Introduction to R-GLPK

Louis Luangkesorn *

October 12, 2006

1 Introduction

This document introduces the use of the GLPK package[1] for R. The GNU Linear Programming Package (GLPK) is intended for solving linear programming (LP) and mixed integer programming (MIP) and other related problems. In addition, it includes facilities for converting problem information between the GNU MathProg language (a subset of the AMPL mathematical programming language), free and fixed MPS, and the CPLEX LP formats.[2] The GLPK package is an interface into the C Application Programming Interface (API) to the GLPK solver.

This document will introduce the use of the GLPK package through the use of the cannery problem from Dantzig[3] which is used in the GNU MathProg documentation.[4] The model file describing the cannery problem can be found in Appendix A.

2 Entering the model

To use `glpk`, first load the package.

```
> library(glpk)
```

Next read in the model and data.

There are several ways of entering the model. `glpk` can read the model and data in a GNU MathProg Language (GMPL) model file[5]. Alternatively, the model and data can be entered using the GLPK API.

*lluang@yahoo.com. Thanks to Leo Lopes for his comments and suggestions.

[1] Package GLPK maintained by Lopaka Lee

[2] GNU Linear Programming Kit: Reference Manual, Version 4.9 Draft, January 2006.

[3] The demand data here is from the GLPK documentation, which differs slightly from Dantzig, Linear Programming and Extensions, Princeton University Press, Princeton, NJ, 1963. The documentation demand values are used here for consistancy.

[4] GNU Linear Programming Kit: Modeling Language GNU MathProg, Version 4.9 Draft, January 2006.

[5] or MPS or CPLEX LP formats.

2.1 Reading a GNU MathProg Language model

If a GMPL model file has already been produced, it can be read directly. After setting the current directory, the model can be read using `lpx_read_model()`. `lpx_read_model()` takes three parameters:

`lpx_read_model(modelfile, datafile, outputfile)`

The `modelfile` is required. However, `datafile` and `outputfile` may be `NULL`. A `NULL datafile` would indicate that the data was in `modelfile` or that data would be entered via the API. If `modelfile` included a data section, the data in `datafile` would override data in `modelfile`. If `outputfile` was `NULL`, the data would be sent to the stdout using the routine `print`.

```
> lp <- lpx_read_model("transport.mod")

Reading model section from transport.mod...
Reading data section from transport.mod...
62 lines were read
Generating cost...
Generating supply...
Generating demand...
Model has been successfully generated
```

Then examine the problem size within R.

The rows represent the objective function as well as the supply and demand constraints.

```
> numrows <- lpx_get_num_rows(lp)
> numrows

[1] 6

> for (i in 1:numrows) {
+     print(lpx_get_row_name(lp, i))
+ }

[1] "cost"
[1] "supply[Seattle]"
[1] "supply[San-Diego]"
[1] "demand[New-York]"
[1] "demand[Chicago]"
[1] "demand[Topeka]"
```

The columns represent the decision variables, which are the units sent over the cannary-market links.

```
> numcols <- lpx_get_num_cols(lp)
> numcols

[1] 6
```

```
> for (j in 1:numcols) {
+     print(lpx_get_col_name(lp, j))
+ }

[1] "x[Seattle,New-York]"
[1] "x[Seattle,Chicago]"
[1] "x[Seattle,Topeka]"
[1] "x[San-Diego,New-York]"
[1] "x[San-Diego,Chicago]"
[1] "x[San-Diego,Topeka]"

> print(lpx_get_num_nz(lp))

[1] 18
```

After the model and data are entered, the model can then be solved using any one of many algorithms and the output would go to the specified output file. For the Simplex method, the `lpx_simplex()` takes the problem name and solves it using the Simplex method.

```
> lpx_simplex(lp)

      0:   objval =   0.000000000e+00   infeas =   1.000000000e+00 (0)
      4:   objval =   1.561500000e+02   infeas =   0.000000000e+00 (0)
*     4:   objval =   1.561500000e+02   infeas =   0.000000000e+00 (0)
*     5:   objval =   1.536750000e+02   infeas =   0.000000000e+00 (0)
OPTIMAL SOLUTION FOUND
[1] 200
```

