GAMS READER

1.1. Introduction

GAMS is a software package to solve systems of equations. GAMS stands for General Algebraic Modelling System and is constructed by the GAMS Development Corporation. GAMS contains different solvers for different purposes.

The homepage of the GAMS corporation (www.gams.com) contains a lot of useful information. From the homepage, a full user guide can be downloaded at www.gams.com/docs/document.htm; the user guide contains the syntax for all GAMS commands and very helpful as a reference when writing GAMS models. Note that the user guide is also available via the Help function in GAMS-IDE.

There is a free version of GAMS available for installation on your own computer. This is a limited version of GAMS, which cannot solve large problems, but it may be handy when you are building proto-type models. It is available for download at http://www.gams.com/download.

1.2. For what type of problems can GAMS be used?

GAMS has its origins in economic modelling, but this does not mean that the models you specify have to be in the field of economics. As you can see the subject index of the GAMS model library - http://www.gams.com/modlib/modlib.htm - GAMS can provide a support optimisation program for the several fields.

Specifying an economic model means that we have to write down one or more equations with some economic relationship. This relationship can be between labour demand and wages, between demand and supply of a good, *et cetera*. The geographical scope of the model can range from writing an economic model for an individual firm to a model that tries to describe the global economy. Many models take the scope of a national economy.

1.3. GAMS Statement Formats

GAMS commands follow a simple syntax:

- Lines with an * (asterisk) in the very first character are comments. For example, an initial comment identifies each of the above example files.
- Lines with a dollar sign in the very first character are GAMS input/output options. For example, `$include` '*filename*' copies in the content of *filename* as if it had been typed at this point. Such $-option lines have no end punctuation.
- All other GAMS statements are coded over one or more lines and end in a semicolon, with component items separated by commas.

- Line breaks, extra spaces and blank lines may be added for readability without effect (i.e. Multiple lines per statement, embedded blank lines, and multiple statements per line are allowed)
- GAMS is not case sensitive. That is, `XX`, `xx` and `Xx` are the same entity.
- Declaration statements such as `free variable(s)` and `equations(s)` may contain a few words of double-quoted "explanatory text" elaborating on the meaning of each defined item immediately after its name is specified. This explanatory text will accompany the item on outputs to make results easier to decipher.
- Names in GAMS consist of up to 9 letters and digits, beginning with a letter. Internal commas, spaces and other special characters should not be used, but underlines (_) are allowed.
- All GAMS command words (e.g. `variable`, `table`, `equation`, `model`) and function names (e.g. `sum`, `log`, `sin`) are reserved, and should not be used for declared names. Naming entities with some standard computer words such as `for`, `if`, `elseif`, `else`, `while`, `file`, and `system` also causes errors.
- Subscripts on variables and constraints are enclosed in parentheses, with explicit (non-varying) subscripts enclosed in quotes.
- Numerical constants in statements may have a decimal point, but they may not contain commas (i.e. use `20000` not `20,000`).

