Bruce McCarl's GAMS Newsletter Number 42

March 2018

This Newsletter addresses the following

Contents

1 New commands 1
2 Solvers 3
3 Object Oriented APIs 4
4 Embedded Python Code 4
4.1 Execution Time Example without use of the API 5
4.2 Example at Compile Time Without API 8
4.3 Example With Use of the API to Solve Model instance 10
4.4 Another Example With Use of the API to Solve Model instance 13
5 Editor alternatives IDE, GMS-Manager, Studio and Gtree 13
5.1 Gtree 13
5.2 GMS-Manager 14
5.3 GAMS Studio 14
5.4 Others 14
6 McCarl Guide, Documentation and the Future 15
7 Basic, Advanced and Combined courses offered soon 15
8 Unsubscribe or subscribe to future issues of this newsletter 16

It is been a while since I have issued a newsletter. In the interim, there have been two GAMS releases (24.9 and 25.0). An examination of the associated release notes, plus other material available to me shows broad interest items regarding: a) a few new, but rather specialized, language alterations; b) some solver developments, c) expanded API capabilities, d) means to embed Python code in a GAMS program; and e) alternatives to the IDE all of which I cover herein. I will also make a few comments on documentation and announce the courses I will be giving.

1 New commands

In list form, the developments regarding new commands that can be used are

- The new Embedded Code Facility allows one to include Python code within a GAMS program for use during compile and execution time as discussed below.
- A new command line option (procDirPath) tells GAMS where to create and store the temporary directories (called process directories and is the location where the files 225/a/b/c etc. are placed on a Windows machine). Namely, during a GAMS run, a number of intermediate files are created to pass information back and forth within and between GAMS and the solver. These are stored in the process directory. The location of this directory can be altered via use of the option **procDirPath**. This option is used in the GAMS command line call or in the white command line box in the upper right-hand corner of the IDE. Entries look like –procdirpath=c:\myfiles\ or procdirpath=..\

- Some new compile time constants have been introduced that involve settings for model attributes for solPrint, and solveOpt. Discussion of these and a number of other compile time constants and their use appears here.
- New ways have been introduced to specify the solPrint and sysOut parameters that can be set through the command line, an option statement or a model attribute.
  - solPrint controls the amount of output included in the list file after a solve and can be set to "**off**", "**on**", and "**silent**". In cases numerical values of 0, 1 and 2 also work but GAMS is trying to phase this out.
  - **sysOut** controls whether additional solver generated output (namely the solver status file) is included in the listing file and can be set to **off** or **on**. In cases, numerical values of 0 and 1 also work but GAMS is trying to phase this out.
- Looking at solPrint parameter possibilities led me to experiment with the solPrint silent option which I set through modelname.solprint= %solPrint.silent% which suppresses all solver information in the LST file. I found this convenient and realized that it as useful to suppress solution output when using embedded Python code as in the target price example below. I do note this implementation does not suppress solution information that is of little value in the log file. You can use $offlog to try do reduce that although information still gets through from the solver.
- A revision has been made in the way that the command line parameter **dumpOpt** works. When this option is invoked it causes GAMS to write a file in GMS format that has everything in the model within it (e.g. folding in all the includes). After the revision the file now also contains settings for a number of user specified dollar control options. The resultant file is named with the root name of the file and the extension dmp (tran.dmp for the file tran.gms). I think the main use for it is to create a file you can send to others (like GAMS support) who want to reproduce a run.
- A file was added to the possible library of files that can be included in GAMS files that functionally solves a model instance without regenerating the model through Python. The file is called **pyEmbMI.gms** and is located in the subdirectory of the GAMS system called inclib. It is addressed through the use of a $libinclude statement with arguments that specify a) the name of the model instance that was solved (note this named model instance can have data manipulated in it and be resolved as illustrated in the Python example below); b) the part of the solve statement that identifies the model name, optimization type and variable to optimize (i.e. 'modelname using NLP maximizing z'); c) parameters to include in the GAMS call and d) procedures to use in manipulating parameter and variable attribute data within the GAMS instance. An example where **pyEmbMI.gms** is used appears within two of the embedded code examples below and is discussed there. There is also another example available in the model library files embmiex1, spbender3 and spbender5.
- A new capability has been added to the **Put_Utility** command, namely one can now use the keyword **Save**. Use of **Put_utility save** in a program will cause GAMS to generate a **save file** that encapsulates the current status of the GAMS run as of the time when the put_utility statement is executed. Subsequent programs can be restarted from that file just as is normally done in the GAMS save restart procedure which is extensively discussed in the McCarl Guide.

