Data Input/Output in R

Baburao Kamble

September,2014

1 Introduction to Data Input/Output

Reading data into a programming language for analysis and exporting the results to some other system for report writing can be frustrating tasks that can take far more time consuming than the statistical analysis itself, even though most readers will find the latter far more appealing. Now that you have knowledge of data type and data structures, you need to put some data in to the R system. As a Researcher, you're typically faced with data that comes to you from a variety of experiments and in a variety of formats. Your task is to import the data into your tools, analyze the data.

Importing and Exporting data into R is very simple. The data format of the file is a relatively important item to import data in R. There are several methods available to import and export data files into R based on data file formats such as txt, csv, shape, images, databases, data loggers, json, etc. and a more general approach will be followed here.

2 Text Data Input/Output

Here we explore how to define a data set in an R session. We will explore common text data format (text, csv, excel)

2.1 *.TXT Data

A tab-separated values file is a simple text format for storing data in a tabular structure (e.g. database or spreadsheet data). Delimited text files (.txt), in which the TAB character (ASCII character code 009) typically separates each field of text. Tab-delimited files are text files organized around data that has rows and columns.

R uses `read.table` to reads a tab separated text file in table format and creates a data frame from it, with cases corresponding to lines and variables to fields in the file.

```
>read.table(file,header=FALSE,
sep="\t", ,)
```

file: the name of the file which the data are to be read from.

header: a logical value indicating whether the file contains the names of the variables as its first line the field separator character.

sep: the field separator character.

Write text data file

In R you can easily write out your analyzed data to a file with values delimited by commas, tabs, spaces, or other characters.

```
>write.table(myDF,"mydata.txt",sep="\t"
)
```

2.2 *.CSV Data

The comma separated values format (CSV) (is also called as character-separated values) has been used for exchanging and converting data between various spreadsheet programs for quite some time. Comma separated values text files (.csv), in which the comma character (,) typically separates each field of text. A comma-separated values (CSV) file stores tabular data (numbers and text) in plain-text form.

```
>MyData <- read.csv(file="csvdata.csv",
header=TRUE, sep=",")
```

This will create a new variable called "`MyData`." If you type in "`MyData`" at the prompt and hit enter, all of the numbers stored in the variable will be printed out. Try this, and you should see that it is difficult to make any sense out of the numbers.

All of the data are stored within the data frame as separate columns. If you are not sure what kind of variable you have then you can use the attributes command.

```
>attributes(MyData)
```

This will list all of the things that R uses to describe the variable.

Write csv data

The easiest way to write data to a file is to use write.csv(). By default, write.csv() includes row names, but these are usually unnecessary and may cause confusion.

```
>write.csv(data,"data.csv",row.names=F)
```

2.3 Excel Data

Excel (*.xls and *. Xlsx)_ is a spreadsheet data format developed by Microsoft for Microsoft Windows and Mac OS. Data is stored in rows and columns.

The best way to read an Excel file is to export it to a comma delimited file and import it using the method described in above CSV section. On windows systems you can use the RODBC package to access Excel files. The first row should contain variable/column names.

```
>library(RODBC)
>channel                                <-
>odbcConnectExcel("MyData.xls")
>mydata <- sqlFetch(channel, "mysheet")
odbcClose(channel)
```

There are other way to read and write excel file on all operating system.

XLConnect is a powerful package that allows R users to read and write Excel files in a highly integrated manner from within R. XLConnect allows you to produce formatted Excel reports, including graphics, straight from within R.

install XLConnect packages

```
>install.packages("XLConnect",
dependencies=TRUE)
```

Load the XLConnect package

```
>require(XLConnect)
```

Load the workbook

```
>myfile<-
loadWorkbook("MyExcelData.xlsx")
```

Read the data on the sheet named "Sheet2"

```
>mydata<-readWorksheet(myfile, sheet =
"Sheet2", header = TRUE)
```

check the variables

```
>head(mydata)
```

Write Excel data file

writeWorksheet() writes data into a worksheet (name or index specified as the sheet argument) of an Excel workbook (object). The startRow and startCol are both 1 by default, meaning that if they are not explicitly specified, the data will start being filled into the A1 cell of the worksheet.

Load workbook (create if not existing)

```
>wb<loadWorkbook("writeWorksheet.xlsx",
create = TRUE)
```

Create a worksheet called 'Output'

```
>createSheet(wb, name = "Output")
```

Write built-in data set 'Output' to the worksheet created above;

```
>writeWorksheet(wb,res,sheet=
"Output",rownames="Row Names")
```

save workbook (this actually writes the file to disk)

```
>saveWorkbook(wb)
```

2.4 SPSS, Stata and SAS Data files

R supports SPSS (*.sav), Stata(*.dta), and SAS(*.xpt) data files.

You will need foreign library to read above data files. The functions read.spss(), read.dta(), and read.xport() of the package foreign import SPSS, Stata, and SAS data files, respectively.

Use the following syntax to import the three types of data files:

```
>spssdata<- read.spss("Mydata.sav")
>statadata <- read.dta("Mydata.dta")
>sasdata <- read.xport("Mydata.xpt")
```

To check the size of your database, use the dim() function. You will obtain two numbers, the first one refers to the cases (rows in your database), while the second one is the number of variables (the columns of your database).

```
>dim(myfile)
```

2.5 Working with multiple files and data frames

We often encounter situations where we have data in multiple files, at different frequencies, magnitude and on different subsets of observations, but we would like to match them to one another as completely and systematically as possible. In R, the merge() command is a great way to match two data frames together.
Just read the two data frames into R

```
>mydata1=read.csv(path1,header=T)
>mydata2=read.csv(path2,header=T)
```

Then , use merge

```
>myfulldata=merge(mydata1,mydata2,by="ID")
```

As long as mydata1 and mydata2 have at least one common column (ID) with an identical name (that allows matching observations in mydata1 to observations inmydata2), this will work like a charm. It also takes three lines.

