Rbit007

7/05/2014

Elgin Perry

eperry@chesapeake.net

**Using Lists**

Before we delve into processing data by groups, we should take a look at an R-structure called the **list**.

A list is simply a tool that allows you to collect a set of r-objects together under one name. It is analogous to zipping together a collection of files in a zip-file. The components of a list are not required to have special relationships. For example, a matrix is a collection of vectors and these vectors must all have the special property of being the same length and same class. Components of a list might be a mixture of a single number, a vector of character data, a data frame, or even another list which could also be a conglomerate of objects. The R-functions that process data in groups use lists to define the groups and sometimes return results in lists. Frequently the results returned by r-functions, e.g. lm(), are a list of objects. In the examples that follow we illustrate these ideas.

First look at simple lists

To create a list, use the list() function –

contact1 <- list(

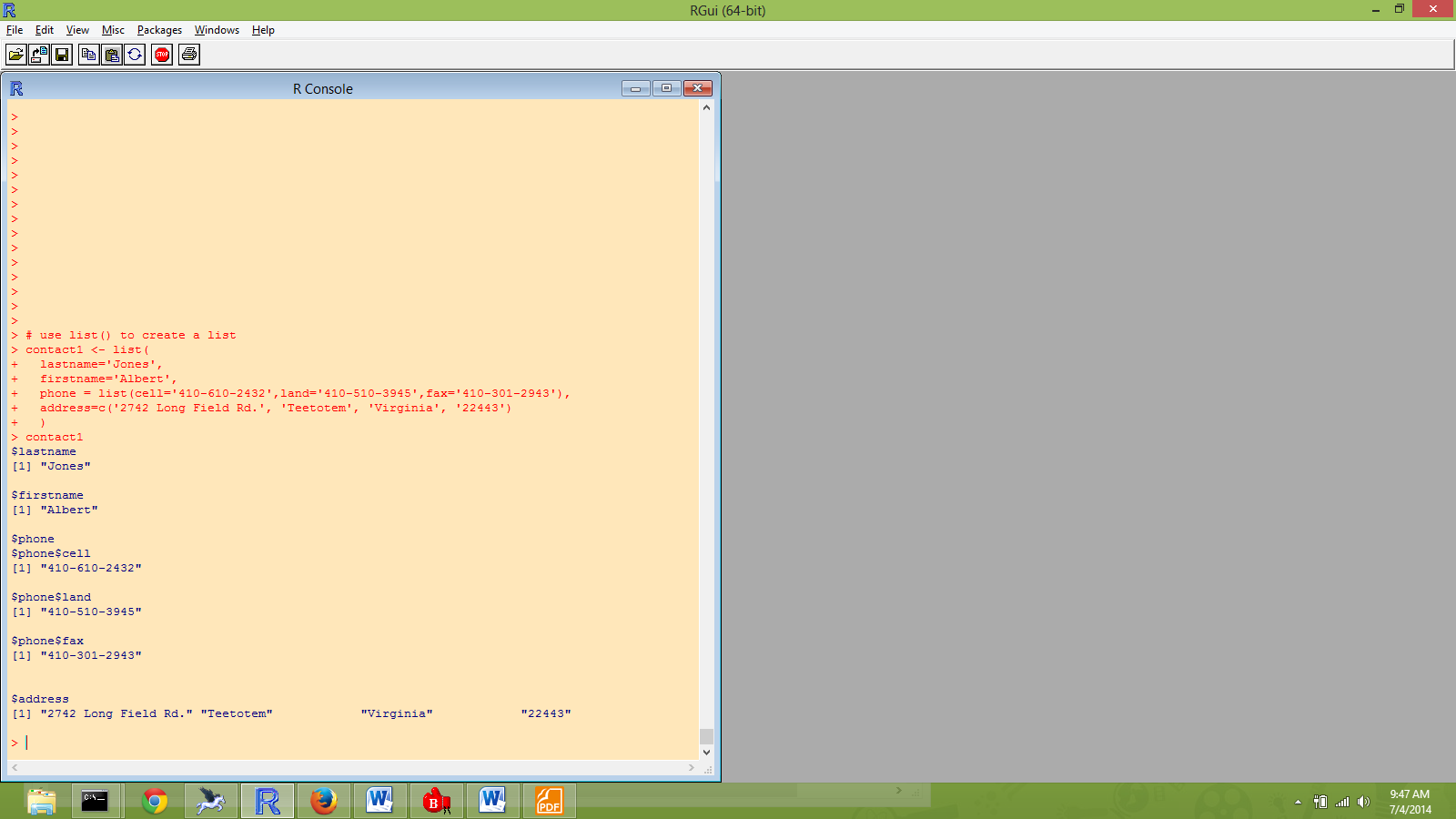
lastname='Jones',

firstname='Albert',

phone = list(cell='410-610-2432',land='410-510-3945',fax='410-301-2943'),

address=c('2742 Long Field Rd.', 'Teetotem', 'Virginia', '22443'))

contact1



This list has four components delimited by commas. In the above example each component is on one line. Structuring the command for the list function with one component per line is done for clarity; it is not a syntax requirement. Each component has a name, which is followed by ‘=’ , which is followed by the contents of the component. For the first component the name ‘lastname’ is assigned the contents ‘Jones’. The second component, firstname is assigned the contents ‘Albert’. The third component, phone, is assigned a list of three phone numbers. The last component ‘address’ is assigned a character vector of length 4 with address information.

Components of the list can be referenced by name or number. If using numbers, use a double bracket [[ ]] notation. If using names use a ‘$’ notation.

# methods of referencing list components

contact1[[1]]