- The function of the **Put_Utility Statement** has been modified so that the user no longer needs to define a put file name before using it as explained **here** . Rather when put_utility is invoked, a file is automatically defined.
- A new command line option has been introduced that allows one to append a string from an environment variable to the file stem when creating LST, LOG and LXI files. This option is called **fileStemApFromEnv**. The option is apparently useful for users submitting GAMS jobs via mpirun/mpiexec. In that environment multiple, instances of GAMS will be run simultaneously and each instance will involve a run of a file with an identical name wherein only an environment variable called PMI_RANK will be different. That variable contains a number that is unique to that particular model instance (0,1,2,...). This would ordinarily result in overwriting of the LST, LOG and LXI files since GAMS will normally write to modelname.log, lxi and lst. Use of **fileStemApFromEnv** cause the file stem of the LST and other files to be augmented with the contents of the environment variable and will thus allow differentiation among the solver output LST files. This involves specifying **fileStemApFromEnv**= PMI_RANK in the command line and results files modelname1.lst, modelname2.lst etc. Other environment variables could also be used.
- A number of functions within GAMS have been rewritten to improve their reliability, performance and/or precision. These include the **loggamma**, **gamma**, **logbeta**, **beta**, and **binomial** functions. In addition, the gamma and beta functions are now classified as smooth and are allowed for use within NLP models instead of being restricted to DNLP models. Finally, the domains of the beta and binomial functions have been changed.

2 Solvers

A number of changes have been made in terms of the solvers with many, perhaps almost all having updated versions. The ones I perceive as receiving the largest changes and the basic nature of those changes are listed below.

- CPLEX and the associated GAMS in core linkage has been enhanced with respect to its MIP solution capability and in the Benders Decomposition implementation.
- LocalSolver has been enhanced with its preprocessing features entirely rewritten.
- GAMSCHK - after years of languishing unchanged a few changes were made. Steve Dirkse at GAMS with a little bit of help from me fixed a few bugs in the Fortran code. For example, we improved the way the program limited the number of items portrayed when analysis finds many cases of an error within a named variable or equation. We also corrected the way the program dealt with scaled models within the displaycr and postopt routines. Also, the process window and LXI navigation window now contains clickable links to the output from the main GAMSCHK procedures employed in a run. Additionally, a change was made allowing smaller lower bounds (now 10**(-5)) for **LevelFilt** and **MargFilt**. These are used to identify large shadow prices and solution values in the NONOPT procedure when seeking causes of unbounded and infeasible solutions.
- GUROBI was enhanced in its capabilities to solve mixed integer quadratic programming models (MIQPs).
- Lindo/LindoGlobal received enhancements in its solvers for LP, MIP, Global optimization, and NLP.

- XPRESS was enhanced relative to MIP solutions in its use of parallel processors, memory, heuristics, and cutting plane formation. Additionally the presolve was rewritten, and changes were made to its crossover tools.

One other point relative to solvers is that support for 32-bit machines has been eliminated for a number of the major solvers. This is true generally across much of the software industry and in a comment the GAMS staff says "**the future is 64bit and the sooner the users eliminate their dependency on 32bit the better."**

3 Object Oriented APIs

GAMS contains a number of application program interfaces (APIs) that allow developers to more smoothly move data in and out of GAMS when programming in selected environments. APIs are available with some differences by platform (see comments here) for

- .NET which works with the .NET framework version 4
- **C++** which works with C++ versions C++11 or later,
- **Java** which works with Java versions Java SE 7 or later, and
- **Python** which works with Python version 2.7 (and also with versions 3.4 and 3.6 in differing ways on Windows, Mac, and Linux platforms).

In addition there are **expert-level (or low-level) GAMS APIs** which can be used but users should note that their use requires advanced knowledge of GAMS component libraries. These are discussed in the document **Executing GAMS from other Environments**.

My brief examination in this arena leads me the conclusion that as a whole the inherent procedures and complexity of the documentation makes for a steep learning curve for people who don't live and breathe threads and instances and namespaces and classes.

4 Embedded Python Code

GAMS releases 24.9 and thereafter contain the ability to embed Python code within GAMS code at compilation and execution time although the feature is still in Beta (The beta status meaning that until a full release occurs some future changes may be made that could compromise the operability of code reliant on current features).

When I look into the embedded code capabilities, I see two main levels of possible implementation. One involves directly using the embedded code and nothing else. The other involves using the embedded code in conjunction with the **Object-oriented GAMS Python API** (Application Programming Interface).

In particular, across these two classes of use

- Going ahead *without the API* allows one to: a) use Python code to alter data within GAMS at either compilation or execution time; and b) access GAMS resident parameter/set/variable/equation items including solution information at execution time. In modifying data, one can change data within already defined items such as parameters, bounds on variables, right hand sides, or scaling factors. However, it is important to note that at execution time one cannot expand domains of sets, define new tables, or define new named variables/equations. But, domains can be expanded at compile time as the Excel related example below shows.

- Going ahead in interaction *with the API* allows one to modifying data then solve a model and parse through a generated model recovering structural elements. Again, this does not allow one to add new variables or equations. However, the contents of pre-existing variables and equations can be altered. This also permits strategies where variables or equations are predefined with dummy content that subsequently can be changed.