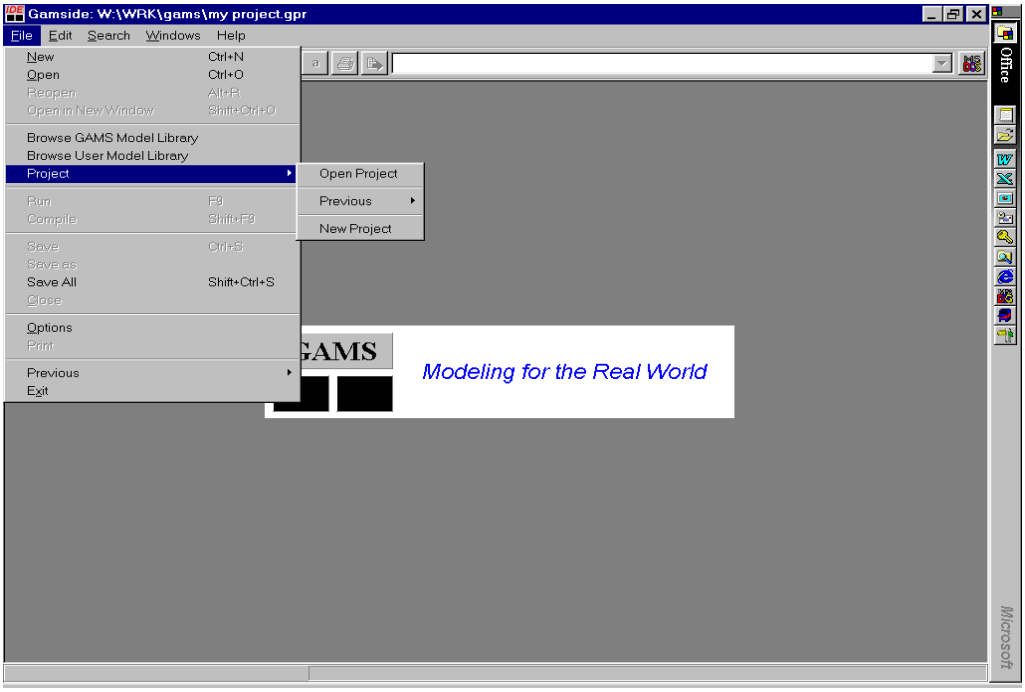
1.4. Working with GAMS-IDE

When GAMS-IDE is started, a window will appear with a menu bar along the top and a main Edit Window for GAMS applications. As with most such systems, input and output operations are controlled by the **File** pull down menu, with other menu items used in edit operations, and in running the GAMS system.

Users should begin each session by selecting a "project". A project is a system file you save but never have to touch. Still, its location is important because the folder (directory) of the current project file is where (`.gms`) input and (`.lst`) output files are saved by default. This allows you to easily keep all the input and output files for any task together in the same directory, and use different directories for different projects. The starting project file (if any) is shown at the top of the main GAMS window. To select another, or create a new one, use the **Project** item on the **File** menu.

The IDE version provides for standard, mouse-driven editing of input files in the main GAMS Edit Window. If the appropriate file is not already displayed, use the **New** or **Open** commands on the **File** menu to activate one. Then create or correct the file with the mouse and tools provided on the **Edit** and **Search** menus. The **Matching Parenthesis** button helps with the many parentheses in GAMS by skipping the cursor to the parenthesis that corresponds to the one it is now positioned in front of. The **Find in file** is also a useful tool, if you work with a complex model.

The GAMS-IDE without any open files looks as follows:



Different parts of GAMS are presented in different colours. For example, all text that is written as a comment appears in grey, keywords are in blue and set elements are in green (at least in the lines where they are defined).

1.5. The general structure of GAMS programs

The first step in modelling in GAMS is to write an input file. Though it is not strictly necessary, normally a GAMS input file has a file-extension `.gms`. You write the input file, run the model in GAMS and look at the output file for the results (the output file has an extension `.lst`).

The general structure of a simple GAMS input file contains the following elements:

PARAMETERS
{gives the data or exogenous constants of the model; these values are fixed}

VARIABLES
{indicates the variables that will be determined (calculated) within the model}

EQUATIONS
{first, the equations have to be declared, then they are defined}

MODEL
{the model is given a name}

SOLVE
{the solution mode is specified, as well as a declaration whether the optimand should be maximised or minimised}

Each of these elements can exist more than once in a single GAMS model.

The restrictions and special meanings of these words are all together called the syntax of a model. GAMS is a computer package and will only understand what you want if you write your model in the correct syntax.

The basic portions of GAMS code are now discussed in more detail.

PARAMETERS

The first step in writing a GAMS model is to provide the constant elements (also called 'exogenous coefficients'). These are data that are not determined within the model, but they have a fixed value that you have to provide.

There are two stages in specifying parameters, first you must declare them by making a list of all parameters and closing the list with a semi-colon, and then you define the values and close each definition with a semicolon. (You can learn more about the use of semicolon at the end of this chapter.)

Suppose you have a parameter `A` and want to give it a value of `3`. Then, the GAMS code is

```
PARAMETERS
      A     Explanatory text on the meaning of A;
      A = 3;
```

The first line is the 'code-word', telling GAMS which part of the model will follow. In this case, the 'code-word' is `PARAMETERS`. First give the name of the parameter, then you may write some explanatory text on the meaning (though this is not necessary). In a new section you define the value of the parameter.