We can then look at the solution in terms of the objective and constraints

```
> for (i in 1:numrows) {
+     print(lpx_get_row_name(lp, i))
+     print(lpx_get_row_prim(lp, i))
+ }

[1] "cost"
[1] 153.675
[1] "supply[Seattle]"
[1] 350
[1] "supply[San-Diego]"
[1] 550
[1] "demand[New-York]"
[1] 325
[1] "demand[Chicago]"
[1] 300
[1] "demand[Topeka]"
[1] 275
```

as well as the decision variables.

```
> for (j in 1:numcols) {
+     print(lpx_get_col_name(lp, j))
+     print(lpx_get_col_prim(lp, j))
+ }
```

```
[1] "x[Seattle,New-York]"
[1] 50
[1] "x[Seattle,Chicago]"
[1] 300
[1] "x[Seattle,Topeka]"
[1] 0
[1] "x[San-Diego,New-York]"
[1] 275
[1] "x[San-Diego,Chicago]"
[1] 0
[1] "x[San-Diego,Topeka]"
[1] 275
```

2.2 Using the API

If the problem data already in *R*, such as pulled from a database or the result of previous analysis, the model and the data can be specified using the API.

First create R data objects to hold the various model parameters.

```
> print("USING API")
```

```
[1] "USING API"
```

```
> canneries <- c("Seattle", "San-Diego")
> capacity <- c(350, 600)
> markets <- c("New-York", "Chicago", "Topeka")
> demand <- c(325, 300, 275)
> distance <- c(2.5, 2.5, 1.7, 1.8, 1.8, 1.4)
> dim(distance) <- c(2, 3)
> freight <- 90
```

To use the API, define a problem instance and indicate that the objective is to minimize cost.

```
> lpi <- lpx_create_prob()
> lpx_set_prob_name(lpi, "cannery API")
> lpx_set_obj_name(lpi, "Total Cost")
> lpx_set_obj_dir(lpi, LPX_MIN)
```

There are 6 columns, corresponding to the six potential cannery-market pairs whose transport the model solving for, each of which has a lower bound of zero.

```
> numlinks <- length(distance)
> nummarkets <- length(markets)
> numcanneries <- length(canneries)
> lpx_add_cols(lpi, numlinks)

[1] 1

> for (i in 1:numcanneries) {
+     cannerystartrow <- (i - 1) * nummarkets
+     for (j in 1:nummarkets) {
+         colname <- toString(c(canneries[i], markets[j]))
+         transcost <- distance[i, j] * freight/1000
+         lpx_set_col_name(lpi, cannerystartrow + j, colname)
+         lpx_set_col_bnds(lpi, cannerystartrow + j, LPX_LO, 0,
+             0)
+         lpx_set_obj_coef(lpi, cannerystartrow + j, transcost)
+     }
+ }
```

Next, we will add constraints. There are 5 constraints, two supply constraints relating to the canneries and three demand constraints relating to the markets. In addition, we will make the first row correspond to the objective function. The objective row will be free, and does not have upper or lower bounds.

```
> numcanneries <- length(canneries)
> nummarkets <- length(markets)
> lpx_add_rows(lpi, numcanneries + nummarkets + 1)

[1] 1

> lpx_set_row_name(lpi, 1, lpx_get_obj_name(lpi))
> for (i in 1:numcanneries) {
+     lpx_set_row_name(lpi, i + 1, toString(c("Supply", canneries[i])))
+     lpx_set_row_bnds(lpi, i + 1, LPX_UP, 0, capacity[i])
+ }
> for (j in 1:nummarkets) {
+     lpx_set_row_name(lpi, numcanneries + j + 1, toString(c("Demand",
+         markets[j])))
+     lpx_set_row_bnds(lpi, numcanneries + j + 1, LPX_LO, demand[j],
+         0)
+ }
```

Now, load the constraint matrix which represents the objective function and the constraints. The non-zero values of the matrix are entered as three vectors, each with one element for each non-zero value. A vector to indicate the row, a vector to indicate the column, and a vector which contains the matrix element value. Last, we call `lpx_load_matrix(lpi)` to finish. Note that in *R* the size of the vectors does not need to be prespecified, as *R* will increase the size of the vectors as necessary.

```
> ia <- numeric()
> ja <- numeric()
> ar <- numeric()
> for (i in 1:numcols) {
+     ia[i] <- 1
+     ja[i] <- i
+     ar[i] <- lpx_get_obj_coef(lpi, i)
+ }
> for (i in 1:numcanneries) {
+     cannerysupplyrow = numcols + (i - 1) * nummarkets
+     for (j in 1:nummarkets) {
+         ia[cannerysupplyrow + j] <- (i + 1)
+         ja[cannerysupplyrow + j] <- (i - 1) + numcanneries *
+             (j - 1) + 1
+         ar[cannerysupplyrow + j] <- 1
+     }
+     marketdemandrow = numcols + numcanneries * nummarkets
+     for (j in 1:nummarkets) {
+         colnum <- (i - 1) * nummarkets + j
+         ia[marketdemandrow + colnum] <- numcanneries + j + 1
+         ja[marketdemandrow + colnum] <- colnum
+         ar[marketdemandrow + colnum] <- 1
+     }
+ }
> lpx_load_matrix(lpi, length(ia), ia, ja, ar)
```