contact1$lastname

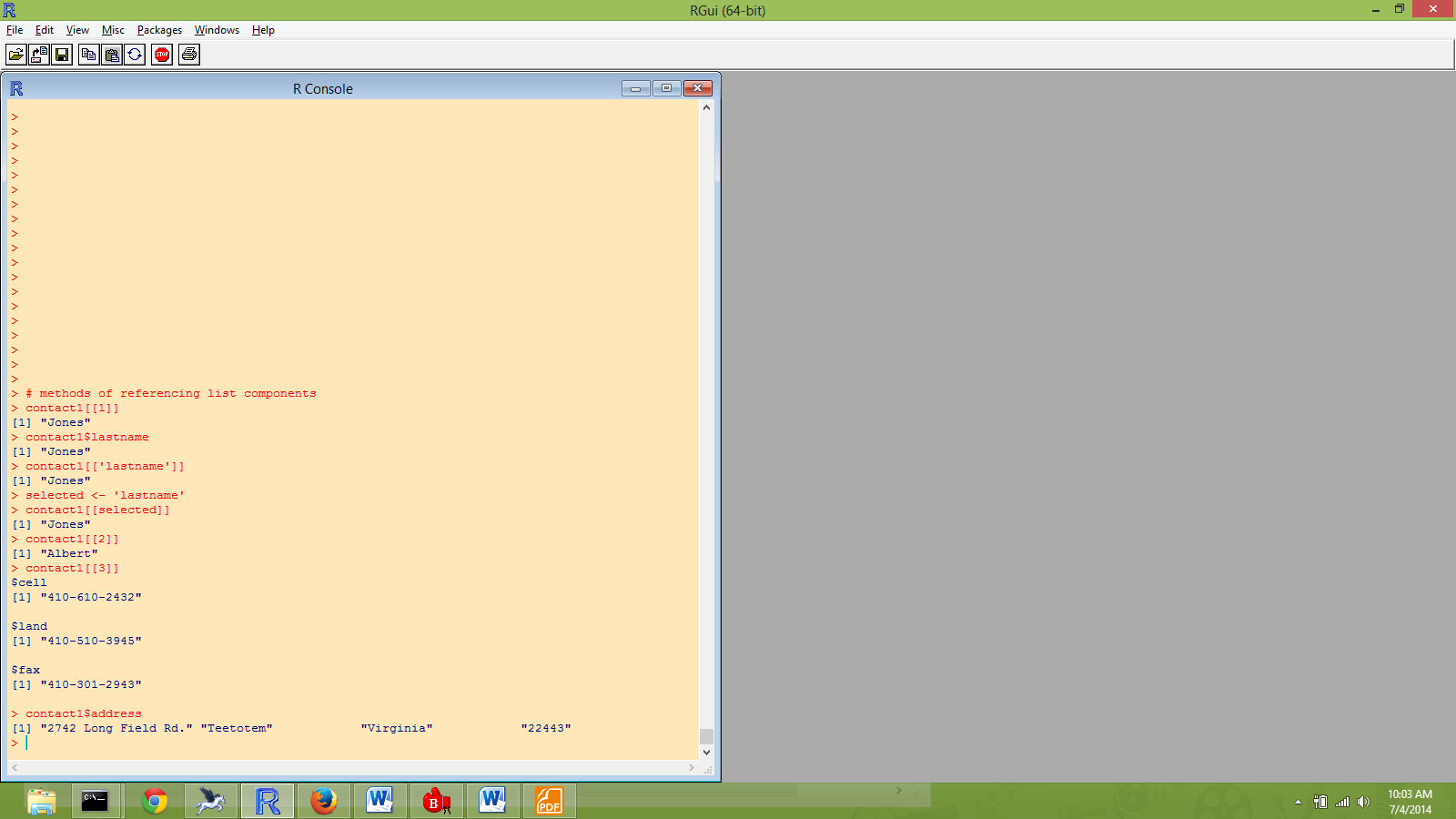
contact1[['lastname']]

selected <- 'lastname'

contact1[[selected]]

contact1[[2]]

contact1[[3]]

contact1$address

If a component of a list has elements, you just continue adding the appropriate subscripting notation. Component 3 is a list so to access single elements of component 3 you may use

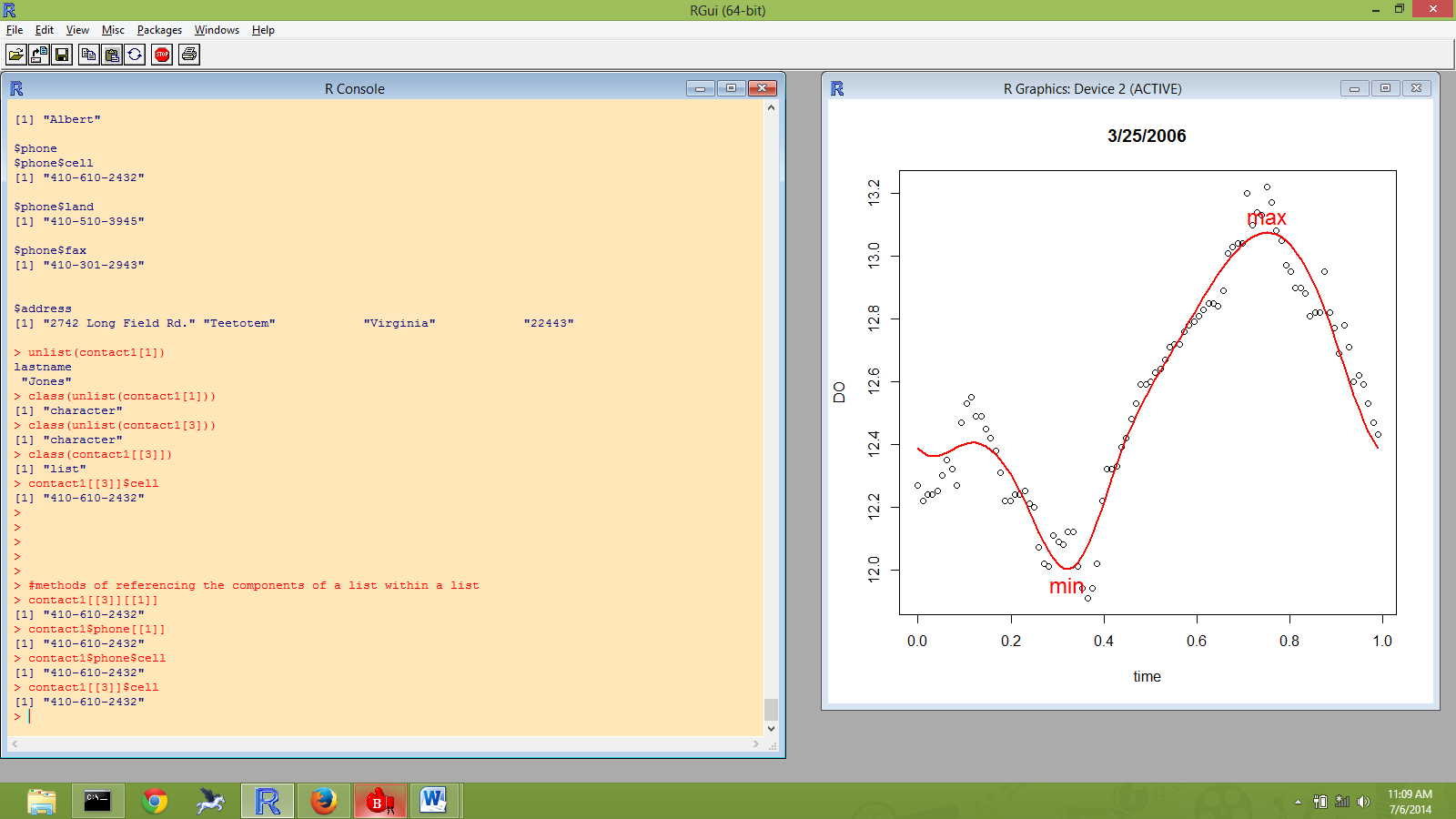
#methods of referencing the components of a list within a list

contact1[[3]][[1]]

contact1$phone[[1]]

