# GDXMRW: Interfacing GAMS and MATLAB

Michael C. Ferris* Rishabh Jain† Steven Dirkse‡

February 7, 2011

**Abstract**

This document briefly describes GDXMRW, a suite of utilities to import/export data between GAMS and MATLAB (both of which the user is assumed to have already) and to call GAMS models from MATLAB and get results back in MATLAB. The software gives MATLAB users the ability to use all the optimization capabilities of GAMS, and allows visualization of GAMS models directly within MATLAB. As of GAMS Distribution 23.4, the most recent version is now included as part of GAMS.

## 1 Introduction

Optimization is becoming widely used in many application areas as can be evidenced by its appearance in software packages such as Excel and MATLAB. While the optimization tools in these packages are useful for small-scale nonlinear models (and to some extent for large linear models), the lack of a capability to compute automatic derivatives makes them impractical for large scale nonlinear optimization. In sharp contrast, modeling languages such as GAMS and AMPL have had such a capability for many years, and have been used in many practical large scale nonlinear applications.

On the other hand, while modeling languages have some capabilities for data manipulation and visualization (e.g. Rutherford's GNUPLOT), to a large extent specialized software tools like Excel and MATLAB are much better at these tasks.

This paper describes a link between GAMS and MATLAB. The aim of this link is two-fold. Firstly, it is intended to provide MATLAB users with a sophisticated nonlinear optimization capability. Secondly, the visualization tools of MATLAB are made available to a GAMS modeler in an easy and extendable manner so that optimization results can be viewed using any of the wide variety of plots and imaging capabilities that exist in MATLAB.

*Computer Sciences Department, University of Wisconsin – Madison, 1210 West Dayton Street, Madison, Wisconsin 53706 (ferris@cs.wisc.edu)

†Industrial and System Engineering, University of Wisconsin - Madison, 3270 Mechanical Engineering, 1513 University Avenue, Madison, Wisconsin 53706 (jain5@wisc.edu)

‡GAMS Development Corp., 1217 Potomac Street, NW, Washington, DC 20007 (sdirkse@gams.com)

## 2 Installation

This section describes the installation procedure for all machines. The following section describes the testing procedure for verifying a correct installation.

First of all, you need to install both MATLAB and GAMS on your machine. For brevity, we will assume that the GAMS system (installation) directory is (for Windows)

```
c:\gams
```

and for non-Windows systems:

```
/usr/local/gams
```

All of the utilities come as a part of the GAMS distribution, so to use them you have only to add the GAMS directory to the MATLAB path. One way to do this is from the MATLAB command prompt, as follows:

```
>> addpath 'C:\gams'; savepath;
```

OR this can be done by following these steps:

1. Start MATLAB
2. Click on 'File' tab.
3. Now click on 'Set Path'
4. Click on 'Add Folder'
5. Select GAMS directory and click 'OK'.
6. Save it and then close it.

## 3 Testing

The GAMS system comes with some tests that you should run to verify the correct configuration and operation of the GDXMRW utilities. In addition, these tests create a log file that can be useful when things don't work as expected. To run the tests, carry out the following steps.

1. Create a directory to run the tests in, e.g.

   ```
   % mkdir \tmp
   ```

2. Extract the test models and supporting files from the GAMS test library into the test directory.

   ```
   % cd \tmp
   % testlib gdxmrw03
   % testlib gdxmrw04
   % testlib gdxmrw05
   ```

3. Execute the GAMS files `gdxmrw03` and `gdxmrw04`. These files test that 'rgdx' and 'wgdx' are working properly. In addition to calling MATLAB in batch mode, they verify that the data are read and written as expected and give a clear indication of success or failure.

4. The GAMS file `gdxmrw05` tests the 'gams' utility. Like the other tests, it can be run in batch mode. You can also run it interactively by starting MATLAB, making `tmp` the current directory, and running the script "testinst.m".

   ```
   >> testinst
   ```

   In addition to messages indicating success or failure, this test produces a log file `testinstlog.txt` that will be useful in troubleshooting a failed test.