The code can be embedded at either compile time of execution time. Compile time embedding is done using **$onEmbeddedCode Python:** followed by **$offEmbeddedCode [arguments]**. Execution time embedding is done using at first **embeddedCode Python:** then later one uses either **pauseembeddedcode [arguments]** or **endembeddedcode [arguments]** to return back to GAMS. If the execution is paused then later Python code can be included and Python execution restarted with **continueembeddedcode:** which starts the execution up again retaining the results from the earlier Python execution. In doing this, I draw from several documents.

- Technical GAMS aspects of embedding Python code appear in the document https://www.GAMS.com/latest/docs/UG_EmbeddedCode.html
- The API is discussed at https://www.GAMS.com/latest/docs/API_PY_OVERVIEW.html.
- There is some information on types application in the presentation within the file https://www.GAMS.com/fileadmin/resources/presentations/informs2018_EmbeddedCode.pdf.
- There is a family of example applications that can be accessed through use of the library manager within the IDE. These include
    - Alternative implementations of stochastic Benders decomposition in the model library,
    - Codes which test the get and set features in the test library
    - Code for sorting and decomposing set element names in the data utility library.

Here I present four examples that illustrate basic functionality.

4.1 Execution Time Example without use of the API

A simple code that illustrates the very basics of execution functionality appears below and is built upon the model library code transport is available for download as a link with this newsletter and is called Python_embed_trnsport_without_API.gms. The key Python related parts are color coded below. Also just after the code is presented I enter some notes on the basic function of the code segments.

```
embeddedCode Python:
storeallsupplymarg      = []
storealldemandmarg      = []
storeallxval            = []
storeall_solution_stuff = []
savea        = list(gams.get('saved_a'))
saveb        = list(gams.get('saved_b'))
changetomake = dict(gams.get('changetomake'))
pauseembeddedcode

singleton set case(cases) /base/;
parameter solution_stuff(*);
loop(cases,
    case(cases)=yes;
*    now change the data using python code
    continueembeddedCode:
    # get the data from GAMS
    case         = list(gams.get('case'))[0]
    supplychange = changetomake.get((case,'supply'),1)
    demandchange = changetomake.get((case,'demand'),1)
    # multiply the rhs data by the constant
    a = [(i[0],supplychange*i[1]) for i in savea]
    b = [(i[0],demandchange*i[1]) for i in saveb]
    gams.set("a",a)
    gams.set("b",b)
    pauseEmbeddedCode a b
    display a,b;
    option solvelink=2;
    solve transport using lp minimizing z;
    solution_stuff("Obj Value")=z.l;
    solution_stuff("Solvestat")=transport.solvestat;
    solution_stuff("Modelstat")=transport.modelstat;
    solution_stuff("Solve_time")=transport.etsolver;
    solution_stuff("Number_infeasibilities")=transport.numInfes;
*    now back to python to build saved data arrays
    continueEmbeddedCode:
    # get solution data from GAMS
    for irec in gams.get('supply'):
      storeallsupplymarg.append((irec[0],case,irec[1][GMS_VAL_MARGINAL]))
    for jrec in gams.get('demand'):
      storealldemandmarg.append((jrec[0],case,jrec[1][GMS_VAL_MARGINAL]))
    for ijrec in gams.get('x'):
      storeallxval.append((*ijrec[0],case,ijrec[1][GMS_VAL_LEVEL]))
    for mrec in gams.get('solution_stuff'):
      storeall_solution_stuff.append((mrec[0],case,mrec[1]))
    pauseEmbeddedCode
);
* Now bring the reports back to GAMS
continueEmbeddedCode:
gams.set("sv_x_sol ",storeallxval)
gams.set("sv_supply_shadprice ",storeallsupplymarg)
gams.set("sv_demand_shadprice ",storealldemandmarg)
gams.set("sv_solution_status ",storeall_solution_stuff)
endEmbeddedCode sv_x_sol sv_supply_shadprice sv_demand_shadprice sv_solution_status
```

Now some notes on this

1. We need one instance of embeddedCode Python: to start up the Python implementation. In this case since the Python code will be started and stopped multiple times with saving state we cannot have the embeddedCode Python: statement inside the loop. We also cannot end the code before the loop as we want to pick up from within the loop so we use a pause command pauseembeddedCode here which will later allow Python to pick up from the prior execution and continue. This code segment does some initialization and draws some data out of GAMS using the get command as will be explained next.

2. Data within the GAMS program are accessed using a gams.get command of the form

   savea =list(gams.get('saved_a')) where the symbol on the left (savea) is the name of the data item within Python and the symbol in quotes (saved_a) is the name of the item within GAMS.

3. Inside the loop when the continue statement is executed this causes the execution to continue from the prior Python code segment retaining everything generated from all

previous paused segments. This is done using continueembeddedCode: followed by some Python code which is again paused with pauseembeddedCode.