contact1$phone$cell

contact1[[3]]$cell



For the fourth component of the list which is a vector, use

# methods for referencing the elements of vectors in a list

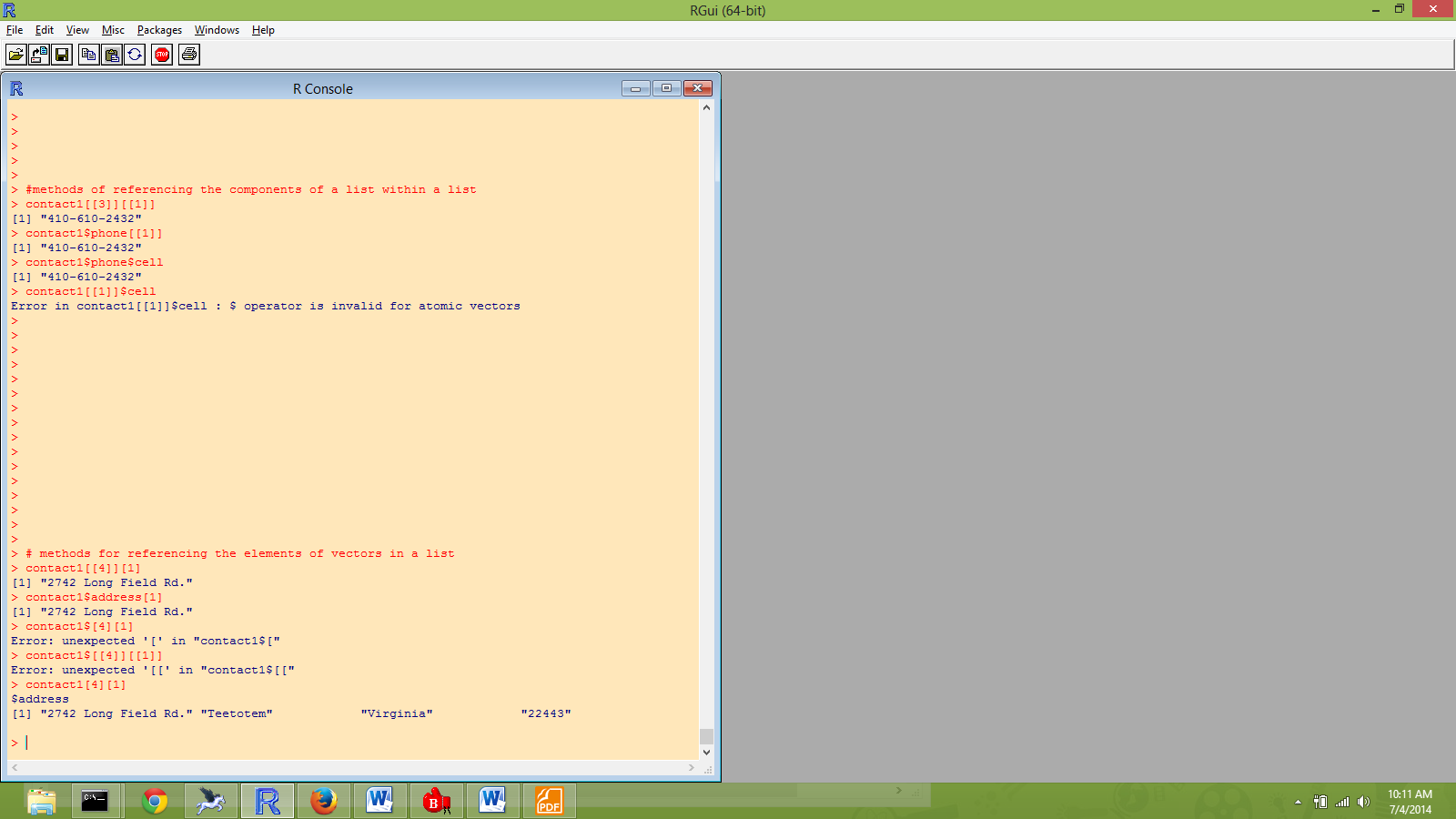
contact1[[4]][1]

contact1$address[1]

contact1$[4][1]

contact1$[[4]][[1]]

contact1[4][1]



I have intentionally included a couple of syntax errors here so you can see the result. Notice the last line

contact1[4][1]

which does not produce an error, but not necessarily the expected result either. The single bracket notation [ ] is used for creating sub-lists from lists. The double bracket notation [[ ]] is used for extracting a single component from a list. The difference is that a sub-list will still be of class ‘list’ while an extracted component will be of the class used to create the list component. To illustrate, contact1[1] has class ‘list’ while contact1[[1]] has class ‘character’.

# [] vs. [[ ]] for lists

a <- contact1[1]; class(a)

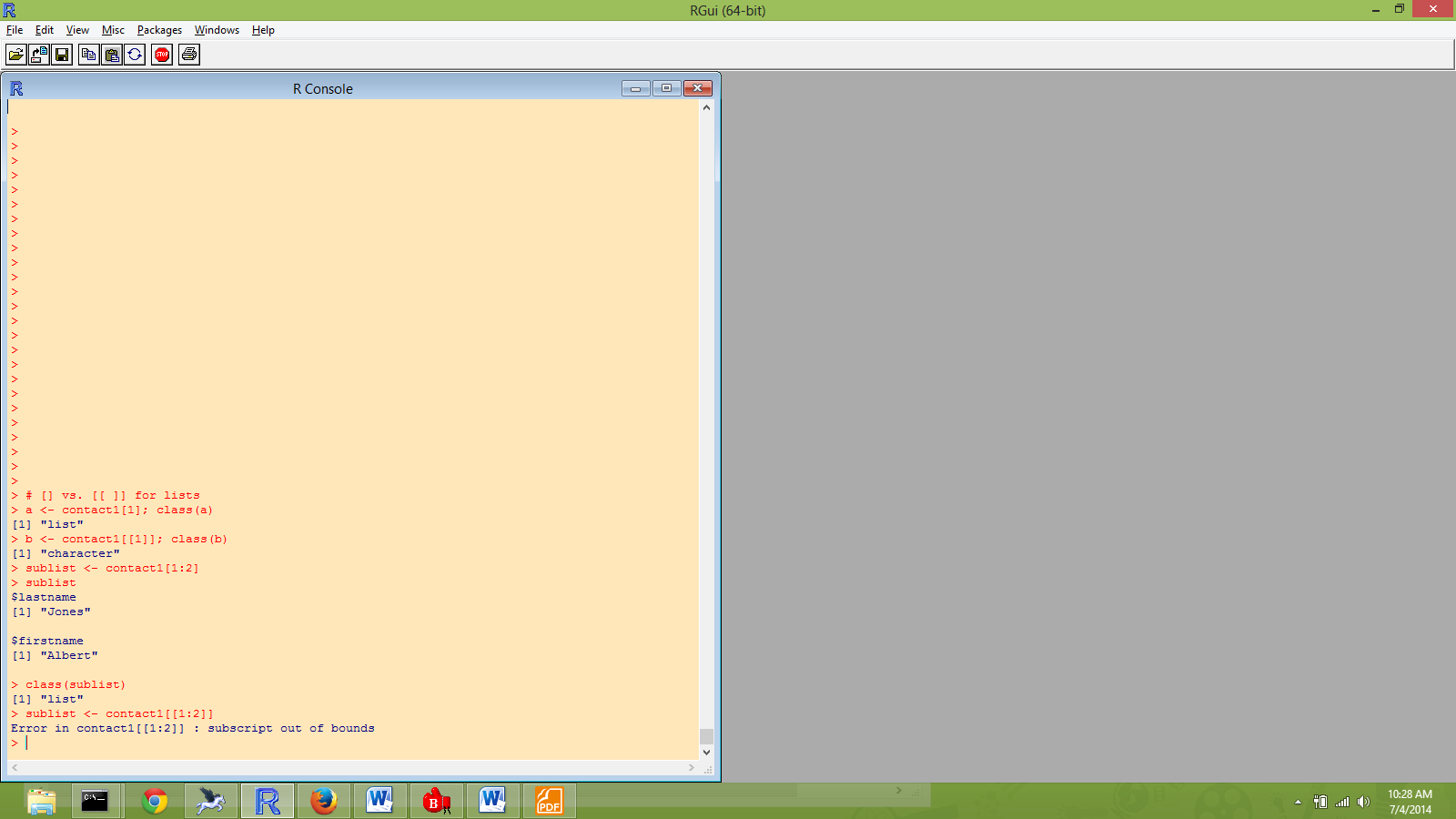
b <- contact1[[1]]; class(b)