# 4 Data Transfer

This paper suggests a better approach to import and export data from GAMS to MATLAB using GDX files. GDX is GAMS Data Exchange file. This is a platform independent file that stores data efficiently in binary format. Unlike flat files, which might be machine dependent, GDX file are machine independent. Data in GDX file is stored in sparse format and is consistent with no duplication, contradiction or syntax errors, making it a better tool to store data than flat files. In this paper we are going to explain three MATLAB routines, namely 'rgdx', 'wgdx' and 'gams'. The first two are used to read and write data from a GDX file into MATLAB and the third routine will take user input to execute a gams model from MATLAB and get results back in MATLAB. All of these routines are such designed to get data in almost same form as is stored in a GDX file.

One of the most important features of a GDX file that we are going use in this paper is its UEL, Unique Element List. GDX files have only one global UEL and every element is mapped to this UEL. Thus 'rgdx' output will contain a global UEL as one of its fields. To write data into a GDX file, a user can enter a local UEL for each symbol and 'wgdx' will create one global UEL; otherwise it will create a default UEL ranging from 1 to n.

## 4.1 rgdx

rgdx is the MATLAB utility to import data from a GDX file. It takes structural input and returns data back in the form of a structure. This is a very flexible routine as it gives user control over the output data structure. `rgdx` can read set/parameter/equation/variable from a GDX file and display results in either full/dense or sparse form. A user can also perform a filtered read to read only certain specific elements of a symbol. It can also perform compression to remove

extra zeros.

This routine can take up to two arguments. The first argument is a string input containing the GDX file name. It can be with or without the '.gdx' file extension. If you call this routine with only the GDX file name as an argument then the 'uels' field of output structure will be the global UEL of the GDX file and the rest of the fields of the output structure will be NULL. The second argument is a structure input containing information regarding the desired symbol. The syntax for this call will look like:

x = rgdx('fileName', structure);

As an example, we read a 3D parameter, 'test3' from 'sample.gdx'. Here we want to display this parameter in full format but without redundant zeros. e.g.

```
>> s.name = 'test3';
>> s.form = 'full';
>> s.compress = true;
>> x = rgdx('sample', s)

x =

    name: 'test3'
    type: 'parameter'
     dim: 3
     val: [4x2x2 double]
    form: 'full'
    uels: {{1x4 cell}  {1x2 cell}  {1x2 cell}}

>> x.val

ans(:,:,1) =

     3     4
     4     5
     5     6
     6     7


ans(:,:,2) =

     4     5
     5     6
     6     7
     7     8
```

```
>> x.uels{1}

ans =

    '1'    '2'    '3'    '4'

>> x.uels{2}

ans =

    'j1'    'j2'

>> x.uels{3}

ans =

    'k1'    'k2'
```

In the following subsections we will explain the input and output structures. Please note that except for the 'name' and 'uels' fields, all other string fields take case insensitive input. All boolean fields can also be entered as string values as well.

#### 4.1.1 Input structure

To read a symbol from a GDX file we just need to know its name in string format. Thus, the only mandatory field of the input structure is 'name'. e.g.

```
>> s.name = 'test3';
```

There are several other optional fields of the input structure that give user more control over the output sturcture. These optional fields are as follows:

1. form
   This field represents the form of the output data. Output data can be either in 'full' or 'dense' form or it can be in [i, j,.., val] form. In this paper we will label [i, j,.., val] as 'sparse'. A user can enter it as string input with value 'full' or 'sparse'. e.g.

   ```
   >> s.form = 'full';
   ```

   By default the data will be in 'sparse' format.

2. compress
   By default the uels in the output structure will be a global UEL of the GDX file and the 'val' field data will be indexed to this UEL. The `rgdx` routine allows a user to remove rows and columns with all zeros from the 'val' data matrix and re-indexes the uels accordingly. This is called

compression of the data. This can be achieved by setting compress as true in the input structure. Valid values for this field are true and false, either in logical form or in string form. e.g.

```
>> s.compress = 'true';
```

