can be restarted by a third program. Simply put `R=<NAME> S=<NAME2>` in the command prompt and then start the third program with the command `R=<NAME2>`.

There many other useful commands you can put into the command prompt window – they are discussed in the Appendix B.3 of Userguide.

GDX: Gams Data eXchange

In recent years there have been significant improvements in the way in which results can be displayed. The GAMS Data Exchange (GDX) is a powerful tool that allows GAMS to interface with other applications, such as MS Excel, R, MS Access, etc. This can be used for inputting data from a spreadsheet or outputting results to one.

There are a large variety of ways in which GDX can be used and it is well-worth investing time in exploring the options.

Displaying results

The easiest way of putting results into a gdx file is to include a command in the command prompt window. Thus, writing "`gdx=results`" will cause GAMS to write the values of all entities to a file called 'results.gdx'. This can be opened in either GAMSIDE itself or GDXVIEWER, a standalone programme.

It is easy to change the format of the data in GDXVIEWER and to export it to other applications. Simply right click on the variable you want to export and follow the instructions. Warning: while you can export all your results to an MS Excel file, it is likely to be large.

You can also use gdx to display charts of your data and results. This is very useful for quick inspection of key results.

A useful tool, especially when debugging code, is Reference Files. The reference window provides a very convenient way of looking at your model and other associated information.

To implement it write '`rf=filename`' in the command prompt field.

When you run your model, you will see a line `----RefFile <filename>` in blue towards the top of the log file. Double click on it. The contents of the reference file are self explanatory. Try double clicking on an entry.

Using GDX to Input Data

Often it is easier to read and maintain data in a spreadsheet format than in a GAMS file. GDX can also be used to input data from MS Excel (or other file formats).

The GAMSIDE help gives a document showing the various ways in which this can be done (gdxutil.pdf).

One way is using the 'gdxxrw' utility. Basically this reads an MS Excel file and creates a gdx file that can then be read at any stage in your programme.

Some notes on specific aspects of GDXXRW

The examples in the help files are very useful, but there are some aspects of gdxxrw that take a while to get to know. This section provides an intuitive account of some of them.

We use gdxxrw to read data in from an MS Excel file. There are different ways of doing this. One is to include at an appropriate point in your code a line like

```
$CALL 'GDXXRW i=datafile.xlsx o=data.gdx index=index!A1'
```

The $Call calls the gdxxrw. i is short for input and tells gdx which xlsx file it is to work with. o is short for 'output', and tells gdxxrw which gdx file to create. We could tell it in the call what it is to read from the xlsx file, and where they are. But it is easier to use the index function. It is also more systematic: you can store all of the details of a particular version of a model in an MS Excel workbook, so that running different versions simply requires calling a different workbook. Hint: it is good practice to develop your own systematic way of setting out the workbook. For example, if the index always starts at cell A6 in the a sheet called index, you will get into the habit of constructing the data workbooks systematically and quickly.

The workbook can contain all of the elements of sets and all numerical data for parameters and variables.

`Index` tells it where to look for each piece of information. The set-up in the "Index" sheet of th Excel Workbook "datafile.xlsx" would look like this:

	A	B	C	D	E	F
1				rdim	cdim	dim
2	set	AC	Sets!A5	1		
3	dset	A	Sets!D5	1		
4	dset	C	Sets!I5	1		
5	par	SAM	SAM!A5	1	1	
6	par	LESELAS2	Helas!F5	2	1	
7	par	scale	SAM!A3			0

GOOD MODELING PRACTICE

There are many different styles of modelling, most of which work. You need to develop the style that best suits you. However there are some general tips that we recommend

Replicating the Base

The GAMS syntax checker only verifies that we have not made any errors in the coding of our program, not that our model makes sense or works properly. We need to check this. Although it is not fool proof, the standard way modellers use to do this is to see whether the program solves for the base. In other words, if we ask it to find a solution based on the initial values of the exogenous variables, does it come up with the initial values of the endogenous variables? If it does not, something is seriously wrong. If it does, then ***perhaps maybe*** all is well.

You should always follow this practice: your first run of the model should be an attempt to replicate the base.

There may be other tests you can use for certain kinds of models. For example, most Walrasian models have the property that they are homogeneous of degree zero in prices – if we change all

prices by a uniform proportion it should not affect real variables. We can check this by shocking the numeraire in such models.

It is good modelling practice to devise such checks: it is too easy to make a silly mistake in the coding that does not prevent it from running as a GAMS program, but is nonsense as an economic model.