Gather-Update-Solve-Scatter (GUSS)

1 Introduction

The purpose of this chapter is to detail an extension of the GAMS modeling system that allows collections of models (parameterized exogenously by a set of samples or indices) to be described, instantiated and solved efficiently.

As a specific example, we consider the parametric optimization problem $\mathscr{P}(s)$ defined by:

$$\min_{x \in X(s)} f(x;s) \quad \text{s.t.} \quad g(x;s) \leq 0 \tag{1.1}$$

where $s \in S = \{1, \dots, K\}$. Note that each scenario s represents a different problem for which the optimization variable is x. The form of the constraint set as given above is simply for concreteness; equality constraints, range and bound constraints are trivial extensions of the above framework. Clearly the problems $\mathscr{P}(s)$ are interlinked and we intend to show how such problems can be easily specified within GAMS and detail one type of algorithmic extension that can exploit the nature of the linkage. Other extensions of GAMS allow solves to be executed in parallel or using grid computing resources.

Note that in our description we will use the terms indexed, parameterized and scenario somewhat interchangeably.

An extended version of this chapter containing several examples is available as a paper at `http://www.gams.com/modlib/adddocs/gusspaper.pdf`.

2 Design Methodology

One of the most important functions of GAMS is to build a model instance from the collection of equations (i.e. an optimization model defined by the GAMS keyword `MODEL`) and corresponding data (consisting of the content of GAMS (sub)sets and parameters). Such a model instance is constructed or generated when the GAMS execution system executes a `SOLVE` statement. The generated model instance is passed to a solver which searches for a solution of this model instance and returns status information, statistics, and a (primal and dual) solution of the model instance. After the solver terminates, GAMS brings back the solution into the GAMS database, i.e. it updates the level (`.L`) and marginal (`.M`) fields of variable and equation symbols used in the model instance. Hence, the `SOLVE` statement can be interpreted as a complex operator against the GAMS database. The model instance generated by a `SOLVE` statement only lives during the execution of this one statement, and hence has no representation within the GAMS language. Moreover, its structure does fit the relational data model of GAMS. A model instance consists of vectors of bounds and right hand sides, a sparse matrix representation of the Jacobian, a representation of the non-linear expressions that allow the efficient calculation of gradient vectors and Hessian matrices and so on.

This chapter is concerned with solving collections of models that have similar structure but modified data. As an example, consider a linear program of the form:

$$\min c^T x \quad \text{s.t.} \quad Ax \geq b,\ \ell \leq x \leq u.$$

The data in this problem is (A, b, c, ℓ, u). Omitting some details, the following code could be used within GAMS to solve a collection of such linear programs in which each member of the collection has a different A matrix and lower bound ℓ:

```
Set i / ... /, j / ... /;
Parameter
```

```
3      A(i,j), b(i);
4   Variable
5      x(j), z, ...;
6   Equation
7      e(i), ...;
8   e(i).. sum(j, A(i,j)*x(j)) =g= b(i);
9   ...
10  model mymodel /all/;
11
12  Set s / s1*s10 /
13  Parameter
14     A_s(s,i,j) Scenario data
15     xlo_s(s,j) Scenario lower bound for variable x
16     xl_s(s,j)  Scenario solution for x.l
17     em_s(s,i)  Scenario solution for e.m;
18  Loop(s,
19    A(i,j) = A_s(s,i,j);
20    x.lo(j)= xlo_s(s,j);
21    solve mymodel min z using lp;
22    xl_s(s,j) = x.l(j);
23    em_s(s,i) = e.m(i);
24  );
```

Summarizing, we solve one particular model (`mymodel`) in a loop over `s` with an unchanged model rim (i.e. the same individual variables and equations) but with different model data and different bounds for the variables. The change in model data for a subsequent solve statement does not depend on the previous model solutions in the loop.

The purpose of this new Gather-Update-Solve-Scatter manager or short GUSS is to provide syntax at the GAMS modeling level that makes an instance of a problem and allows the modeler limited access to treat that instance as an object, and to update portions of it iteratively. Specifically, we provide syntax that gives a list of data changes to an instance, and allows these changes to be applied sequentially to the instance (which is then solved without returning to GAMS). Thus, we can simulate a limited set of actions to be applied to the model instance object and retrieve portions of the solution of these changed instances back in the modeling environment.