3. uels
   This input field is used to perform a filtered read i.e. output data matrix will contain values only corresponding to the entered uels. Filtered read is very useful if user just wants certain specific set of data. Uels should be entered in cell array form. It has to be in 1xN form with each column being a cell array representing the uels for that dimension. Each column can have strings, doubles or combinations of both. It also allow user to enter double data in shorthand notation or a 1 x N matrix. For example, in the previous example we can perform a filtered read to get data corresponding to only the '1', '3' elements of the first index of the parameter 'test3'.

```
>> s.uels = {{1 3}, {'j1', 'j2'}, {'k1', 'k2'}};
>> s.compress = false;
>> x = rgdx('sample', s)

x =

    name: 'test3'
    type: 'parameter'
     dim: 3
     val: [2x2x2 double]
    form: 'full'
    uels: {{1x2 cell}  {1x2 cell}  {1x2 cell}}

>> x.val

ans(:,:,1) =

     3     4
     5     6


ans(:,:,2) =

     4     5
     6     7
```

Here it should be noted that we turned off compression while performing the filtered read. This is necessary because the filtered read will give data

in accordance with the entered uels and the output uels will be the same as the input uels; thus compression is not possible.

4. field
   This field is required when variables or equations are to be read from a GDX file. Sets and parameters in the GDX file do not have any field value but variables and equations have 5 fields namely, level, marginal, lower, upper and scale. Thus, it may be useful to enter field as an input when reading an equation or a variable. A user can enter it as a string with valid values being 'l/m/up/lo/s'. e.g.

   ```
   >> s.field = 'm';
   ```

   By default, the output will be the level value of a variable or an equation.

5. ts
   This represents the text string associated with the symbol in the GDX file. If a user sets this field as 'true', then the output structure will have one more string field 'ts' that contains the text string of the symbol. e.g.

   ```
   >> s.ts = true;
   ```

6. te
   GAMS allows a modeler to enter text elements for a set. Similarly to the 'ts' field, if a user sets 'te' to be true in the input structure, then the output structure will contain one more field representing the text elements for that symbol. Please note that text elements only exist for 'sets'. e.g.

   ```
   >> s.te = true;
   ```

#### 4.1.2 Output Structure

As mentioned earlier, output of the `rgdx` routine will be in structure form. This structure is very similar to the input structure. To get information regarding any symbol, we always need to display its basic characteristics, such as its name, type, value, uels, form, etc. An output structure will always have these fields. The required fields are as follows:

1. name
   It is same as that entered in the input structure name field i.e. the symbol name in the GDX file.

2. val
   It represents the the value matrix of the symbol. To save MATLAB memory by default it will be in 'sparse' format. e.g.

```
>> s = rmfield(s, 'form');
>> s

s =
        name: 'test3'
    compress: 0

>> x = rgdx('sample', s)

x =

    name: 'test3'
    type: 'parameter'
     dim: 3
     val: [16x4 double]
    form: 'sparse'
    uels: {{1x8 cell}  {1x8 cell}  {1x8 cell}}
```

Here val is a 16x4 double matrix. As it is a parameter; thus the last column of the sparse matrix will represent the value and the rest (i.e. the first three columns) will represent its index. Please note that in the case of 'set', the number of columns in the sparse matrix will be equal to its dimension i.e. it doesn't have a column representing its value. Here, the presence of each row in the output 'val' field corresponds to the existence of a set element at that index. This is represented as a 1 or a zero in case of a 'full' matrix.

3. form
   It represents the format in which the 'val' field is being displayed. As mentioned earlier it can be either in 'full' or 'sparse' form.

4. type
   Whlie reading a symbol from a GDX file it is often very useful to know its type. The `rgdx` routine is designed to read set, parameter, variable and equation. This field will store this information as a string.

5. uels
   This represents the unique element listing of the requested symbol in the form of a cell array. It is a 1 x N cell array, where N is the dimension of the symbol. Each column of this array consists of string elements. By default, the output uels will be the same as the global uel of the GDX file, but it can be reduced to element specific local uels if compress is set to be true in the input structure. If a user is making a filtered read, i.e. calling rgdx with input uels then the output uels will be essentially the same as the input uels.

6. dim

It is a scalar value representing the dimension of the symbol.

Apart from these necessary fields there are a few additional fields as well. They are as follows:

7. field
   If we are reading variables or equations, then it becomes useful to know which field we had read. This information is displayed via this field in the form of a string.

8. ts
   It display the explanatory text string associated with the symbol. This field only exists in the output structure if the 'ts' field is set as 'true' in the input structure.