4. Data are placed back into the GAMS program using: 1) a .set command; and 2) a set of arguments on the statement ending or pausing this Python code segment; those arguments tell GAMS what items to change in its database. The set command appears in a form like gams.set("a",a) which identifies the name of the GAMS symbol that will receive the data in quotes ("a" in this case) and specifies the name of the Python item to be transferred to GAMS after the comma (a in this case). The second needed entry is the pause or end (pauseEmbeddedCode or endEmbeddedCode) statement which needs to have a list following it of the symbol names within GAMS to be changed. In this example the list a,b tells GAMS that Python is sending new data for a and b.

5. When using paused and continued Python code in conjunction with a solve statement it **will only work if you also use a solvelink** command with a setting that makes GAMS remain in memory (option solvelink=2; or option solvelink=5). If this is not done then during execution GAMS will eliminate the saved status of the Python execution. This will cause generation of an error message like *** Error at line 148: Error executing "continueEmbeddedCode" section: No embedded library initialized.

6. When one is starting and stopping Python execution as is done in the example above then one must use the **pauseembeddedcode** and **continueembeddedcode** commands.

7. When one wishes to bring information from a GAMS model solution into Python one again uses the **get** command. This is done in the example by the code gams.get('supply').. When this is done with a command like xvalue = list(gams.get('x')) where the x is the name of a variable or equation then the entire solve related vector is brought in plus the set element names that define the specific x case. This means there are multiple elements for each variable and equation. By index position [0][0] would be the first set element for the variable or equation and [0][1] would be the second so the first row has the set information. Then there is a second row with 5 entries where the solution level is in [1][0], the marginal in [1][1], the lower bound in [1][2], the upper bound in [1][3], and the scaling factor in [1][4]. Thus, one will have to go within these elements to copy out the specific values wanted. These can also be safely addressed using Python constants that are automatically defined when Python is run within GAMS that are named GMS_VAL_MARGINAL and GMS_VAL_LEVEL. An illustration of this usage appears within the commands

```
for irec in gams.get('supply'):
   storeallsupplymarg.append((irec[0],case,irec[1][GMS_VAL_MARGINAL]))
```

This command retrieves the solution information for the GAMS equation named supply and merges the shadow price (marginal) information from the position in the first row [1] where the [GMS_VAL_MARGINAL] is located and in this case places it into the Python object storeallsupplymarg.

8. The example illustrates that we have to explicitly incorporate statements to get information on the model solution status as the current GAMS Python interface does not facilitate direct access to the post solution model attributes. Thus, we use GAMS commands to form a parameter containing the information we want

```
solution_stuff("Obj Value")=z.l;
solution_stuff("Solvestat")=transport.solvestat;
solution_stuff("Modelstat")=transport.modelstat;
solution_stuff("Solve_time")=transport.etsolver;
solution_stuff("Number_infeasibilities")=transport.numInfes;
```

and then later use the get command to move the information into Python

```
for mrec in gams.set('solution_stuff'):
   storeall_solution_stuff.append((mrec[0],case,mrec[1]))
```

9. The main point of this example is to show the functionality of getting data back and forth between Python and GAMS. In practice, the exact context of this example is **not something that one would never want to do using Python**. In particular, in this example: a) data are changed within a Python using a simple formula, then are placed back in GAMS and b) reports are formed in Python and placed back in GAMS. Generally, the data manipulation and report formation are much more conveniently done in GAMS without need to code this in Python.

   Nevertheless, there are things that are not convenient to do in GAMS. For example,

   - Data may need to be sorted as is done with Python in the GAMS data utility models library example called embeddedSort.
   - One may wish to form quantiles over GAMS results as done within an example on Erwin Kalvelagen's web page at http://yetanothermathprogrammingconsultant.blogspot.de/2018/09/quantiles-with-pandas.html.
   - Complex numerical formulas involving address calculation and access may be involved that are inconvenient to program in GAMS.

4.2 Example at Compile Time Without API

Another example that operates at compile time including data from Excel is given in the file python_embed_excel_compile_time.gms. Key parts are identified below but for brevity I will not explain the concepts behind statements already covered above

```
$onEmbeddedCode Python:
  alldemand=[]
  demand=[]
  demandq=[]
  supplyq=[]
  allsupply=[]
  supply=[]
  cost=[]
  from openpyxl import load_workbook
  wb = load_workbook(filename = 'myspread.xlsx')
  problem = wb['input']
  for n in range(2, problem.max_column):
      demandelement = problem.cell(row=1,column=n).value
      demandqelement = problem.cell(row=problem.max_row,column=n).value
      demand.append(demandelement)
      demandq.append((demandelement,demandqelement))
  for n in range(1, problem.max_column+1):
      demandelement = problem.cell(row=1,column=n).value
      alldemand.append(demandelement)
  for n in range(2, problem.max_row):
      supplyelement = problem.cell(row=n,column=1).value
      supplyqelement = problem.cell(row=n,column=problem.max_column).value
      supply.append(supplyelement)
      supplyq.append((supplyelement,supplyqelement))
  for n in range(2, problem.max_row+1):
      supplyelement = problem.cell(row=n,column=1).value
      allsupply.append(supplyelement)
  for n in range(2,problem.max_column):
      for r in range(2, problem.max_row):
          cost.append( (problem.cell(row=r,column=1).value,
                        problem.cell(row=1,column=n).value,
                        problem.cell(row=r,column=n).value) )
  print (cost)
  for supply, demand in [('supplyplace',supply), ('demandplace',demand),
                         ('supplymax', supplyq), ('demandmin', demandq),
                         ('trancost',cost)]:
       gams.set(supply, demand)
  wb.close()
$offEmbeddedCode supplyplace demandplace supplymax demandmin trancost

embeddedCode Python:
  from openpyxl import load_workbook
  wb = load_workbook(filename = ' myspread.xlsx')
  if 'output' in wb.get_sheet_names():
    output=wb.get_sheet_by_name('output')
    wb.remove_sheet(output)
  output = wb.create_sheet('output')
  for rec in gams.get('x', keyFormat=KeyFormat.FLAT, valueFormat=ValueFormat.FLAT):
    output.append(rec[0:3])
  wb.save('myspread.xlsx')
endEmbeddedCode
```