sublist <- contact1[1:2]

sublist

class(sublist)

sublist <- contact1[[1:2]]



Having the first contact end with a ‘1’ implies there will be a ‘2’.

contact2 <- list(

lastname='Smith',

firstname='Robert',

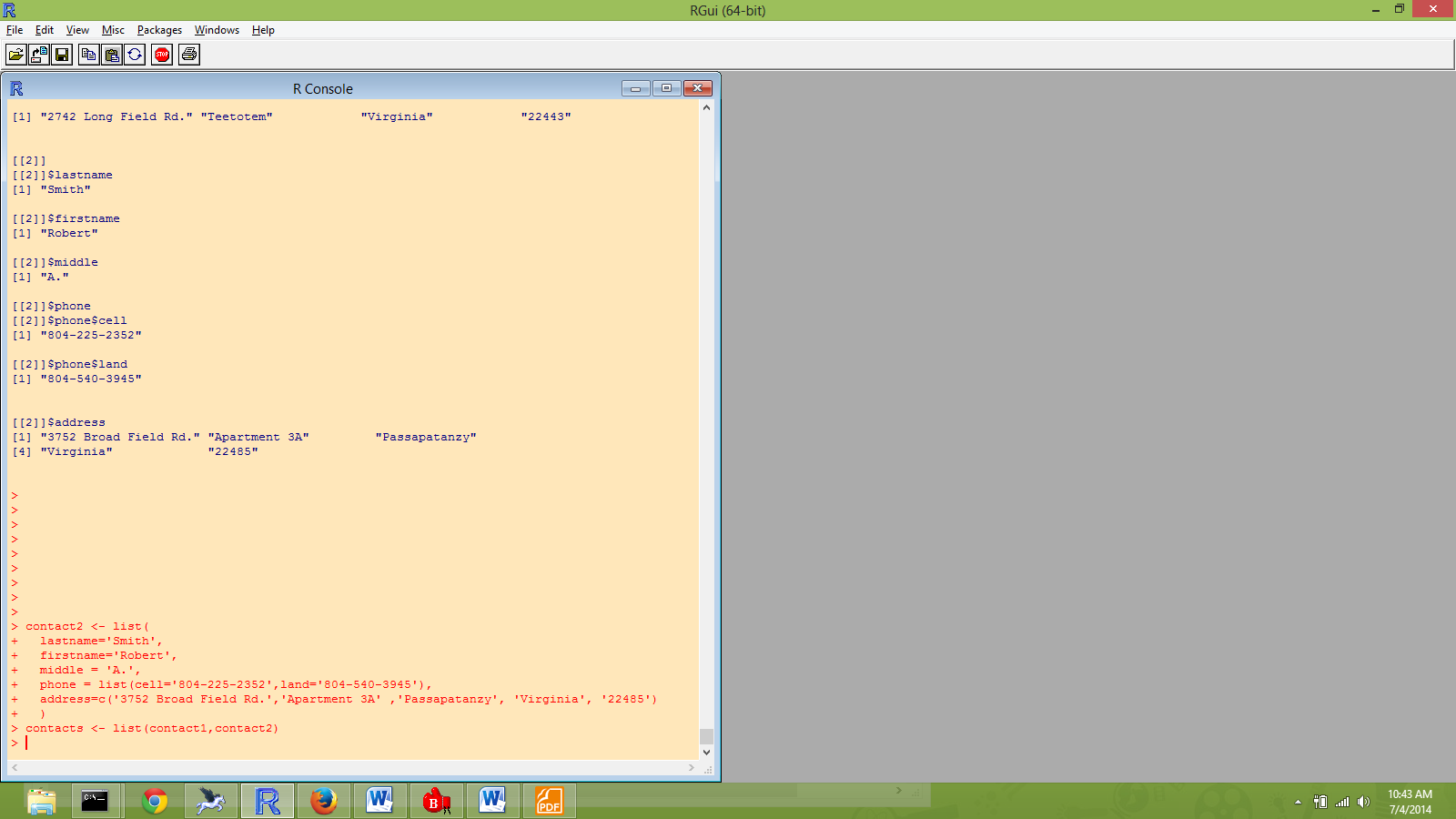
middle = 'A.',

phone = list(cell='804-225-2352',land='804-540-3945'),

address=c('3752 Broad Field Rd.','Apartment 3A' ,'Passapatanzy', 'Virginia', '22485')

)

contacts <- list(contact1,contact2)



Note that the second contact has a different structure than the first. Contact2 has a component called ‘middle’, the phone list only has two components, and the address vector has 5 elements. Even with these differences, there is no problem with putting contact1 and contact2 together in as list call contacts. Having these differences in not a recommended practice, but it illustrates the flexibility of lists.

**Processing Data by Groups**

We will look at two general methods of processing data in groups. One method uses base-r functions that provide a functionality similar to the ‘by’ statement in SAS. A second method uses loops and we got a peak at the last time (Rbit006), where we developed a user defined function to do an analysis for one day of the ConMon data and then looped through the days. I had hoped to cover a third method that uses functions of an add-on package call ‘doBy’, but I will leave that for self-study.

Most of the base-R functions that allow processing of data by groups have ‘apply’ as part of their name. We will look at the ‘apply’ functions: apply(), tapply, and sapply() and another group processing function aggregate(). Other ‘apply’ functions are lapply(), vapply, and mapply() that you might review in help. I find that these group processing functions are easiest to use when the function that you want to apply to each group produces a single number summary of a vector of data. For example mean(x) returns just the mean of x. I think is is possible to use these functions to apply complex functions such as lm(), but I have had little luck with that. For repeatedly applying complex functions, I use loops.