9. te
   It is an N dimensional cell array representing the text elements associated with each index of the set. This field only exists in the output structure if the 'te' field is set as true in the input structure and the symbol is a set.

## 4.2 wgdx

`wgdx` is a MATLAB routine to create a GDX file containing MATLAB data. Similar to the `rgdx` routine, it takes input in structural form but it can write multiple symbols into a single GDX file in one call. Its first argument is a string input for the GDX file name to be created. Similarly to rgdx, it can be with or without the '.gdx' file extension. The rest of the arguments are structural input, each containing data for different symbols to be writen in the GDX file. e.g.

```
>>wgdx('fileName', s1, s2 ...);
```

If the GDX file already exists in the MATLAB current directory, then `wgdx` will overwrite it; otherwise a new file will be created. After a successful run, it doesn't return anything back into MATLAB. Most of the fields of its input structure are the same as those of the `rgdx` output structure. e.g.

```
>> s.name = 'l';
>> s.uels = {{'i1', 'i2', 'i3'}, {'j1',  'j2'}};
>> c.name = 'par';
>> c.type = 'parameter';
>> c.val = eye(3);
>> c.form = 'full';
>> c.ts = '3 x 3 identity';
>> wgdx('foo', s, c)
```

Here we used the `wgdx` routine to create foo.gdx that contains a set 'l' and a parameter 'par'. In the following section we will explain the input fields in detail.

### 4.2.1 Input Structure

Necessary fields required to represent any symbol are as follows:

1. name
   It is a string input representing the name of the symbol.

2. val
   It represents the value matrix of the parameter or set. It can be entered in either full or sparse format, whichever is convenient to the user, but make sure to specify the corresponding format as string input in the 'form' field. By default the value matrix is assumed to be in sparse format.

3. type
   It is a string input to specify the type of the symbol. The `wgdx` routine can write a set or parameter into the GDX file. In the previous example, we didn't specify the type for structure 's' because by default it is assumed to be a set.

4. form
   This is a string input representing the format in which the val matrix has been entered. By default it is assumed that the data is specified in sparse format.

5. uels
   Similarly to the rgdx uels field. this represents the local unique element listing of the symbol in an 1 x N cell array form. Each column of this cell array can contain string or double or both. If a user enters the structure with only two fields, name and uels, as in the previous example (structure s), then the wgdx call will create a full set corresponding to the global uels. i.e.

   ```
   set a /i1*i3/;
   set b /j1*j2/;
   set l(a,b);
   l(a,b) = yes;
   ```

   Optional fields are as follow:

6. dim
   This field is useful when a user wants to write a zero dimensional or 1 dimensional data in full format. As every data matrix in MATLAB is at least 2D, it becomes necessary to indicate its dimension for writing purposes.

7. ts
   This is the text string that goes with the symbol. If nothing is entered then 'MATLAB data from GDXMRW' will be written in the GDX file.

# 5 Calling GAMS model from MATLAB

Until now we have discussed the data Import/Export utility between MATLAB and GAMS. In this section, we will discuss a new MATLAB utility `'gams'` that initializes a GAMS model with MATLAB data then executes GAMS on that model and bring the results back into MATLAB. This 'gams' routine is based on the same design as `rgdx` and `wgdx` but instead it does everything in one call. This routine can take multiple input arguments and can return multiple output arguments. Its standard syntax is as follows:

```
>>  [x1, x2, x3] = gams('model', s1, s2.., c1, c2..);
```

Here note that the first argument of `gams` is the GAMS model name plus any user specific command line settings. If a user wants to solve the given model (in this case found in qp.gms) using a different solver then it can be done by adding that solver to the GAMS model name as "qp nlp=baron". This feature allows a user to change the execution time behaviour of the model.

The rest of the input arguments of GAMS are structures. Their positioning is not important. These structures are of two kinds, one similar to the input structure of `wgdx` and the other structure will have just two string fields, name and val. This latter structure is used to set or overwrite values in the model using the "$set" variables syntax of GAMS. We will explain it in detail a later section.