2.6 Subsetting Data

There are many reasons to remove variables from the data frame. R's subsetting operators are powerful and fast. Mastery of subsetting allows you to succinctly express complex operations in a way that few other languages can match.

Knowing that you don't want 3rd and 5th variable, you could exclude them with the statement

```
>Mydata2 <- MyData[c(-3,-5)]
```

subset() function

The subset function is available in base R and can be used to return subsets of a vector, martix, or data frame which meet a particular condition. The subset function is probably the easiest way to select variables and observations.

```
>MyData3 <- subset(MyData, Tmean >= 16 | Windspeed < 7,select=c(1:10))
```

3 Spatial Data Input/Output

3.1 Shapefile Data

The Esri shapefile, or simply a Shapefiles, is a popular geospatial vector data format for geographic information system. A shapefile is a digital vector storage format for storing geometric location and associated attribute information. Shapefiles are simple because they store the primitive geometric data types of points, lines, and polygons.

To read sahepfile you need geospatial package installed on your R. There are many geospatial packages rgdal, shapefiles, sp, maptools, fastshp). Most common geospatial package is rgdal provides bindings to Geospatial Data Abstraction Library (GDAL) (>= 1.6.3) and access to projection operations from the PROJ.4 library .

This includes reading, thinning of points and matching of points to containing shapes. The main aim for this package is to provide the speed to support large shapefiles (millions of points).

Install rgdal package (bindings for the Geospatial Data Abstraction Library)

```
>install.packages("rgdal",dependencies=TRUE)

library(rgdal)
```

Read shapefile for conus USA

```
>USA.conus<readOGR("shapefile/GIS_Data","US_Conus")
```

Projection information

```
proj4string(USA.conus)
```

Read other shapefile for conus USA

```
USA.conus<-
readOGR("shapefile/GIS_Data","US_Conus")
```

Read different spatial data layers

Point

```
>USA.cities<readOGR("shapefile/cities","cities")
```

Lines

```
>USA.Rivers<readOGR("shapefile/rivers",
"rivers")

>USA.hydroln<readOGR("shapefile/hydroln
","hydroln")
```

Polygons

```
>USA.huc250k<readOGR("shapefile/huc250k
","huc250k")
```

Transform projection

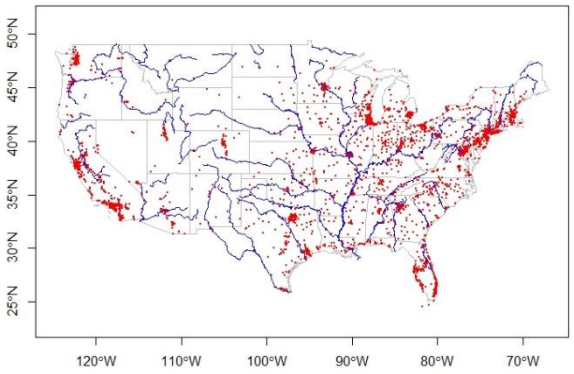
```
>USA.huc250k2<-spTransform(USA.huc250k,
CRS=CRS("+proj=longlat +ellps=GRS80"))
```

#overlay data for plot

```
>plot(USA.conus,axes=TRUE,border="gray"
)

>points(USA.cities, pch=20, col="red",
cex=0.5)

>lines(USA.Rivers, col="blue",
lwd=1.0),col="blue", lwd=2.0)
```



Write shaepfile

```
>writeOGR(USA.cities,
"shapefile/GIS_Data", "USA_cities",
driver="ESRI Shapefile")
```

3.2 Raster Data

In its simplest form, a raster consists of a matrix of cells (or pixels) organized into rows and columns (or a grid) where each cell contains a value representing information, such as temperature, elevation etc.

A spatial data model that defines space as an array of equally sized cells arranged in rows and columns, and composed of single or multiple bands.

Rasters are digital aerial photographs, imagery from satellites, digital pictures, or even scanned maps.

R supports reading writing geospatial and non-geospatial raster data.

The *raster* package has functions for creating, reading, manipulating, and writing raster data. The package provides, among other things, general raster data manipulation functions that can easily be used to develop more specific functions

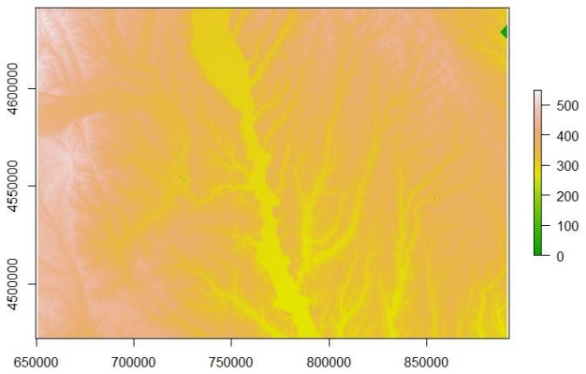
```
install.packages("raster",
dependencies=TRUE)

>library(raster)
```

Read elevation data from USGS

```
>MyDEM= raster("DEM.tif")

>plot(MyDEM,col=(terrain.colors(1000)))
```



Psychometric constant calculation from elevation

```
>psy <-(0.0673645*((293-0.0065*
MyDEM)/293)^5.26)
```

You can write an entire Raster* object to a file, using one of the many supported formats. When writing a file to disk, the file format is determined by the 'format=' argument if supplied, or else by the file extension (if the extension is known).

```
>outputraster <- writeRaster(psy,
filename="psychrometric.tif",
overwrite=TRUE)
```