Note that there is an alternative way of assigning values to parameters (and scalars). In the declaration, you can directly specify the value between slashes:

```
PARAMETERS
      A     Explanatory text on the meaning of A      /3/;
```

Constant elements can be specified in several ways: as single parameters called scalars, (if applicable) as vector parameters or as tables. GAMS is indifferent to the way the constant elements are specified, but you'll find that all types have their advantages. The following rules of thumb apply. First, tables are the most compact, so if you can use tables, do so. But

only put data together into one table that have some cohesion with each other. Second, use parameters for constants with two or more dimensions (the dimensions are given by the indices used) and, finally, scalars are used for single values that do not change (the constants in the narrow sense).

Notice that you don't write a semi-colon after each line, but only after the last line of the block with declarations. You can write a block of variables in the same way (see below).

VARIABLES

The variables are what you are really interested in as a modeller. These are the things that are determined endogenously within the model, and the value of which you cannot calculate beforehand. The values of the variables are determined by solving the equations. However, declaration of the variables must precede any use of the variable names later in the file.

In this declaration of the variables, you can also provide an explanatory text to the variable, to help you understand what the meaning of the variable is. The GAMS code is:

```
VARIABLES
      X1    Explanatory text on the meaning of X1
      X2    Explanatory text on the meaning of X2
      Y     Explanatory text on the meaning of Y;
```

So just as with the parameters, you first write the 'code-word', then provide the variables line-by-line. You can provide the explanatory text, but this is not necessary; the value of the variables are calculated by the model therefore you do not have to define value here.

Note that if you want to build more complex models, it becomes important to choose good names for your parameters and variables. Using the words "supply" and "demand" instead of just "s" and "d" helps you in later stages to read the model code. Try to keep the names and structure of the code as logical as possible.

EQUATIONS declaration (defining objectives and constraints)

The objective function and main constraints of GAMS mathematical programs are entered as "equation(s)". The thing to remember is that you have to take two steps:

First, `equation(s)` statements declare names for the equations of the model:

```
equation dem "annual demand";
```

(it declares an equation named `dem` and notes that it corresponds to annual demand)

Second, you write the equation itself in the equation definition section:

*equationname***..**

(and continuing with left-hand side and right-hand side expressions separated.)

The declaration of the equations is straightforward. You can use any name you want to declare the equations. Most people has some standard way of naming the equations. For example, a useful way of naming the equations is to take the variable that is determined by the equation and put a 'Q' in front of the variable name.

```
EQUATIONS
      QX1   Explanatory text on the meaning of the equation for X1
      QX2   Explanatory text on the meaning of the equation for X2
      QY    Explanatory text on the meaning of the equation for Y;
```

You may find it easier to you write down the equations themselves first, and then just above the equations you write the declarations.

One equation always sets the objective function equal to the (`free`) objective value variable.

Any equation can be indexed over one or more subscript sets to define a whole system of similar constraints.

A variety of standard functions may also be included:

Function	Description
`abs()`	absolute value
`arctan()`	arctangent
`ceil()`	integer ceiling
`cos()`	cosine
`exp()`	exponential
`floor()`	integer floor
`log()`	natural logarithm
`log10()`	common logarithm
`max( ,..., )`	max of arg1, arg2, ...

Function	Description
`min( ,..., )`	min of arg1, arg2, ...
`mod( , )`	arg1 modulo arg2
`power( , )`	arg1 to arg2 (integer) power
`round( , )`	round arg1 to arg2 (integer) decimal places
`sign()`	+, 0, - sign
`sin()`	sine
`sqr()`	square
`sqrt()`	square root
`uniform( , )`	random number between arg1 and arg2

Equations definition

The core of any model is given by the equations that have to be solved. In GAMS, you can write the equations fairly straightforward. You write them one by one in the following way:

```
QX1.. X1 =L= A;
QX2.. X2 =E= 5;
QY..  Y =E= X1 + X2;
```

In the example above, the first equation determines the value of `X1`. The equation is named `QX1`, and states that the value of `X1` should be less than `A`. The second equation tells us that `X2` should equal `5`. In the third equation, the value of `Y` is determined as the sum of `X1` and `X2`.

You can use three types of equations:

- the left-hand-side should be less than or equal to (`=L=`),
- greater than or equal to (`=G=`) or
- equal to (`=E=`) the right-hand-side.