Then, examine the problem entered in the API.

```
> numrows <- lpx_get_num_rows(lpi)
> numrows

[1] 6

> numcols <- lpx_get_num_cols(lpi)
> numcols

[1] 6

> for (i in 1:numrows) {
+     print(lpx_get_row_name(lpi, i))
+ }

[1] "Total Cost"
[1] "Supply, Seattle"
[1] "Supply, San-Diego"
[1] "Demand, New-York"
[1] "Demand, Chicago"
[1] "Demand, Topeka"
```

```
> for (j in 1:numcols) {
+     print(lpx_get_col_name(lpi, j))
+ }

[1] "Seattle, New-York"
[1] "Seattle, Chicago"
[1] "Seattle, Topeka"
[1] "San-Diego, New-York"
[1] "San-Diego, Chicago"
[1] "San-Diego, Topeka"

> print(lpx_get_num_nz(lpi))

[1] 18
```

Finally solve using the simplex method and look at the solution.

```
> lpx_simplex(lpi)

      0:   objval =   0.000000000e+00   infeas =   1.000000000e+00 (0)
      4:   objval =   1.545750000e+02   infeas =   0.000000000e+00 (0)
*     4:   objval =   1.545750000e+02   infeas =   0.000000000e+00 (0)
*     5:   objval =   1.536750000e+02   infeas =   0.000000000e+00 (0)
OPTIMAL SOLUTION FOUND
[1] 200

> for (i in 1:numrows) {
+     print(lpx_get_row_name(lpi, i))
+     print(lpx_get_row_prim(lpi, i))
+ }

[1] "Total Cost"
[1] 153.675
[1] "Supply, Seattle"
[1] 325
[1] "Supply, San-Diego"
[1] 575
[1] "Demand, New-York"
[1] 325
[1] "Demand, Chicago"
[1] 300
[1] "Demand, Topeka"
[1] 275

> for (j in 1:numcols) {
+     print(lpx_get_col_name(lpi, j))
+     print(lpx_get_col_prim(lpi, j))
+ }
```

```
[1] "Seattle, New-York"
[1] 325
[1] "Seattle, Chicago"
[1] 300
[1] "Seattle, Topeka"
[1] 0
[1] "San-Diego, New-York"
[1] 0
[1] "San-Diego, Chicago"
[1] 0
[1] "San-Diego, Topeka"
[1] 275
```

Note that the solution using the API has the same objective value as the solution from when the problem was read using the GNU MathProg Language, even if the actual solution may be different. A more readable summary of the solution can be found by the command `lpx_print_sol(lpi, filename)`. The output of this is found in Appendix B.

2.3 Using API to Modify Model

Now, we will solve the version of the problem that is found in Dantzig. The demand at New York and Topeka are both 300 instead of 325 and 275. This next section will use the API to modify the problem as read through the MathProg file.

In order to examine an individual row, we need to index the rows and columns. This is done through the use of `lpx_create_index(lp)`. Then we can use the `lpx_find_row(lpi, rowname)` and `lpx_find_col(lpi, colname)`

```
> cindex <- lpx_create_index(lp)
> new_york_row = lpx_find_row(lp, "demand[New-York]")
> topeka_row = lpx_find_row(lp, "demand[Topeka]")
> new_york_row

[1] 4

> topeka_row

[1] 6

> lpx_set_row_bnds(lp, new_york_row, LPX_LO, 300, 0)
> lpx_set_row_bnds(lp, topeka_row, LPX_LO, 300, 0)
```