To illustrate applying functions to groups, we load the 5-day data from Mattawoman Creek ConMon (see ProcessingGroupsRbit007.r) using the usual tools.

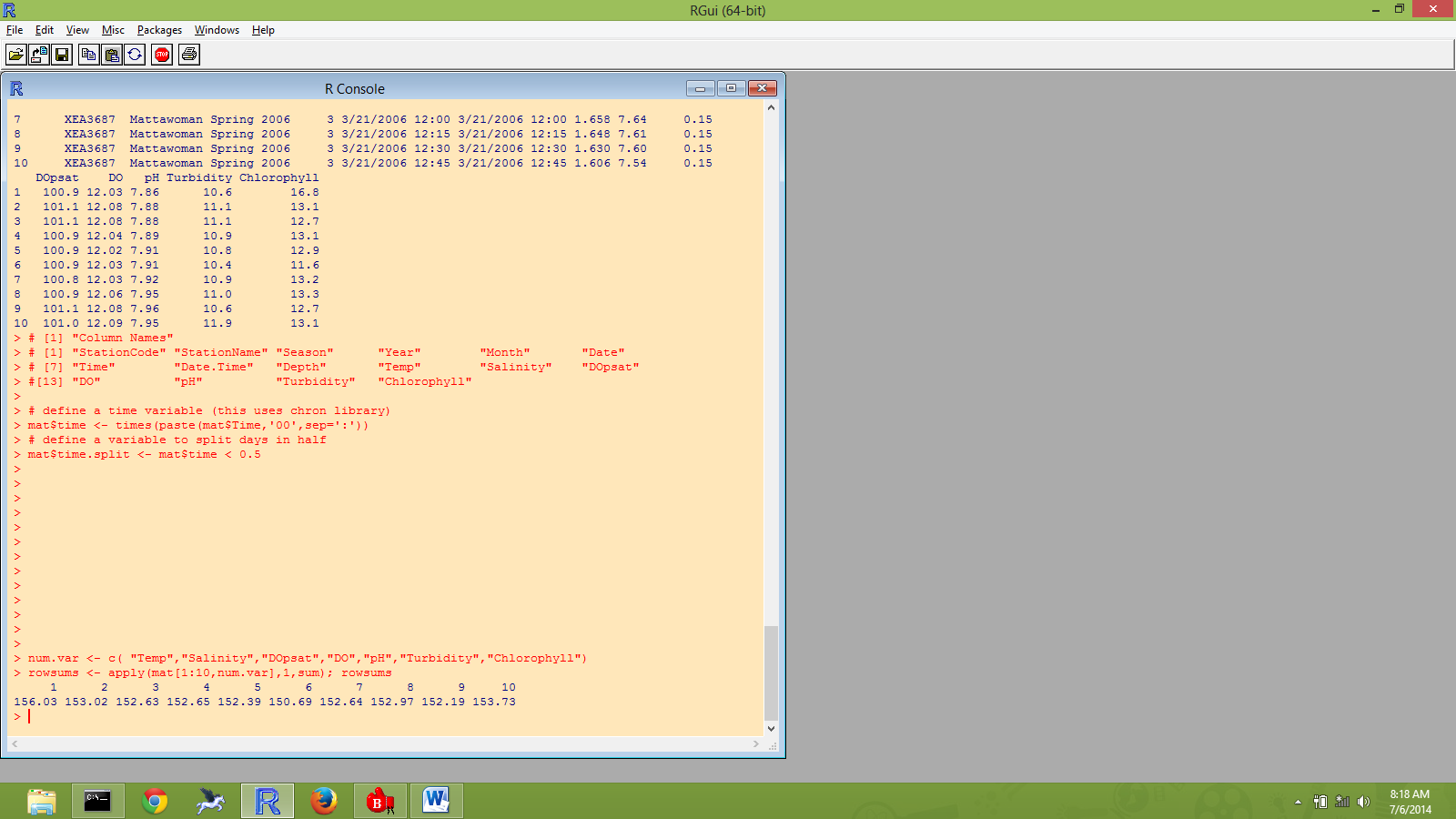
.

First illustrate the use of apply(). This function will apply a function specified as an argument to either the rows or the columns of a matrix. In the example below, I first create num.var, a character vector that allows me to easily selecdt the numeric data columns in the data frame mat. Then I call apply, where the first argument is a data frame (equivalent to a matrix) of the first 10 rows of mat and the columns specified by num.var. The second argument ‘1’ is an indicator: 1 = rows, 2 = columns. The third argument, ‘sum’, is the function to be applied. The result is a vector of length 10 containing the row sums of the numeric data.

# row and column operations with apply()

num.var <- c( "Temp","Salinity","DOpsat","DO","pH","Turbidity","Chlorophyll")