Notes

1. The first part starts up Python `$onEmbeddedCode Python:` Here by using the $ form of the command causes this to be done at compile time.

2. The next part is the command `from openpyxl import load_workbook`. This cause Python to access the bring in the package `openpyxl` to open the workbook. The `openpyxl` package is not on your machine in a normal install. Thus before using `openpyxl` you must install it. To do this

   - Get into CMD mode (this can be done by opening the IDE and punching the MS – DOS in upper right) and go to the GMSPython subdirectory (on my machine this is done by typing **cd C:\GAMS\win64\25.0\GMSPython**)
   - Enter the command "**pip install openpyxl**". *Execution of this command will find the package from the internet and install it. You do not have to download anything manually.*
   - Documentation and details on the package openpyxl can be found using an internet search for **openpyxl**.

3. Subsequently the code uses openpyxl procedures and Python commands to get the data from the worksheet. This is the group of statements starting with and shaded as `problem = wb['input']`

4. Finally we get to some GAMS interaction using `gams.set(supply, demand)` to put data into GAMS and the command **`$offEmbeddedCode supplyplace demandplace supplymax demandmin trancost`** to tell GAMS what elements to update and to suspend Python execution and return to GAMS.

5. After the Solve we add some execution time code (`embeddedCode Python)` to put data back into Excel using an append (**`output.append(rec[0:3])`**)to place the solution back in the spreadsheet.

4.3 Example With Use of the API to Solve Model instance

There is also a major solution related advantage one can gain by using Python and it's API in conjunction with GAMS. The advantage involves use of Python code to repeatedly change data in a generated GAMS model without having to regenerate the model from scratch. This could be quite beneficial in circumstances where a model that took substantial time to generate had to be solved numerous times with modest revisions in the contained data. There is also the possibility of using this across machines as implemented in spbenders5 in the model library.

Before beginning discussion of this example let me briefly, define a term that will be used - **model instance**. When GAMS executes a solve statement this causes the GAMS software to create a version of the model and the data contained for passage to the solver. This is called a **model instance**. Inside GAMS, the **model instance** contains current values for all parameters used in the formulation and a mapping of how the GAMS parameters are factored into the model. This permits GAMS parameters that are used in the numerical realization of a model to be changed then factored into the numerical realization without completely regenerating the model **(note the qualification in the next paragraph)**. In turn, then through the procedures explained below one can interact with the **model instance** to change the data within GAMS items that are used in a model and in turn have the numerical aspects of the model formulation adjusted accordingly. (The GUSS solver also uses the model instance).

I also just found out about a major **limitation related to a model instance** that affects embedded code possibilities and GUSS. ***One cannot use this procedure to modify active set elements or modify parameters that are used in $ commands within the model .. equations.*** This is the subject of a technical explanation at https://www.gams.com/latest/docs/S_GUSS.html#GUSS_IMPLEMENTATION_DETAILS. More simply put this is true because of the way this feature is implemented (the parameters to be changed are actually treated as variables in the model instance) and from what I gather is not going to change anytime soon. In fact, trying to do this will earn you error messages like

Exception from Python: GAMS return code not 0 (2), set the debug flag of the GamsWorkspace constructor to DebugLevel.KeepFiles or higher or define a working_directory to receive a listing file with more details.

*** Error at line 210: Error executing "continueEmbeddedCode" section: Check log above

One could devise a way to work around this by manipulating variable bounds (holding variables you don’t want to zero) or constraint right hand sides (rendering constraints non binding) but I have not tried this as of yet.

This makes solution of a sequence of models substantially faster particularly when the generation, solver close down, GAMS start up and solver start up time are a large component of the elapsed total job time. On larger models, this may save substantial time depending on the relative amount of time used in model generation and the overhead of GAMS and solver close down and start up and on the success of specifying the advanced basis.