Such changes can be done to any model type in GAMS, including nonlinear problems and mixed integer models. However, the only changes we allow are to named parameters appearing in the equations and lower and upper bounds used in the model definition.

Thus, in the above example GUSS allows us to replace lines 18-24 by

```
Set dict / s.   scenario. ''
           A.   param.    A_s
           x.   lower.    xlo_s
           x.   level.    xl_s
           e.   marginal. em_s   /;
solve mymodel min z using lp scenario dict;
```

The three dimensional `set dict` (you can freely choose the name of this symbol) contains mapping information between symbols in the model (in the first position) and symbols that supply required update data or store solution information (in the third position), and the type of update/storing (in the second position). An exception to this rule is the tuple with label `scenario` in the second position. This tuple determines the symbol (in the first position) that is used as the scenario index. This scenario symbol can be a multidimensional set. A tuple in this set represents a single scenario. The remaining tuples in the `set dict` can be grouped into input and output tuples. Input tuples determine the modifications of the model instance prior to solving, while output tuples determine which part of the solution gets saved away. The following keywords can be used in the second position of the set `dict`:

Input:

`param`:	Supplies scenario data for a parameter used in the model
`lower`:	Supplies scenario lower bounds for a variable
`upper`:	Supplies scenario upper bounds for a variable
`fixed`:	Supplies scenario fixed bounds for a variable

Output:

`level`:	Stores the levels of a scenario solution of variable or equation
`marginal`:	Stores the marginals of a scenario solution of variable or equation

Sets in the model cannot be updated. GUSS works as follows: GAMS generates the model instance for the original data. As with regular `SOLVE` statements, all the model data (e.g. parameter `A`) needs to be defined at this time. The model instance with the original data is also called the *base case*. The solution of the base case is reported back to GAMS in the regular way and is accessible via the regular `.L` and `.M` fields after the `SOLVE` statement. After solving the base case, the update data for the first scenario is applied to the model. The tuples with `lower`, `upper`, `fixed` update the bounds of the variables, whereas the tuples with `param` update the parameters in the model. The scenario index `k` needs to be the first index in the parameters mapped in the `set dict`. The update of the model parameters goes far beyond updating the coefficients of the constraint matrix/objective function or the right hand side of an equation as one can do with some other systems. GAMS stores with the model instance all the necessary expressions of the constraints, so the change in the constraint matrix coefficient is the result of an expression evaluation. For example, consider a term in the calculation of the cost for shipping a variable amount of goods `x(i,j)` between cities `i` and `j`. The expression for shipping cost is `d(i,j)*f*x(i,j)`, i.e. the distance between the cities times a freight rate `f` times the variable amount of goods. In order to find out the sensitivity of the solution with respect to the freight rate `f`, one can solve the same model with different values for `f`. In a matrix representation of the model one would need to calculate the coefficient of `x(i,j)` which is `d(i,j)*f`, but with GUSS it is sufficient to supply different values for `f` that potentially result in many modified coefficient on the matrix level. The evaluation of the shipping cost term and the communication of the resulting matrix coefficient to the solver are done reliably behind the scenes by GUSS.

After the variable bound and the model parameter updates have been applied and the resulting updates to the model instance data structures (e.g. constraint matrix) has been determined, the modified model instance is passed to the solver. Some solvers (e.g. Cplex, Gurobi, and Xpress) allow modifying a model instance. So in such a case, GUSS only communicates the changes from the previous model instance to the solver. This not only reduces the amount of data communicated to the solver, but also, in the case of an LP model, allows the solver to restart from an advanced basis and its factorization. In the case of an NLP model, this provides initial values. After the solver determines the solution of a model instance, GUSS stores the part of the solution requested by the output tuples of `dict` to some GAMS parameters and continues with the next scenario.