We can solve this modified problem.

```
> lpx_simplex(lp)
```

```
!     5:   objval =   1.512000000e+02   infeas =   0.000000000e+00
OPTIMAL SOLUTION FOUND
[1] 200

> for (i in 1:numrows) {
+     print(lpx_get_row_name(lp, i))
+     print(lpx_get_row_prim(lp, i))
+     print(lpx_get_row_dual(lp, i))
+ }

[1] "cost"
[1] 151.2
[1] 0
[1] "supply[Seattle]"
[1] 350
[1] 0
[1] "supply[San-Diego]"
[1] 550
[1] 0
[1] "demand[New-York]"
[1] 300
[1] 0.225
[1] "demand[Chicago]"
[1] 300
[1] 0.153
[1] "demand[Topeka]"
[1] 300
[1] 0.126

> for (j in 1:numcols) {
+     print(lpx_get_col_name(lp, j))
+     print(lpx_get_col_prim(lp, j))
+     print(lpx_get_obj_coef(lp, j))
+ }

[1] "x[Seattle,New-York]"
[1] 50
[1] 0.225
[1] "x[Seattle,Chicago]"
[1] 300
[1] 0.153
[1] "x[Seattle,Topeka]"
[1] 0
[1] 0.162
[1] "x[San-Diego,New-York]"
[1] 250
[1] 0.225
```

```
[1] "x[San-Diego,Chicago]"
[1] 0
[1] 0.162
[1] "x[San-Diego,Topeka]"
[1] 300
[1] 0.126
```

A Model file

TRANSPORT.MOD

```
# A TRANSPORTATION PROBLEM
#
# This problem finds a least cost shipping schedule that meets
# requirements at markets and supplies at factories.
#
#  References:
#              Dantzig, G B., Linear Programming and Extensions
#              Princeton University Press, Princeton, New Jersey, 1963,
#              Chapter 3-3.
set I;
/* canning plants */

set J;
/* markets */

param a{i in I};
/* capacity of plant i in cases */

param b{j in J};
/* demand at market j in cases */

param d{i in I, j in J};
/* distance in thousands of miles */

param f;
/* freight in dollars per case per thousand miles */

param c{i in I, j in J} := f * d[i,j] / 1000;
/* transport cost in thousands of dollars per case */

var x{i in I, j in J} >= 0;
/* shipment quantities in cases */

minimize cost: sum{i in I, j in J} c[i,j] * x[i,j];
/* total transportation costs in thousands of dollars */

s.t. supply{i in I}: sum{j in J} x[i,j] <= a[i];
/* observe supply limit at plant i */
```

```
s.t. demand{j in J}: sum{i in I} x[i,j] >= b[j];
/* satisfy demand at market j */

data;

set I := Seattle San-Diego;

set J := New-York Chicago Topeka;


param a := Seattle     350
           San-Diego   600;

param b := New-York    325
           Chicago     300
           Topeka      275;

param d :              New-York   Chicago   Topeka :=
           Seattle     2.5        1.7       1.8
           San-Diego   2.5        1.8       1.4  ;

param f := 90;

end;
```

B Output

The following is the output of the command: `lpx_print_sol(lpi, "transout.api")`

```
Problem:    cannery API
Rows:       6
Columns:    6
Non-zeros:  18
Status:     OPTIMAL
Objective:  Total Cost = 153.675 (MINimum)

No.   Row name  St Activity  Lower bound Upper bound  Marginal
--- ----------- -- --------- ----------- ----------- ---------
  1 Total Cost  B    153.675
  2 Supply , Seattle
                B        325                     350
  3 Supply , San-Diego
                B        575                     600
  4 Demand , New-York
                NL       325         325                 0.225
```

```
  5 Demand , Chicago
                NL       300        300                 0.153
  6 Demand , Topeka
                NL       275        275                 0.126

No. Column name St Activity  Lower bound Upper bound  Marginal
--- ----------- -- --------- ----------- ----------- ---------
  1 Seattle, New-York
                B        325           0
  2 Seattle, Chicago
                B        300           0
  3 Seattle, Topeka
                NL         0           0                 0.036
  4 San-Diego, New-York
                NL         0           0                 < eps
  5 San-Diego, Chicago
                NL         0           0                 0.009
  6 San-Diego, Topeka
                B        275           0

Karush-Kuhn-Tucker optimality conditions:

KKT.PE: max.abs.err. = 1.24e-014 on row 1
        max.rel.err. = 8.02e-017 on row 1
        High quality

KKT.PB: max.abs.err. = 0.00e+000 on row 0
        max.rel.err. = 0.00e+000 on row 0
        High quality

KKT.DE: max.abs.err. = 0.00e+000 on column 0
        max.rel.err. = 0.00e+000 on column 0
        High quality

KKT.DB: max.abs.err. = 0.00e+000 on row 0
        max.rel.err. = 0.00e+000 on row 0
        High quality

End of output
```