An example of this again done in the context of the transport library model is:

```
embeddedCode Python:
from gmomcc import *
$set useSolverLog 1
$set solverlog
$if set useSolverLog $set solverlog output=sys.stdout
```

```
def solveMI(mi, mInfo, symIn=[], symOut=[]):
  for sym in symIn:
    gams.db[sym].copy_symbol(mi.sync_db[sym])
  mi.solve(%solverlog%)
  for sym in symOut:
    mi.sync_db[sym].copy_symbol(gams.db[sym])
  gams.set(mInfo,[ ('solveStat',mi.solver_status), ('resUsd',
gmoGetHeadnTail(mi._gmo,gmoHresused))])
pauseEmbeddedCode
Option solvelink=2;
$libinclude pyEmbMI miTran 'mod using lp min z' -all_model_types=cplexd a.Zero b.Zero
loop(cases,
    supplychange=1;
    supplychange $changetomake(cases,"supply")= changetomake(cases,"supply");
    demandchange=1;
    demandchange $changetomake(cases,"demand")= changetomake(cases,"demand");
    a(i)=saved_a(i)*supplychange;
    b(j)=saved_b(j)*demandchange;
*   solve transport using lp minimizing z;
    continueEmbeddedCode:
    solveMI(miTran,'mInfo',['a','b'],['z','x','supply','demand'])
    pauseEmbeddedCode z x supply demand mInfo
    saved_x_solution(i,j,cases)=x.l(i,j);
    saved_supply_shadow_price(i,cases)=supply.m(i);
    saved_demand_shadow_price(j,cases)=demand.m(j);
    saved_solution_stuff("Obj Value",cases)=z.l;
    saved_solution_stuff(ht,cases)=mInfo(ht);
    saved_solution_stuff("Number_infeasibilities",cases)=sum(i$supply.infeas(i),1)
                                                         +sum(j$demand.infeas(j),1)
                                                       +sum((i,j)$x.infeas(i,j),1);
```

Now for some notes on the key parts of this code

1. We first began with an embedded code section that begins the Python - embeddedCode Python:. This code segment contains two main components. First, we use from gmomcc import * which informs the API that we will use the function gmoGetHeadnTail to access some of the model solution result attributes. Second, the code contains definition of a function **def solveMI(mi, mInfo, symIn=[], symOut=[]):** that will cause the model to be solved changing the input data listed between the brackets in **symIn=[ ]** and returning the output listed between the brackets in **symOut=[ ]**). In the next note below I cover the main elements of this function.

2. The statement **def solveMI(mi, mInfo, symIn=[], symOut=[]):** defines: a) the name of the function **(solveMI) that will be called elsewhere within Python**; b) the name of the model instance to be solved (**mi)** by this statement; c) the name of a Python parameter that will be built within the function for passage back to GAMS containing solution status information (**mInfo);** d) the names of the data items encased in brackets that will be sent from GAMS to the function that will alter the numerical realization of the model instance (in **symIn=[]**), and e) the names of the solution items encased in brackets that will be passed back to GAMS (in **symOut=[]**) after the solution is complete**.**

3. Now, to improve clarity, let us look at a call of this function. Within the GAMS loop there is a continue through pause Python code segment that contains the line **solveMI(miTran,'mInfo',['a','b'],['z','x','supply','demand'])**. This indicates a) the model instance called **miTran** is to be solved; b) that the function will return solution status information to Python in the data item named **'mInfo'; c)** when the model instance is set up for a solution that alternative data will be passed into the function for the

parameters **['a','b']**; and d) after the model is solved the function will return the solution information for the variables and equations named **['z','x','supply','demand']**.

4. Now let us look at the contents that appear within the function. First, the lines

```
for sym in symIn:
    gams.db[sym].copy_symbol(mi.sync_db[sym])
```

transfer data from the GAMS database to the model instance for the parameters named within the first set of brackets in the call namely in this case a and b **['a','b']**.

The next line

```
mi.solve(%solverlog%)
```

causes the model instance to be solved.

After that we retrieve the information from the model instance solution and place it back into Python using two command sequences. The first of which is

```
  for sym in symOut:
        mi.sync_db[sym].copy_symbol(gams.db[sym])
```

and retrieves the information requested by the function call for the variables and equations **['z','x','supply','demand']** and places those solution results into Python items with the same names.

The second sequence

```
gams.set(mInfo,[ ('solveStat',mi.solver_status), ('resUsd',
gmoGetHeadnTail(mi._gmo,gmoHresused))])
```

retrieves the solver status information using the capabilities of the API (in the full example the statement also recovers modelstat but this not shown here to make things fit). The resultant information is transferred into a GAMS parameter identified by the string mInfo.

5. Later in the code we use a statement to cause the initial model instance to be set up and solved plus GAMS to remain in memory which is

```
Option Solvelink=2;
$libinclude pyEmbMI miTran 'mod using lp min z' -all_model_types=cplexd a.Zero b.Zero
```

In this case we are using the predefined gms file pyEmbMI that is contained in the INCLIB subdirectory of the GAMS system. **We must have the option solvelink, $set and $if statements before this statement for the code to work.** The libinclude call contains: a) the name to give the model instance miTran; b) the text to use in the solve statement giving model name and particulars about optimization 'mod using lp min z'; c) some command line parameters to use when GAMS is invoked -all_model_types=cplexd; and d) what to do if the input data do not exist for a case (a.Zero b.Zero where in this case **a.zero** says treat elements not in the revised parameter **a** as zeros). You can also tell how to handle bounds. For details see the comments in the code in inclib.

