A Tutorial for the `gdxrrw` Package

Steve Dirkse
GAMS Development Corporation

June 15, 2015

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. The GAMS source for the model is not R-specific; it works equally well with or independently of R. Without changing the model source, we can control the model behavior by passing it flags that control the GDX data input and output used.

It's worth noting that this tutorial assumes the user is familar 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 setting and querying this linkage between `gdxrrw` and GAMS. For example, the usage below checks for the GAMS system directory in the environment variable `R_GAMS_SYSDIR`.

```
> library(gdxrrw)
> igdx('')

gamsSysDir arg is empty: no GDX API there
GDX API loaded from R_GAMS_SYSDIR=/home/sdirkse/leg_alpha/gmstest
The GDX library has been loaded
GDX library load path: /home/sdirkse/leg_alpha/gmstest
```

```
> if (! igdx(silent=T)) stop ('Could not load GDX API')
```

Our starting point is a basic transport model where all of the data inputs and outputs are done via GDX. Running this model is easy once we figure out how to get it from the `doc` subdirectory of the package:

```
> gms <- system.file('doc','transport.gms',package='gdxrrw', mustWork=T)
> isWindows <- ("mingw32" == R.Version()$os)
> if (isWindows) gms <- gsub("/","\\",gms,fixed=TRUE)
> ingdx <- system.file('doc','inputs.gdx',package='gdxrrw', mustWork=T)
> if (isWindows) ingdx <- gsub("/","\\",ingdx,fixed=TRUE)
> rc <- gams(paste0(gms, " --INPUT=", ingdx))
> if (0 != rc) stop ('GAMS run failed: rc = ',rc)
```

For reference, we have included the GDX file `outputs.gdx` produced by the run above in the `doc` directory.

```
> if (! file.exists('outputs.gdx')) {
+   ogdx <- system.file('doc','outputs.gdx',package='gdxrrw', mustWork=T)
+   file.copy(ogdx,'outputs.gdx')
+ }
```

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.

```
> outgdx <- 'outputs.gdx'
> if (! file.exists(outgdx)) stop (paste('File not found:',outgdx))
> I <- rgdx.set(outgdx,'I')
> J <- rgdx.set(outgdx,'J')
> a <- rgdx.param(outgdx,'a')
> b <- rgdx.param(outgdx,'b')
> c <- rgdx.param(outgdx,'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:

```
> wgdx.lst('intest',list(I,J,a,b,c))
> rc <- system2('gdxdiff', paste('intest.gdx', ingdx), stdout=F)
> if (0 != rc) stop ('gdxdiff says intest.gdx and inputs.gdx differ: rc = ',rc)
> rc <- gams(paste(gms, '--INPUT intest.gdx --OUTPUT outtest.gdx'))
> if (0 != rc) stop ('gams failed: rc = ',rc)
> # system2('gams', 'transport.gms --INPUT intest.gdx --OUTPUT outtest.gdx')
> zlst <- rgdx('outputs.gdx',list(name='z'))
```

```
> z <- zlst$val
> # we don't need to use the intermediate zlst variable to get the val
> zz <- rgdx('outtest.gdx',list(name='z'))$val
> if (0.0 != round(z-zz,6)) stop (paste("different objectives!! ", z, zz))
```

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

```
> c2 <- c
> c2[[3]] <- c[[3]] * 2
> wgdx.lst('in2',list(I,J,a,b,c2))
> gams(paste(gms,'--INPUT in2.gdx --OUTPUT out2.gdx'))

[1] 0

> z2 <- rgdx('out2.gdx',list(name='z'))$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$x
> names(x) <- state.name
> y <- state.center$x
```

```
> 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')
> wgdx.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(paste(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. In many cases, this is indicated by a nonzero return code from a GAMS run. For example, we could point to a non-existent GDX input:

```
> rc <- gams('transport --INPUT notHere')
> if (0 == rc) print ("SHOULD NOT GET HERE") else print ("abnormal gams return as exp
```

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

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
> rows <- a2$i == 'seattle'
> a2[rows,2] <- a2[rows,2] - 100
> wgdx.lst('inInf',list(I,J,a2,b,c))
> gams(paste(gms,'--INPUT inInf.gdx --OUTPUT outInf.gdx'))

[1] 0

> ms <- rgdx.scalar('outInf.gdx','modelStat')
> if (4 == ms) print ("Reduced supply makes model infeasible")

[1] "Reduced supply makes model infeasible"
```