rowsums <- apply(mat[1:10,num.var],1,sum); rowsums

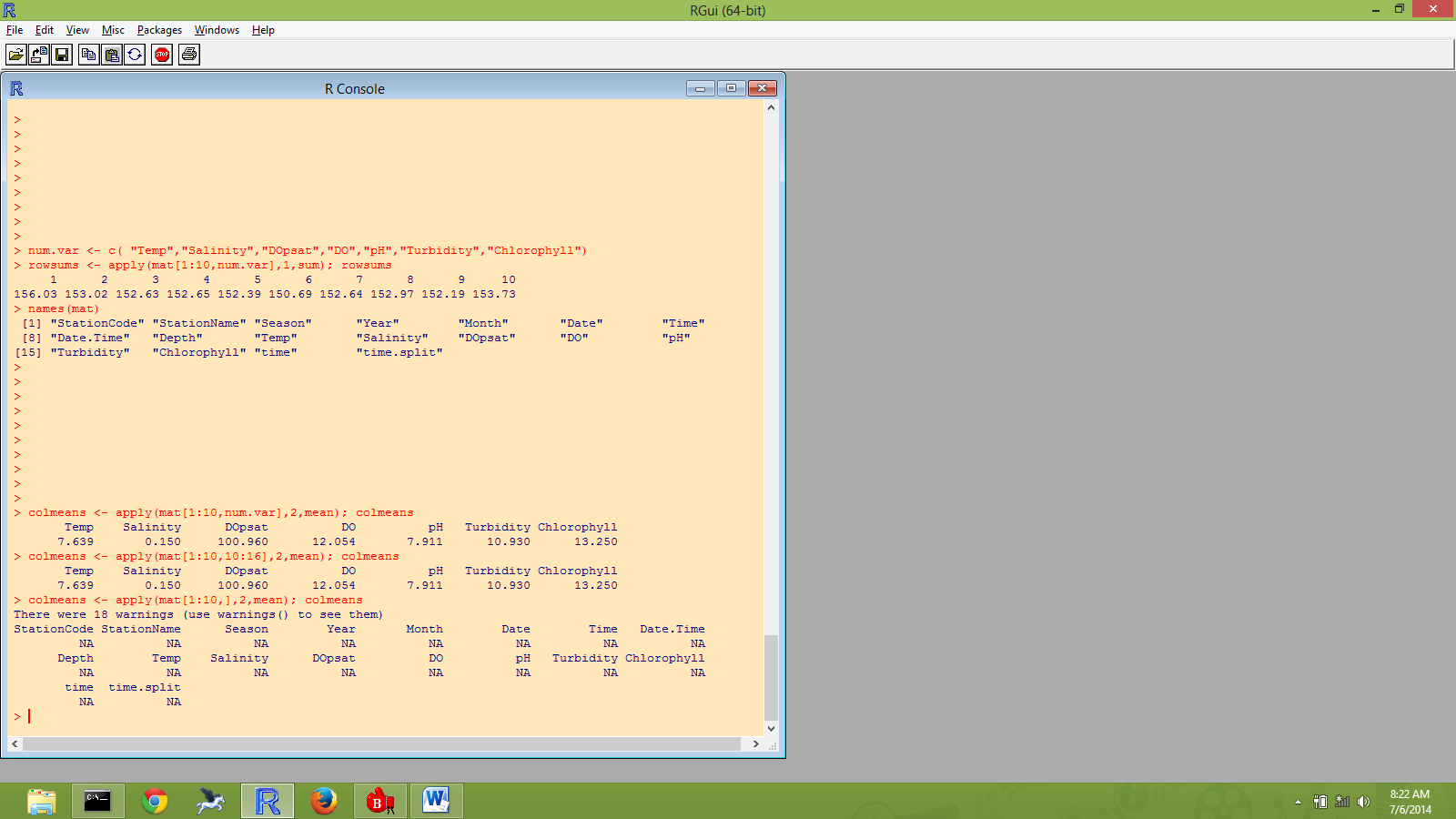


The next example shows using apply() for column means of the numeric columns. I computed these a second time using the 10:16 indices for the numeric columns just to show a different way to do this. Finally I show what happens in apply() if the function being applied finds data that it can’t handle.

colmeans <- apply(mat[1:10,num.var],2,mean); colmeans

colmeans <- apply(mat[1:10,10:16],2,mean); colmeans

colmeans <- apply(mat[1:10,],2,mean); colmeans



Next we look at tapply(). From help we get the basic call

**Usage**

tapply(X, INDEX, FUN = NULL, ..., simplify = TRUE)

The first argument X is typically a vector. Often it is a column of a data frame. The second argument INDEX is a vector of the same length as X which contains grouping information. Often it is a factor type vector from the same data frame as X. FUN is the function to be applied and simplify determines if the results are to be returned as a ‘list’ or simplified to a vector. Here are examples using the Mattawoman ConMon data.

# grouped data operations with tapply

mn.temp <- tapply(mat$Temp,mat$Date,mean); mntemp

sd.temp <- tapply(mat$Temp,INDEX=mat$Date,FUN=sd,na.rm=TRUE); sd.temp

# put results in a matrix

mnsd.temp <- cbind(mn.temp,sd.temp)

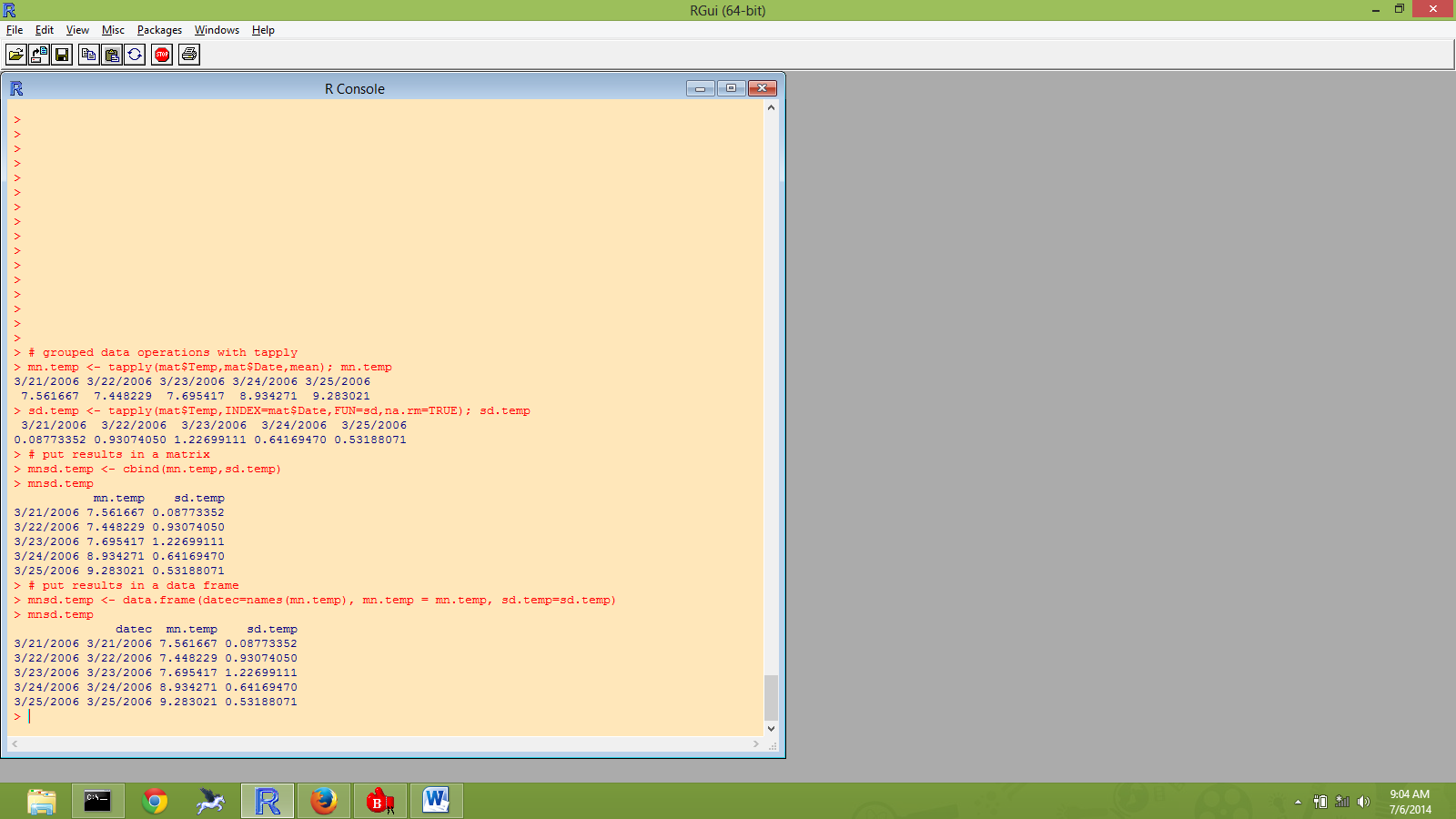
mnsd.temp

# put results in a data frame

mnsd.temp <- data.frame(datec=names(mn.temp), mn.temp = mn.temp, sd.temp=sd.temp)

mnsd.temp

In the first example, mn.temp is created by calling tapply () for the ‘Temp’ column of the mat data frame, the Date column defines the groups, and the ‘mean’ function is to be applied to each group. The result returned is a vector of means for each day. In the second example, I create a similarly structured vector of standard deviations. In this call I was specific about specifying INDEX=mat$Date rather than relying on position (1st, 2nd, 3rd . . .) to define the arguments. In addition, if the function you are applying, sd() in this case, needs additional arguments (e.g. na.rm=TRUE), then they are added after the function argument as arguments of tapply(). Here, na.rm=TRUE, has no effect because there are no missing data. It is used for illustration. After creating mn.temp and sd.temp, I cbind them together into a matrix and also show putting them in a data frame.