6. Finally in the loop we see

```
continueEmbeddedCode:
     solveMI(miTran,'mInfo',['a','b'],['z','x','supply','demand'])
pauseEmbeddedCode z x supply demand mInfo
```

which continues the execution of the Python code. The code calls the **solveMI** function to execute the solve over the model instance named **miTran** the remaining parameters of which are described in points two and three above. After this, the Python code is paused and on the back end of the pause statement the user tells GAMS the names of the parameters that need to be updated in the GAMS database. In this case these are the listed objects appearing after the command **pauseEmbeddedCode** and are **z x supply demand mInfo**.

4.4 Another Example With Use of the API to Solve Model instance

To partially show the solution speed advantage a deliberately slow converging example setting up a subsidy to achieve a price floor was developed (for details on the concept see paper in AJAE by Chang, McCarl, Mjelde and Richardson). The resultant key Python related parts of the code (python_embed_target_price_with_api.gms) are

```
$set useSolverLog 1
$set solverlog
$if set useSolverLog $set solverlog output=sys.stdout
embeddedCode Python:
from gmomcc import *
def solveMI(mi, mInfo, symIn=[], symOut=[]):
  for sym in symIn:
    gams.db[sym].copy_symbol(mi.sync_db[sym])
  mi.solve(%solverlog%)
  for sym in symOut:
    mi.sync_db[sym].copy_symbol(gams.db[sym])
  gams.set(mInfo,[('modelStat',mi.model_status), ('solveStat',mi.solver_status), ('resUsd',
                    gmoGetHeadnTail(mi._gmo,gmoHresused))])
pauseEmbeddedCode
$libinclude pyEmbMI modinstance 'mod using nlp max z' -all_model_types=conoptD initialguess.Zero
```

Which sets up the solve function (solveMI) and solves the first model instance (modinstance)using cosde in the inclib directory of GAMS via the Libinclude of pyEmbMI. Then later in the loop we have

```
continueEmbeddedCode:
       solveMI(modinstance,'mInfo',['initialguess'],['balance'])
pauseEmbeddedCode balance mInfo
```

which continues the Python calling the function to carry out solution of the model instance (modinstance) using the function (solveMI) as in the prior example. Here we are changing the data in initialguess and recovering the solution information for balance plus the solution status in Minfo.

5 Editor alternatives IDE, GMS-Manager, Studio and Gtree

There has been recent dialogue on the GAMSlist regarding the IDE and "modern" editor alternatives to it. Recently announcements were made by two parties regarding alternative GAMS customized editors and there is one other I am aware of that has existed for a number of years. I will briefly cover these below although I must admit that I have not used any of them so the content below is all derived from the information available to me on the web or in emails.

In particular, the following are or are soon to be available.

5.1 Gtree

Gtree provides an editing platform that has been customized for GAMS and has been in existence for at least 10 years. It was developed by Wietse Dol. From what I can tell the latest version was released somewhere shortly after May 2015. According to its documentation Gtree will show the structure of your GAMS code in tree form (integrating the base files and all relevant ones referenced in **$include, $batinclude, $libinclude, $sysinclude** statements). Then by clicking on the tree you can jump into and out of files as well as edit files and run jobs. The documentation indicates Gtree can among other things: a) find and replace text anywhere in the associated tree of files making up a model; b) maintain a to do list; c) run the model with GAMS; d) match beginning and ending characters of the form (), [], {}, " "", //, and \\; e) position the cursor at the point of the first compilation error when one occurs; f) look up the GAMS text for compilation error messages; g) edit GDX files; h) merge and difference GDX files; i) transfer cursor focus to locations with a particular model file family where a symbol is declared, defined, assigned, used as an index or referenced as in the GAMS refreader; j) add bookmarks and jump to them; and k) syntax color not only keywords but also named sets, parameters, variables, equations and models. Gtree is also compatible with Gempack and interfaced with R for statistics and graphics.

Gtree is released through the web page http://www3.lei.wur.nl/GAMStools/index.htm and is free. This is not a product developed by GAMS Corporation.

5.2 GMS-Manager

An alternative, customized editor called GMS-Manager was recently released. It was developed by Ingo Huck and is sold through https://www.gms-manager.com/. According to the documentation GMS-Manager is a "modern GAMS-IDE" that allows one to edit gdx-files and increase GAMS programming efficiency. Editor capabilities mentioned or implied include an auto complete function, a symbol search, automatic error underlining, a real-time symbol table, error detection as code is built, syntax coloring, and tree structure diagrams of include files. For working with GDX files, there are mentions of procedures for adding symbols, adding data items, merging files, altering display formatting when examining contents, provision of specialized copy paste procedures for moving data from Excel files, and construction of data based charts. This again is not a product developed by GAMS Corporation.