The first step is to generate a working GAMS model. For example, we can set up a simple model file to solve a quadratic program

$$\begin{array}{rl} \min_x & \frac{1}{2}x^TQx + c^Tx \\ \text{subject to} & Ax \geq b, x \geq 0 \end{array}$$

The GAMS model for this quadratic problem is as follows:

```
$set matout "'matsol.gdx', x, dual ";

set i /1*2/,
    j /1*3/;
alias (j1,j);

parameter
Q(j,j1) /
        1 .1 1.0
        2 .2 1.0
        3 .3 1.0 /,
A(i,j) /
        1 .1 1.0
        1 .2 1.0
        1 .3 1.0
        2 .1 -1.0
```

```
        2 .3 1.0 /,
b(i) /
        1 1.0
        2 1.0 /
c(j) /
        1 2.0 /;

$if exist matdata.gms $include matdata.gms

variable obj;
positive variable x(j);

equation cost, dual(i);

cost.. obj =e=
  0.5*sum(j,x(j)*sum(j1,Q(j,j1)*x(j1))) + sum(j,c(j)*x(j));

dual(i)..   sum(j, A(i,j)*x(j)) =g= b(i);

model qp /cost,dual/;

solve qp using nlp minimizing obj;

execute_unload %matout%;
```

This model can be executed directly at the command prompt by the following command

```
gams qp (for Unix/Linux)
or
gams.exe qp (for Windows)
```

or the user can simply hit the run button in the GAMSIDE. The optimal value is 0.5. In order to run the same model within MATLAB and return the solution vector x back into the MATLAB workspace, no change is required to the GAMS file. In MATLAB, all you have to do is to execute the following command:

```
>>  x = gams('qp');
```

This command will first collect the input structure data and create 'matdata.gdx' and 'matdata.gms' that contains include statements for the symbols written in a file matdata.gdx. In the previous example there is no structural input, so an empty 'matdata.gdx' file will be created and 'matdata.gms' will have just have a load statement for the GDX file but no load statements for any symbol. This is done to prevent any undesirable loading of data in the main model if there had already existed a 'matdata.gdx' or 'matdata.gms file'. After creating these two files then the *gams* routine will execute "gams qp" using a

system call. When this model is executed, another file 'matsol.gdx' will be created because of execute_unload statement in the last line of the model. Here it should be noted that any model that you want to execute using the MATLAB *gams* routine should contain

```
$set matout "'fileName.gdx', x1, x2 ";
```

either as the first line, or somewhere near the start of the model file. This is a standard GAMS $set statement, setting the value of the local variable 'matout'. The reason to have this statement near the start of the gams file is that the `gams` routine searches the file from the beginning for "$set matout" in the gams file. As these files can be very large, it is wise to have this statement near the start of the file. In this statement 'fileName' is the gdx file name that will be created containing symbols 'x1', 'x2', etc. These symbols can then be exported to MATLAB. The last line of the model should always be

```
execute_unload %matout%;
```

The purpose of setting the first and last line of the model in this manner is to specify what data the user wants to export to MATLAB in a "header" of the model. As MATLAB does not give any information about the output arguments except the number of expected arguments, we have to specify what data to export to MATLAB in the GAMS model with minimum modification to the existing model. In the previous example, there is only one output argument, thus the *gams* routine will get data for its first element from the output gdx file and store it in the MATLAB output argument.

If there are more than one output arguments:

```
>>  [x, u] = gams('qp');
```

then the *gams* routine will read the output gdx file and store its first element information of the GDX file as the first output argument of MATLAB i.e.'x' and the second element information of the GDX file in the second output argument of MATLAB i.e.'u' and so on. If the number of MATLAB output arguments is greater than the number of elements in the GDX file then `gams` will throw an error.

## 5.1 Input Structure

As mentioned earlier, the `gams` routine takes input arguments in structured form. It allows two different types of structure input. One contains the symbol data similar to the wgdx input structure, to be exported to the GDX file. The other structure will just have two string fields 'name' and 'value'. e.g.

```
>> s.name = 'Q';
>> s.val = eye(3);
>> s.form = 'full';
>> m = struct('name','m','val','2');
>> [x] = gams('qpmcp',s, m);
```

In this example both 's' and 'm' are structures but 'm' has only two fields and both are strings. The `gams` routine will use the 's' structure to create a 'matdata.gdx' file and 'm' to modify the execution command line to include "–m=2" at the end i.e. a command that will executed will be "gams qpmcp –m=2".