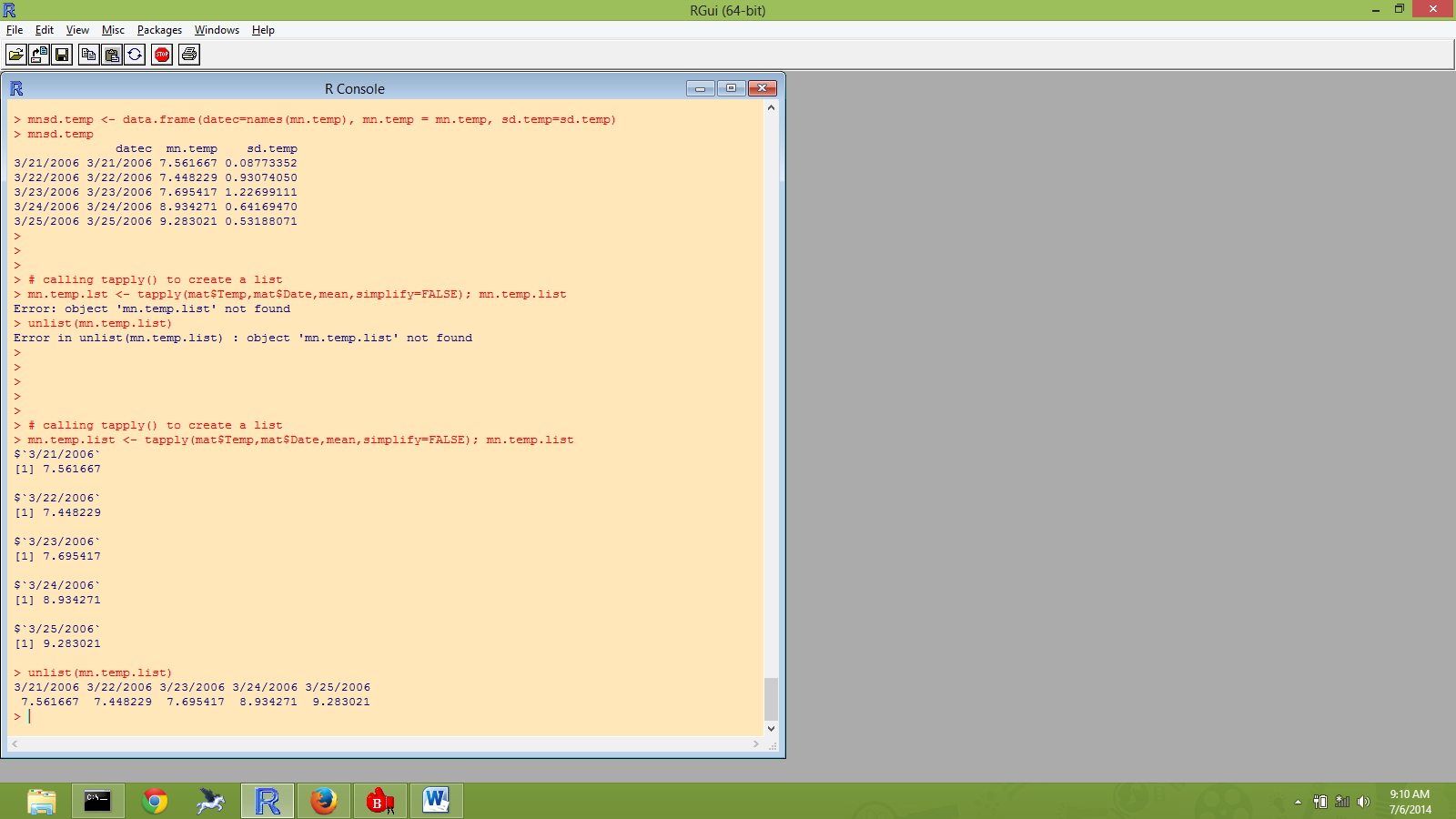


This last example calls tapply() with the ‘simplify=FALSE’ argument. In this case, tapply() returns a list rather than a vector. It can be converted to a vector using the unlist() function. I am uncertain about when it is more useful to have a list returned.

# calling tapply() to create a list

mn.temp.list <- tapply(mat$Temp,mat$Date,mean,simplify=FALSE); mn.temp.list

unlist(mn.temp.list)

