

A Tutorial for the `gdxrrw` Package

Steve Dirkse
GAMS Development Corporation

October 18, 2012

1 Introduction

In this tutorial introduction to `gdxrrw` we carry out a small modeling exercise. We consider a common use case in which one starts with a self-contained, fully-functional GAMS model that reads from and writes to GDX. This model will run in the same way whether run in or independently from R. Without changing the model source, can control the model behavior by passing it different commands and by giving it different data sources.

It's worth noting that this tutorial assumes the user is familiar with GAMS and to a lesser extent with R.

2 First steps

Here we execute the basic steps: transferring data between GAMS and R, and running GAMS.

Before you can use any R package, you must load it. The `gdxrrw` package also depends on the GDX shared libraries found in the GAMS system directory. The `igdx` command is useful for controlling this linkage:

```
> library(gdxrrw)
> # igdx('/path/to/GAMS/sysdir') if not taken from environment
> igdx()
```

The GDX library has been not been loaded

Our starting point is a modified version of the transport model from the GAMS model library where all of the data inputs and outputs are done via GDX. Running this model is easy:

```
> gams('transport')
```

```
[1] 0
```

The model `transport` dumps all of its data (including its inputs) to GDX before it quits. There are 5 inputs: 2 sets and 3 parameters.

```
> I <- rgdx.set('outputs.gdx','I')
> J <- rgdx.set('outputs.gdx','J')
> a <- rgdx.param('outputs.gdx','a')
> b <- rgdx.param('outputs.gdx','b')
> c <- rgdx.param('outputs.gdx','c')
```

As an exercise, we first generate a GDX file whose data is identical to the original inputs, and verify that the solution with this data is unchanged:

```
> wgdxdiff('intest',list(I,J,a,b,c))
> rc <- system('gdxdiff intest inputs')
> if (0 == rc) print ("identical inputs") else print ("different inputs")
```

```
[1] "identical inputs"
```

```
> gams('transport.gms --INPUT intest.gdx --OUTPUT outtest.gdx')
```

```
[1] 0
```

```
> zlst <- rgdx('outtest.gdx',list(name='z'))
> z <- zlst$val
> zlst <- rgdx('intest.gdx',list(name='z'))
> if (zlst$val == z) print ("identical objective") else print ("different objective")
```

```
[1] "identical objective"
```

If we double all transportation costs, we can expect to double the objective.

```
> c2 <- c
> c2[, 'value'] <- c[, 'value'] * 2
> wgdxdiff('in2',list(I,J,a,b,c2))
> gams('transport.gms --INPUT in2.gdx --OUTPUT out2.gdx')
```

```
[1] 0
```

```
> zlst <- rgdx('out2.gdx',list(name='z'))
> z2 <- zlst$val
> print(paste('original=', z, ' double=', z2))
```

```
[1] "original= 153.675   double= 307.35"
```

3 Using New Data

We can define a new problem by changing the sets I and J. For example, we can use the state data in R to create a set of source and destination nodes. The state data includes populations that we can use to scale demands, and we can base transportation costs on the lat/long data for the state centers.

```
> data(state)
> src <- c('California','Washington','New York','Maryland')
> dst <- setdiff(state.name,src)
> supTotal <- 1001
> demTotal <- 1000
> srcPop <- state.x77[src,'Population']
> srcPopTot <- sum(srcPop)
> dstPop <- state.x77[dst,'Population']
> dstPopTot <- sum(dstPop)
> sup <- (srcPop / srcPopTot) * supTotal
> dem <- (dstPop / dstPopTot) * demTotal
> x <- state.center$x
> names(x) <- state.name
> y <- state.center$y
> names(y) <- state.name
> cost <- matrix(0,nrow=length(src),ncol=length(dst),dimnames=list(src,dst))
> for (s in src) {
+   for (d in dst) {
+     cost[s,d] <- sqrt((x[s]-x[d])^2 + (y[s]-y[d])^2)
+   }
+ }
```

Now that we've created the raw data for our transportation problem, we need to put it in proper form for writing out the GDX. In this example, we use a list for each symbol to write, although we could use data frames too.

```
> ilst <- list(name='I',uels=list(src),ts='supply states')
> jlst <- list(name='J',uels=list(dst),ts='demand states')
> suplst <- list(name='a',val=as.array(sup),uels=list(src),
+               dim=1,form='full',type='parameter',ts='supply limits')
> demlst <- list(name='b',val=as.array(dem),uels=list(dst),
+               dim=1,form='full',type='parameter',ts='demand quantities')
> clst <- list(name='c',val=cost,uels=list(src,dst),
+               dim=2,form='full',type='parameter',
```

```
+          ts='transportation costs')
> wgdxd.lst('inStates',list(ilst,jlst,suplst,demlst,clst))
```

Once the data is written to GDX we can call gams, as before. A model status of 1 (Optimal) indicates an optimal solution was found.

```
> gams('transport.gms --INPUT inStates.gdx --OUTPUT outStates.gdx')
```

```
[1] 0
```

```
> ms <- rgdx.scalar('outStates.gdx','modelStat')
> print(paste('Model status:',ms))
```

```
[1] "Model status: 1"
```

4 Abnormal Results

Things often fail to go as planned. A GAMS run indicates this with a nonzero return code. For example, we could point to a non-existent GDX input:

```
> rc <- gams('transport --INPUT notHere')
> if (0 == rc) print ("normal gams return") else print ("abnormal gams return")
```

```
[1] "abnormal gams return"
```

Another case to consider is an infeasible model. This is not an error: infeasible models are sometimes expected. The model status (one of many values available after a solve) can indicate an infeasible model, among other things. If we reduce the supply available from Seattle, we make the model infeasible.

```
> a2 <- a
> a2[a2$i == 'seattle','value'] <- a2[a2$i == 'seattle','value'] - 100
> wgdxd.lst('inInf',list(I,J,a2,b,c))
> gams('transport.gms --INPUT inInf.gdx --OUTPUT outInf.gdx')
```

```
[1] 0
```

```
> ms <- rgdx.scalar('outInf.gdx','modelStat')
> if (4 == ms) print ("The model is infeasible")
```

```
[1] "The model is infeasible"
```