The structure 's' is the same as the input structure for wgdx but with two important differences. Firstly, it can be seen in the above example that 's' doesn't have any 'type' field. In `wgdx` we assume the type to be 'set' by default, but in the `gams` routine the type is assumed to be 'parameter' by default. The second change is an optional additional field (in addition to those given in Section 4.2.1) for the input structure called "load".

8. load
   It is a string input representing how the corresponding data will be loaded into the GAMS program. Depending on the value of the global option "gamso.input" (see next section) the input data will be read into GAMS in different ways. Suppose the input structure "s" has a "name" field of 'foo'. By default (where gamso.input = 'compile'), the file matdata.gms will

   ```
   $loadR foo
   ```

   The GAMS parameter (or set) foo will be replaced by the data that is in the "matdata.gdx" container and called "foo". If the data has been initialized before in the model, this will replace that intial data with the new data from "matdata.gdx". The option can also be explicitly set using

   ```
   s.load = 'replace'
   ```

   There are two other compile time load options, namely 'initialize' and 'merge'. The first is only valid if the parameter values have not been initialized in the GAMS file, otherwise an error is thrown. It uses the GAMS syntax

   ```
   $load foo
   ```

   The merge option is valid when the GAMS file being run has already initialized the parameter values. The new values in the MATLAB structure "s" are merged into the parameter simply overwriting existing values with the new values given. Explicitly, the "matdata.gms" file contains the statement

   ```
   $loadM foo
   ```

   to direct GAMS accordingly.

   Finally, if gamso.input = 'exec', the loading will occur at execution time. In this case, s.load = 'initialize' is not a valid input, the default setting is s.load = 'replace' which carries out

```
execute_load "matdata.gdx" foo
```

and the alternative setting s.load = 'merge' carries out

```
execute_loadpoint "matdata.gdx" foo
```

In this way, the data is loaded at execution time and performs an appropriate replace or merge.

## 5.2 Global input to change default behaviour

Until now we have seen how to specify different input to the `gams` routine and in this section we will see how to change the default behaviour of a `gams` call. This can be done by creating a structure "gamso" in the current workspace and adding different fields to that structure. There are currently nine fields that can be set in that structure to affect the behaviour of the program. Except the uels field, all other string fields take case insensitive data. These are as follows:

- `gamso.output`
  By default, output of the `gams` routine will be in structure form but it might be the case that a user is only interested in the data matrix i.e. val field of that structure. This can be done by setting gamso.output as 'std'. This will give only the value matrix as output. If this is not set to 'std' then output will be in the structure form described in the `wgdx` section.

  ```
  >> gamso.output = 'Std';
  >> x = gams('qp nlp=baron')

  x =
      0.5000
  ```

- `gamso.input`
  By default, the interface updates data at compile time. Thus, if execution time updates are made to the parameters before the line "$include matdata.gms" these may override the data that is provided in "matdata.gms" (i.e. from the command line). This may not be desirable. If you wish to perform execution time updates to the data, you should set `gamso.input` to 'exec'. An example is given in `do_exec.m`. To understand this example, the reader should inspect the exec.lst file at each pause statement to see the effects of the different options.

- `gamso.write_data`
  If this is set to "no", then all parameters on the call to `gams` are ignored, except the program name. This is useful for dealing with large datasets. Consider the following invocation:

  ```
  x = gams('largedata','A');
  y = gams('resolve','A');
  ```

The first call generates a file "matdata.gms" containing the elements of the matrix $A$ for use in the largedata.gms program. The second call rewrites a new "matdata.gms" file that again contains A. If we wish to save writing out A the second time we can use the following invocation:

```
x = gams('largedata','A');
gamso.write_data = 'no';
y = gams('resolve','A');
clear gamso;
```

or the equivalent invocation:

```
x = gams('largedata','A');
gamso.write_data = 'no';
y = gams('resolve');
clear gamso;
```

- `gamso.show`
  This is only relevant on a Windows platform. This controls how the "command box" that runs GAMS appears on the desktop. The three possible values are:
  - 'minimized' (default): The command prompt appears iconified on the taskbar.
  - 'invisible' : No command prompt is seen.
  - 'normal' : The command prompt appears on the desktop and focus is shifted to this box.