This is a very simple model that you could calculate by hand. But the structure of the equations is very general, so writing much more complex model will not lead to much more complex GAMS code. For example, you can do multiplication, raise a variable or scalar to some power, *et cetera*.

The MODEL statement

GAMS can define many models within a single file by collecting different combinations of equations under different names. That is why the user is required to give a name to his/her model even if there is only one.

For simple cases this is accomplished with the statement

```
model modelname /all/;
```

`ALL` refers to that you use all the equations, you can also specify a submodel here by listing all equations you want to include in the model. (Separate the equation names with commas).

In more complicated situations the `all` can be replaced by a list of relevant equation names such as (Separate the equation names with commas).

```
model small /obj,demand/;
model big /obj,demand,detail/;
```

The SOLVE statement

he model name, the solution mode, the optimand and whether to maximise or minimise.

The SOLVE statement is to tell GAMS to solve the model. GAMS does not solve any problems itself. Instead it translates the model into the input required by one of several "solvers" it has available. You provide the ‘code-word’ `SOLVE`, the model name, the solution mode, the optimand (the free variable representing the objective function value) and whether to maximise or minimise:

```
solve modelname using solvertype maximizing objectivevaluevariable;
```

For example, if we want to solve model `TEST` by maximising `Y`, using `DNLP` as the solution mode, we write:

```
SOLVE TEST USING DNLP MAXIMIZING Y;
```

The solution mode tells what type of model you have specified:

GAMS	Description

Type	
CNS	constrained nonlinear system
LP	linear programming
MCP	mixed complementarity problem
MIP	mixed integer linear programming
MINP	mixed integer nonlinear programming
RMIP	solution of the LP relaxation of an integer linear program
MPEC	mathematical program with equilibrium constraints
NLP	local optimization of a nonlinear program over smooth functions
DNLP	local optimization of a nonlinear program with nonsmooth functions
MIDNLP	local optimization of an integer nonlinear program with nonlinearities all in the continuous variables
RMIDNLP	local optimization of the continuous relaxation of an integer nonlinear program with nonlinearities all in the continuous variables

Use of the semi-colon

When you run the input file, GAMS will read the file you wrote line by line. To tell GAMS that the end of a line has arrived, use a semi-colon (";"). The semi-colon is used to tell GAMS the end of a command is reached. In principle, you should end all lines with a semi-colon, except when you declare a list of parameters, variables or equations. Then, this list is regarded as a single block and you should end the block with a semi-colon. For example, if you want to include a second parameter B, you could write:

```
PARAMETERS
A     Explanatory text on the meaning of A;
      A = 3;

PARAMETERS
B     Explanatory text on the meaning of B;
      B = 5;
```

But it is more convenient to write it as a list of scalars and use a semi-colon only at the end of the list:

```
PARAMETERS
A     Explanatory text on the meaning of A
```

```
B     Explanatory text on the meaning of B;
      A = 3;
      B = 5;
```

The definitions of the equations cannot be treated as a block, so you should write a semi-colon after each equation definition.