5.3 GAMS Studio

GAMS Development has indicated that they are working on an alternative to the IDE that they call the GAMS Studio. In an email they state "GAMS Studio is based on C++ and Qt which makes it fast, reliable, and platform independent ([thus usable on] Windows, Mac and Linux)." They also indicate that an early stage GAMS Studio will be released sometime soon, namely during the spring of 2018 within the GAMS 25.1 release. They also indicate that GAMS Studio will be open source with the first version including basic functionality to edit and run GAMS models and view results –" but nothing very advanced". Also that "many new features - interactive debugger, model instance explorer, etc. - are on the development horizon, subject to prioritization and user feedback."

5.4 Others

- There are some other packages that are listed on the GAMS contributed software page https://www.GAMS.com/community/contributed-software/ including a MAC based IDE (https://dridium.wixsite.com/amalGAMS), a package called Veda-BE (see http://support.kanors-emr.org/), and some interface builders from Wolfgang Britz (GGIG), Erwin Kalvelagen (Packaging Optimization Applications), the Gtree developers GAMS Simulation Environment (GSE), and a plug in GAMS Mode for Emacs by Shiro Takeda. Additionally there may be others there that I missed.

6 McCarl Guide, Documentation and the Future

I have been asked by various parties recently about the status of the McCarl guide and more generally GAMS documentation. There are changes here. Namely the GAMS release notes state "The contents of GAMS User's Guide and the McCarl (Expanded) User's Guide have been merged, revised, and reorganized as **User's Guide** as well as **A GAMS Tutorial by Richard E. Rosenthal**. Also other parts of the documentation has been reorganized and are now more closely integrated." Also that "The McCarl GAMS User Guide (CHM and PDF) can now be found in the mccarl/ subdirectory in the distribution."

So what does this all mean. Well, first, the McCarl guide is no longer the official GAMS documentation. In fact, it has now moved into the "contributed documentation" section of the GAMS website. However, a lot of the content has been moved into a new documentation that was developed within GAMS Corporation. In discussions with people at GAMS the main reason for creation of the new document arises from their conclusion that the McCarl guide was not sufficiently concise plus it was incomplete in places. Also despite my invitations the GAMS employees never really bought into maintaining and expanding the guide as it was.

I feel their move to a new documentation was good for the user community in that GAMS now owns the documentation. This means they will keep it up including features that I would never know about. I also think it is good as I must admit my 70th birthday will occur later this year leading to the question of: How long will I bother to keep up the McCarl guide?

On the other hand, I must admit that beauty is in the eye of the beholder and there are certainly cases in the new documentation where for me I find a lack of beauty with the conciseness and technical orientation making it impenetrable. I have always felt the need to translate the release notes for communication to a wider user group and I see such aspects in the documentation. I have not quite decided whether I will try to keep adding to the McCarl guide in the face of these developments. I do think keeping it up would be useful as aspects of its presentation are more accessible to certain groups of users. Nevertheless, even if I do try to keep it up, I will not try to be as comprehensive as I have in the past and only include what I consider major elements. Regarding the above, I would not add anything but the Python section as the other features to me are of pretty narrow interest so are amply supported by the "official" document.

7 Basic, Advanced and Combined courses offered soon

This year I will again be teaching my family of GAMS courses for basic and advanced users. These courses will be offered in early June when there will still be some snow up in the high country and Arapahoe basin may even still be open for skiing. Dates, times and content are

- Basic to Advanced GAMS class June 11, 2018- June 15, 2018 (5 days) in the Colorado mountains at Dillon (near Frisco and Breckenridge). The course spans from Basic topics to an Advanced GAMS class. Details are found at https://www.GAMS.com/fileadmin/news-events/mccarl_Basic_to_Advanced_GAMS_Modeling2018.pdf.
- Basic GAMS class June 11, 2018- June 13, 2018 (3 days) in the Colorado mountains at Dillon (near Frisco and Breckenridge). The course starts assuming no GAMS background. Details are given at https://www.GAMS.com/fileadmin/news-events/mccarl_basic_2018.pdf.
- Advanced GAMS class June 13, 2018- June 15, 2018 (3 days) in the Colorado mountains at Dillon (near Frisco and Breckenridge). The course is for users who have a GAMS

background. Details are found at https://www.GAMS.com/fileadmin/news-events/mccarl_advanced_2018.pdf .

Further information and other courses are listed on http://www.GAMS.com/courses.htm . Note I also give custom courses for individual groups a couple of times a year.

8 Unsubscribe or subscribe to future issues of this newsletter

Please unsubscribe through the web form available at:
http://app.streamsend.com/public/XLmY/5eq/subscribe

Those who wish to subscribe to future issues can do this through the newsletter section of http://www.GAMS.com/maillist/index.htm.

This newsletter is not a product of GAMS Corporation although it is distributed with their cooperation.

March 7, 2018