- `gamso.path`
  This option is used to specify fully qualified path for the gams executable. This is very useful if you have multiple versions of GAMS installed on your system and want to make sure which version you are running for the `gams` call. e.g.

  ```
  >> gamso.path = 'C:\Program Files\GAMS23.4\gams.exe';
  ```

The output of `gams` is similar to rgdx but unlike the rgdx `gams` routine it doesn't take input specific to a particular symbol. Thus it becomes important to implement a way to change the default behaviour of the output. This can be acheived by adding following field to the global structure 'gamso'. All these fields behave similar to that described in rgdx and take the same input as of rgdx.

- `gamso.compress`
- `gamso.form`

- `gamso.uels`
- `gamso.field`

This is a global option however.

# 6 Examples

In this section we will discuss a few examples of the MATLAB and GAMS interface. We will give a simple example of a nonlinear optimization problem that would benefit from this capability and describe the steps that are needed in order to use our interface in this application.

- Special values
  Following example shows how special values are handled by this interface. It can be seen that `rgdx` can retreive all these values from GDX file and display them appropriately in MATLAB.

```
>> s.name = 'special';
>> s.form = 'full';
>> s.compress = true;
>> x = rgdx('sample', s)

x =

    name: 'special'
    type: 'parameter'
     dim: 1
     val: [4x1 double]
    form: 'full'
    uels: {{1x4 cell}}

>> x.val

ans =

                -Inf
                 NaN
   3.141592653589793
                 Inf
```

- Variables and Equations
  In an optimization problem, we are not only interested in level value of variables and equations but also in their marginal values, lower and upper bounds. This interface gives its user ability to read any of these values into MATLAB. By default `rgdx` and `gams` routines will read the level value of

equations and variables but this can be changed very easily by using 'field' in input structure. In gams call user can also specify this in '$set matout' statement. e.g.

```
$set matout "'matsol.gdx', x.m, dual.lo=dl ";
```

In this case the marginal value of variable 'x' will be read and lower bound of dual variable will be read and stored in 'dl'.

- Text string and Text elements
  GAMS allows its user to enter text string and explanatory text elements and all GDX file contain this information as well. Following example shows how to get these text elements in MATLAB.

```
>> s1.name = 'el';
>> s1.te = true;
>> s1.ts = true;
>> s1.compress = true

s1 =

        name: 'el'
          te: 1
          ts: 1
    compress: 1

>> z = rgdx('sample', s1)

z =

    name: 'el'
    type: 'set'
     dim: 2
     val: [3x2 double]
    form: 'sparse'
    uels: {{1x2 cell}  {1x2 cell}}
      ts: 'This is 2D set with text elements'
      te: {2x2 cell}

>> z.te

ans =

    'element1'    'element2'
    '2.j1'                []
```

```
>> z.val

ans =

     1     1
     1     2
     2     1
```

- String elements
  One piece of information that may be needed within MATLAB is the modelstat and solvestat values generated by GAMS for the solves that it performed. This is easy to generate, and is given as the example `do_status.m`. This example is generated by taking the standard gamslib trnsport example, and adding the following lines to the end:

  ```
  $set matout "'matsol.gdx', returnStat, str ";
  set stat /modelstat,solvestat/;
  set str /'grunt', '%system.title%'/;
  parameter returnStat(stat);
  returnStat('modelstat') = transport.modelstat;
  returnStat('solvestat') = transport.solvestat;
  execute_unload %matout%;
  ```

  Note that the relevant status numbers are stored in GAMS into the parameter returnStat which is then written to matsol.gdx and read back into MATLAB using same technique as of rgdx.

  ```
  >> gamso.output = 'std';
  >> gamso.form = 'full';
  >> gamso.compress = true;
  >> s = gams('trnsport')

  s =

       1
       1
  ```

- Advanced Use: Plotting

  One of the key features of the GAMS/MATLAB interface is the ability to visualize optimization results obtained via GAMS within MATLAB.