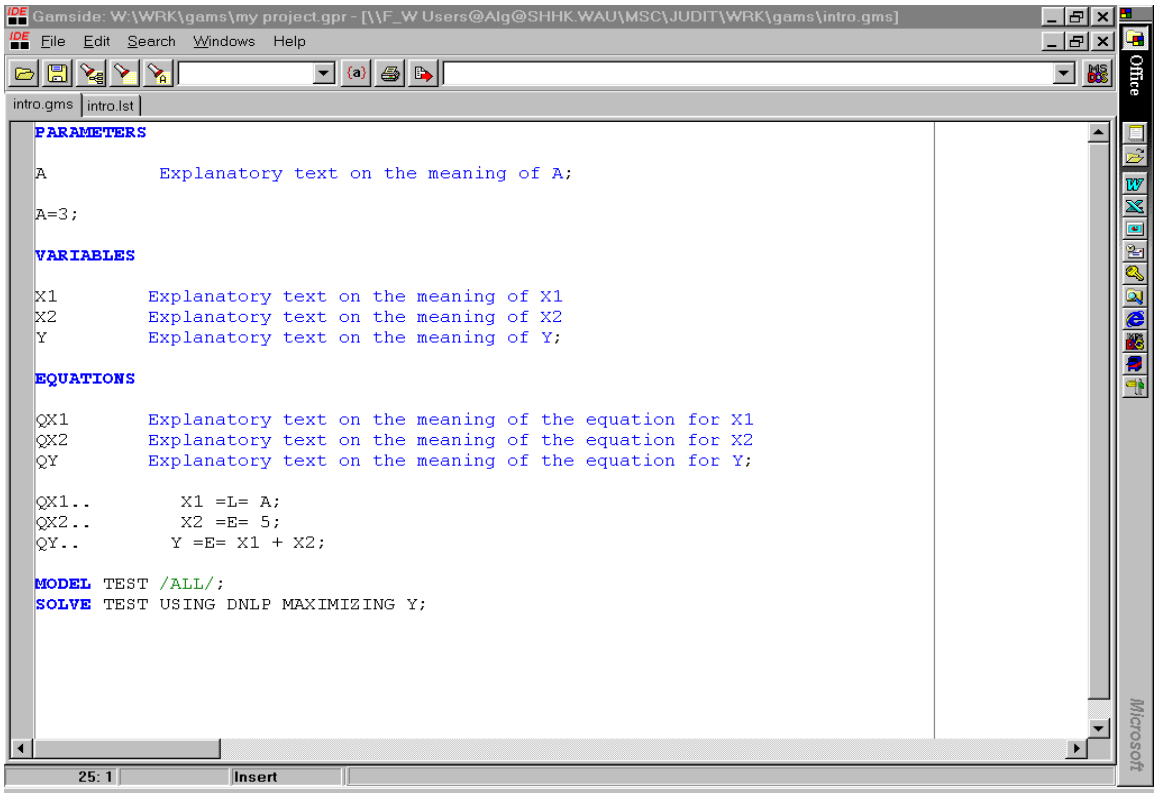
The correct use of the semi-colon will become rapidly clear to you when you start writing your own models, as GAMS will come with an error message if you made a mistake. Still, always be careful in the syntax of your models.

Brief summary:

USE SEMI-COLON	DO NOT USE SEMI-COLON
- after the end of each declaration block (parameters, variables, equations, etc.)	- after each line within a block (for blocks of parameters, variables, etc.)
- after each defined parameter	- after each declared scalar and parameter, only at the end of the block
- after each defined table	- after each declared variable
- after each defined equation	- after each declared equation

The complete code

So, now we have specified a complete model in GAMS code. The full model looks as follows:



Note that the order in which the portions of GAMS code can be specified is quite flexible. The principle that has to be obeyed is that all elements have to be declared before you can use them; so, for instance, an equation declaration must precede the equation specification.

True, this model is not very exiting and you will not be amazed that GAMS can actually compute that the optimal value of Y is 8. But this is just the general structure. Using the same syntax, you can specify much more interesting models and solve difficult systems of equations that you cannot calculate by hand.

1.6. Output

Once a `.gms` file is ready to run, the **Run** item on the main menu bar invokes GAMS. In addition, it automatically causes a `.lst` output to be stored in the current project directory (but not displayed).

Default output begins with an echo of the input like the following (the Process Window). If syntax errors were detected, GAMS includes numbered messages within the echo output and provides a key at the end of the listing.

```
   1  * Echo print example with an error
   2  positive variable x1 "product 1", x2 "product 2";
   3  free variable p "profit";
   4  equations objective, capacity;
   5  objective.. z =e= 10*x1 + 20*x2;
****              $140
   6  capacity.. x1 + x2 =l= 100;
   7  model tiny /all/;
   8  solve tiny using lp maximizing z;
****                                  $257

Error Messages
140  Unknown symbol
257  Solve statement not checked because of previous errors
```

Syntax errors in GAMS input show in red in the Process Window. Clicking on any such red error message brings up the corresponding `.gms` file in the main GAMS window and positions the cursor at the point where the error was detected.