3 GUSS Options

The execution of GUSS can be parameterized using some options. Options are not passed through a solver option file but via another tuple in the `dict` set. The keyword in the second position of this tuple is `opt`. A one dimensional parameter is expected in the first position (or the label `''`). This parameter may contain some of the following labels with values:

`OptfileInit`:	Option file number for the first solve
`Optfile`:	Option file number for subsequent solves
`LogOption`:	Determines amount of log output:
	0 - Moderate log (default)
	1 - Minimal log
	2 - Detailed log
`SkipBaseCase`:	Switch for solving the base case (0 solves the base case)
`UpdateType`:	Scenario update mechanism:
	0 - Set everything to zero and apply changes (default)
	1 - Reestablish base case and apply changes
	2 - Build on top of last scenario and apply changes
`RestartType`:	Determines restart point for the scenarios
	0 - Restart from last solution (default)
	1 - Restart from solution of base case
	2 - Restart from input point
`NoMatchLimit`:	Limit of unmatched scenario records (default 0)

For the example model above the `UpdateType` setting would mean:

```
UpdateType=0:  loop(s, A(i,j) = A_s(s,i,j))
UpdateType=1:  loop(s, A(i,j) $= A_s(s,i,j))
UpdateType=2:  loop(s, A(i,j) = A_base(i,j);
                       A(i,j) $= A_s(s,i,j))
```

The option `SkipBaseCase=1` allows to skip the base case. This means only the scenarios are solved and there is no solution reported back to GAMS in the traditional way. The third position in the `opt`-tuple can contain a parameter for storing the scenario solution status information, e.g. model and solve status, or needs to have the label `''`. The labels to store solution status information must be known to GAMS, so one needs to declare a set with such labels. The following solution status labels can be reported:

```
domusd    iterusd  objest     nodusd     modelstat  numnopt
numinfes  objval   rescalc    resderiv   resin      resout
resusd    robj     solvestat  suminfes
```

The following example shows how to use some of the GUSS options and the use of a parameter to store some solution status information:

```
Set h solution headers / modelstat, solvestat, objval /;
Parameter
   o / SkipBaseCase 1, UpdateType 1, Optfile 1 /
   r_s(s,h) Solution status report;
Set dict / s.   scenario. ''
           o.   opt.      r_s
           a.   param.    a_s
           x.   lower.    xlo_s
           x.   level.    xl_s
           e.   marginal. em_s   /;
solve mymodel min z using lp scenario dict;
```

4 Implementation Details

This section describes some technical details that may provide useful insight in case of unexpected behavior.

GUSS changes all model parameters mentioned in the `dict` set to variables. So a linear model can produce some non-linear instructions (e.g. `d(i,j)*f*x(i,j)` becomes a non-linear expression since `f` becomes a variable in the model instance given to GUSS). This also explains why some models compile without complaint, but if the model is used in the context of GUSS, the compile time check of the model will fail because a parameter that is turned into a variable cannot be used that way any more. For example, suppose the model contains a constraint `e(i).. sum(j$A(i,j), ...)`. If `A(i,j)` is a parameter in the regular model, the compiler will not complain, but if `A` becomes a parameter that shows up in the first position of a `param` tuple in the `dict` set, the GAMS compiler will turn `A` into a variable and complain that an endogenous variable cannot be used in a `$`-condition.

The sparsity pattern of a model can be greatly effected by GUSS. In a regular model instance GAMS will only generate and pass on non-zero matrix elements of a constraint `e(i).. sum(j, A(i,j)*x(j)) ...`, so the sparsity of `A` determines the sparsity of the generated model instance. GUSS allows to use this constraint with different values for `A` hence GUSS cannot exclude any of the pairs `(i,j)` and generate a dense matrix. The user can enforce some sparsity by explicitly restricting the `(i,j)` pairs: `e(i).. sum(ij(i,j), A(i,j)*x(j)) ...`

The actual change of the GAMS language required for the implementation of GUSS is minimal. The only true change is the extension of the `SOLVE` statement with the term `SCENARIO dict`. Existing language elements have been used to store symbol mapping information, options, and model result statistics. Some parts of the GUSS presentation look somewhat unnatural, e.g. since `dict` is a three dimensional `set` the specification the scenario `set` using keyword `scenario` requires a third dummy label `''`. However, this approach gives maximum flexibility for future extension, allows reliable consistency checks at compile and execution time, and allows to delay the commitment for significant and permanent syntax changes of a developing method to handle model instances at a GAMS language level.