  Some simple examples are contained with the program distribution. For example, a simple two dimensional plot with four lines can be carried out as follows. First create the data in GAMS and export it to MATLAB using `gams` routine.

```
$title  Examples for plotting routines via MATLAB

$set matout "'matsol.gdx', a, t, j, sys ";

set sys /'%system.title%'/;
set t /1990*2030/, j /a,b,c,d/;

parameter a(t,j);
a("1990",j) = 1;
loop(t,  a(t+1,j) = a(t,j) * (1 + 0.04 * uniform(0.2,1.8)); );

parameter year(*); year(t) = 1989 + ord(t);

* Omit some data in the middle of the graph:

a(t,j)$((year(t) gt 1995)*(year(t) le 2002)) = NA;

execute_unload %matout%;
```

We make an assumption that the user will write the plotting routines in the MATLAB environment. To create the plot in MATLAB, the sequence of MATLAB commands in Figure 1 should be input (saved as do_plot.m):

Figure 2 is an example created using this utility (and print -djpeg simple).

MATLAB supports extensive hard copy output or formats to transfer data to another application. For example, the clipboard can be used to transfer meta files in the PC enviroment, or encapsulated postscript files can be generated. The `help print` command in MATLAB details the possibilities on the current computing platform.

Scaling of pictures is also most effectively carried out in the MATLAB environment. An example of rescaling printed out is given in Figure 3.

Note that the output of this routine is saved as a jpeg file "rescale.jpg".

Other examples of uses of the utility outlined in this paper can be found in the "m" files:

```
do_ehl
do_obstacle
taxplot
plotit
plotngon
```

that are contained in the distribution.

```
gamso.output = 'std';
gamso.compress = true;
gamso.form = 'full';
[a,xlabels,legendset,titlestr] = gams('simple');
figure(1)

% Plot out the four lines contained in a; format using the third argument
plot(a,'+-');

% only put labels on x axis at 5 year intervals
xtick = 1:5:length(xlabels{1});
xlabels{1} = xlabels{1}(xtick);
set(gca,'XTick',xtick);
set(gca,'XTickLabel',xlabels{1});

% Add title, labels to axes
title(titlestr{1});
xlabel('Year -- time step annual');
ylabel('Value');

% Add a legend, letting MATLAB choose positioning
legend(legendset{1},0);

% match axes to data, add grid lines to plot
axis tight
grid
```

Figure 1: Simple plot in MATLAB

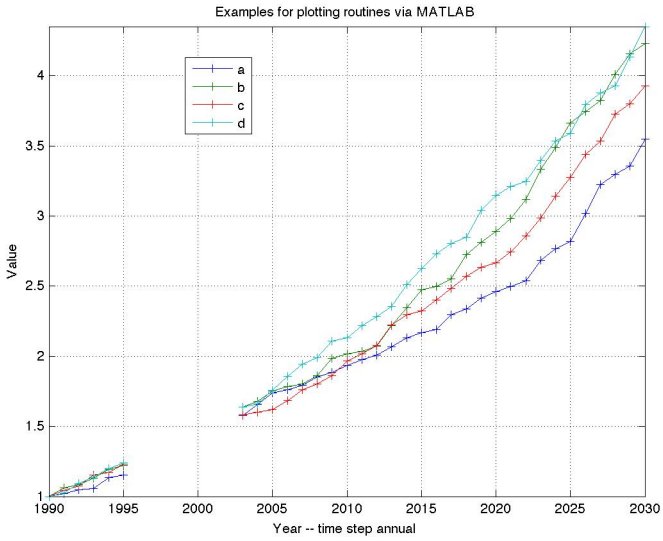


Figure 2: Simple figure created using interface

```
do_plot;
fpunits = get(gcf,'PaperUnits');

set(gcf,'PaperUnits','inches');
figpos = get(gcf,'Position');
pappos = get(gcf,'PaperPosition');
newpappos(1) = 0.25;
newpappos(2) = 0.25;
newpappos(3) = 4.0;
% get the aspect ratio the same on the print out
newpappos(4) = newpappos(3)*figpos(4)/figpos(3);

set(gcf,'PaperPosition',newpappos),
print -djpeg100 rescale.jpg
set(gcf,'PaperPosition',pappos);
set(gcf,'PaperUnits',fpunits);
```

Figure 3: Rescaling printed output from MATLAB

# 7 Acknowledgements

The author would like to thank Alexander Meeraus of GAMS corporation for constructive comments on the design and improvement of this tool.