Once all errors are corrected, the `SOLVE SUMMARY` part of the `.lst` file details results of the optimization. The `.lst` output file can be activated using the **Open** command on the **File** menu. However, it is usually easier to first survey an IDE run by examining the separate Process Window, which is automatically displayed. A brief log of the run appears there, and clicking on any of the boldface lines (including run error messages) will activate the entire `.lst` output file and position you on that message. In particular, clicking on **Reading solution for model** will open the `.lst` and position the window at the `SOLVE SUMMARY`.

```
               S O L V E      S U M M A R Y

     MODEL   TINY                OBJECTIVE  Z
     TYPE    LP                  DIRECTION  MAXIMIZE
     SOLVER  CPLEX               FROM LINE  8

**** SOLVER STATUS     1 NORMAL COMPLETION
```

```
**** MODEL STATUS      1 OPTIMAL
**** OBJECTIVE VALUE             2000.0000

                          LOWER     LEVEL     UPPER    MARGINAL

---- EQU OBJECTIVE         .         .         .         1.000
---- EQU CAPACITY        -INF    100.000   100.000    20.000

                          LOWER     LEVEL     UPPER    MARGINAL

---- VAR X1                .         .        +INF     -10.000
---- VAR X2                .       100.000    +INF        .
---- VAR Z               -INF    2000.000     +INF        .

  X1          product 1
  X2          product 2
  Z           profit
```

The first main part of each `SOLVE SUMMARY` reviews results for model equations. Values are given for each objective and constraint, with the `LEVEL` of each constraint providing the amount of the associated resource used in the final solution and `MARGINAL` showing the corresponding dual variable (Lagrange multiplier) value. Any value having only a decimal point is = 0.

The second part of each `SOLVE SUMMARY` details results for all decision variables. These reports show the final `LEVEL` for each variable along with any upper and lower bounds and a `MARGINAL` value corresponding to the variable's reduced cost. Again, values having only a decimal point are = 0.

Errors may also be reported during solving. Such execution errors usually result from an infeasible or unbounded model, program limits being exceeded, or improper computations such as taking the logarithm of a nonpositive number.

It is recommended that each `.gms` file begin with the commands

```
$offsymxref offsymlist
option limrow=0, limcol=0;
```

which turn off all output except an echo of the input file and a `SOLVE SUMMARY` for each `solve` command of the input. These options are the default in the Windows IDE version, but not in UNIX or MDOS.

1.7. Scenarios and sensitivity analysis

Most model simulations do not stop after one solve of the model. Rather, the first solve is used as a reference scenario, that represents the current situation (or, in a dynamic model, the baseline projection represents the most likely development of the variables over time). Then, a so-called counter-factual analysis is done: some parameter values in the model or equation specifications are changed, the changed model is run and the new results are compared to the reference results. This can all be done within one GAMS model file.

The two major types of counter-factual analyses are scenario analysis (sometimes called uncertainty analysis) and sensitivity analysis. The basic difference between these types is that scenario analysis tries to answer questions of the type ‘what happens if one or more elements (or equations) in the model change’, while sensitivity analysis tries to answer ‘what is the consequence of a misspecification of some parameter value’.

In a scenario analysis, several alternative model specifications are compared to each other. These scenarios may differ due to differences in parameter values, but also due to differences in the model equations. In principle, each of the scenarios specified may be equally viable (though they not always are). Often, three or four scenarios are calculated to show the extremes within which the real value will probably lie (*i.e.* the scenarios are used as ‘corners of the playing field’). The scenarios specified may be used to do policy recommendations. For example, if we lower the tax rate on labour with 1%, total employment may go up with x%. Of course, these policy recommendations are only valid within the boundaries and assumptions that are made in the model.

A sensitivity analysis has another purpose: an (individual) parameter value is changed to analyse the effects of the value chosen on the model results. This gives a clue on the robustness of the model with respect to the specification of the model. For example, the emissions of phosphor from agriculture in the Netherlands may be estimated at 0.31 grams per guilder of agricultural production per year, which results in total phosphorous emissions from agriculture of 132 million kilograms per year. To investigate how the total emissions will change if the emissions per guilder of production are 1% higher, one can do a sensitivity analysis. In this example, the relationship is linear and the result straightforward: total emissions will also be 1% higher (133.32 million kilograms). But imagine a more complex model where relationships are not all linear. For example, what is the effect of a slight misspecification of the phosphor content of animal feed on total deposition of phosphor in water? Then, the results of a sensitivity analysis cannot be predicted so easily, and you need to simulate the sensitivity analysis in the GAMS program.