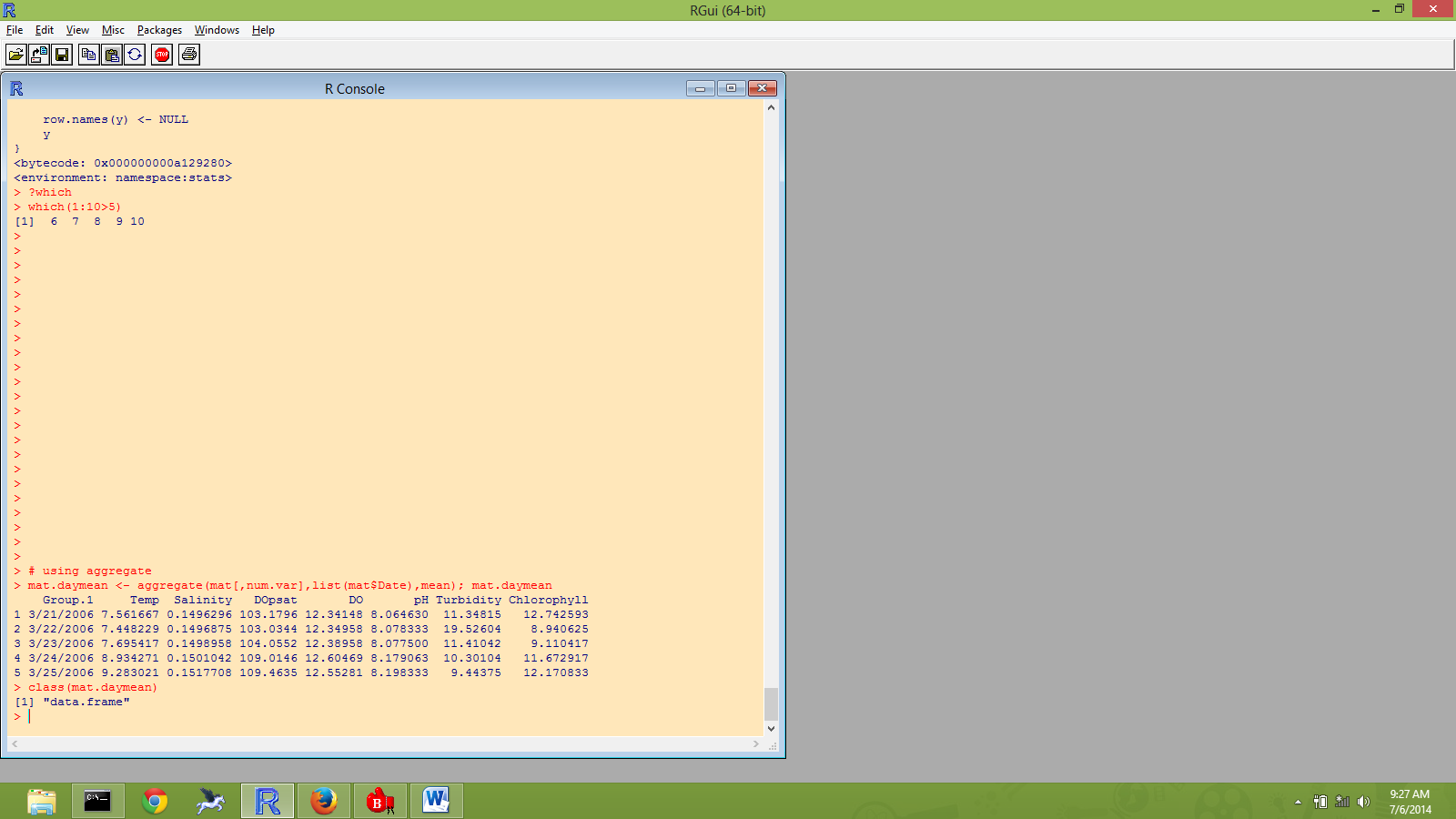


Now we look at aggregate() which will produce summary statistics for multiple columns of a data frame. The call to aggregate() is similar to the call for tapply(). The first argument is the data to which the function is applied, but it should be a matrix or a data frame. The second argument is for indexing, but rather than a simple vector, it is a list of one or more vectors. The third argument is the function to be applied just as with tapply(). In the example below, we supply the numeric columns of mat as referenced by ‘num.var’ as the data arguments, mat$Date as the indexing argument, and mean as the function to apply. The result is a data frame with one row for each unique component of the indexing list. The in this data frame columns correspond to the columns of the data argument plus an additional column(s) for the indexing argument.

# using aggregate

mat.daymean <- aggregate(mat[,num.var],list(mat$Date),mean); mat.daymean

class(mat.daymean)



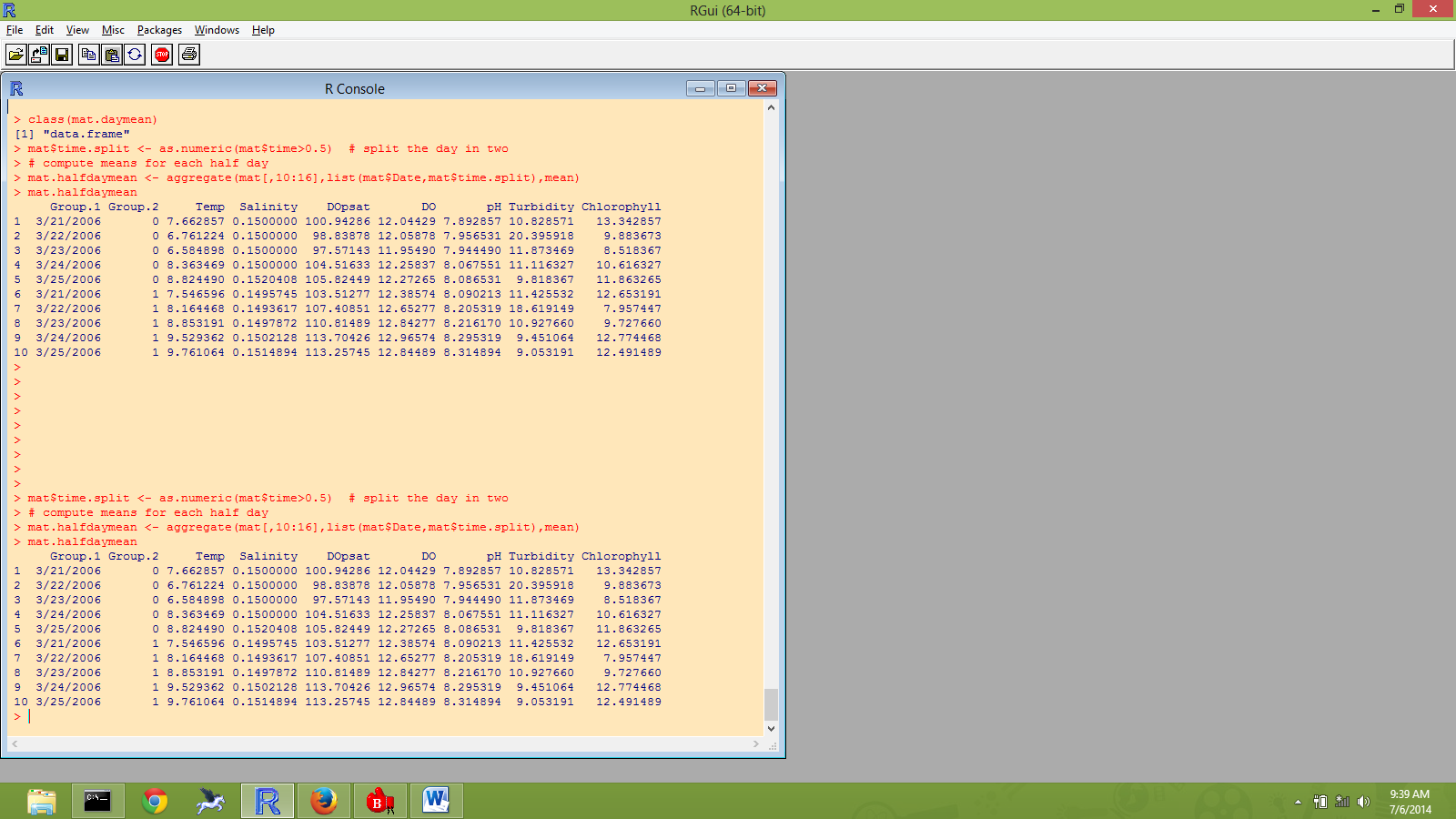
The function aggregate() can be called with two indexing columns.

mat$time.split <- as.numeric(mat$time>0.5) # split the day in two

# compute means for each half day

mat.halfdaymean <- aggregate(mat[,10:16],list(mat$Date,mat$time.split),mean)

mat.halfdaymean



You can supply a user defined function for aggregate() or tapply(). Here I define iqr(), a function to compute the inter-quartile range and use this with tapply() and aggregate().

# write you own function for tapply or aggregate

iqr <- function(x)

{

# x <- mat$Temp

q25 <- quantile(x,0.25)

q75 <- quantile(x,0.75,)

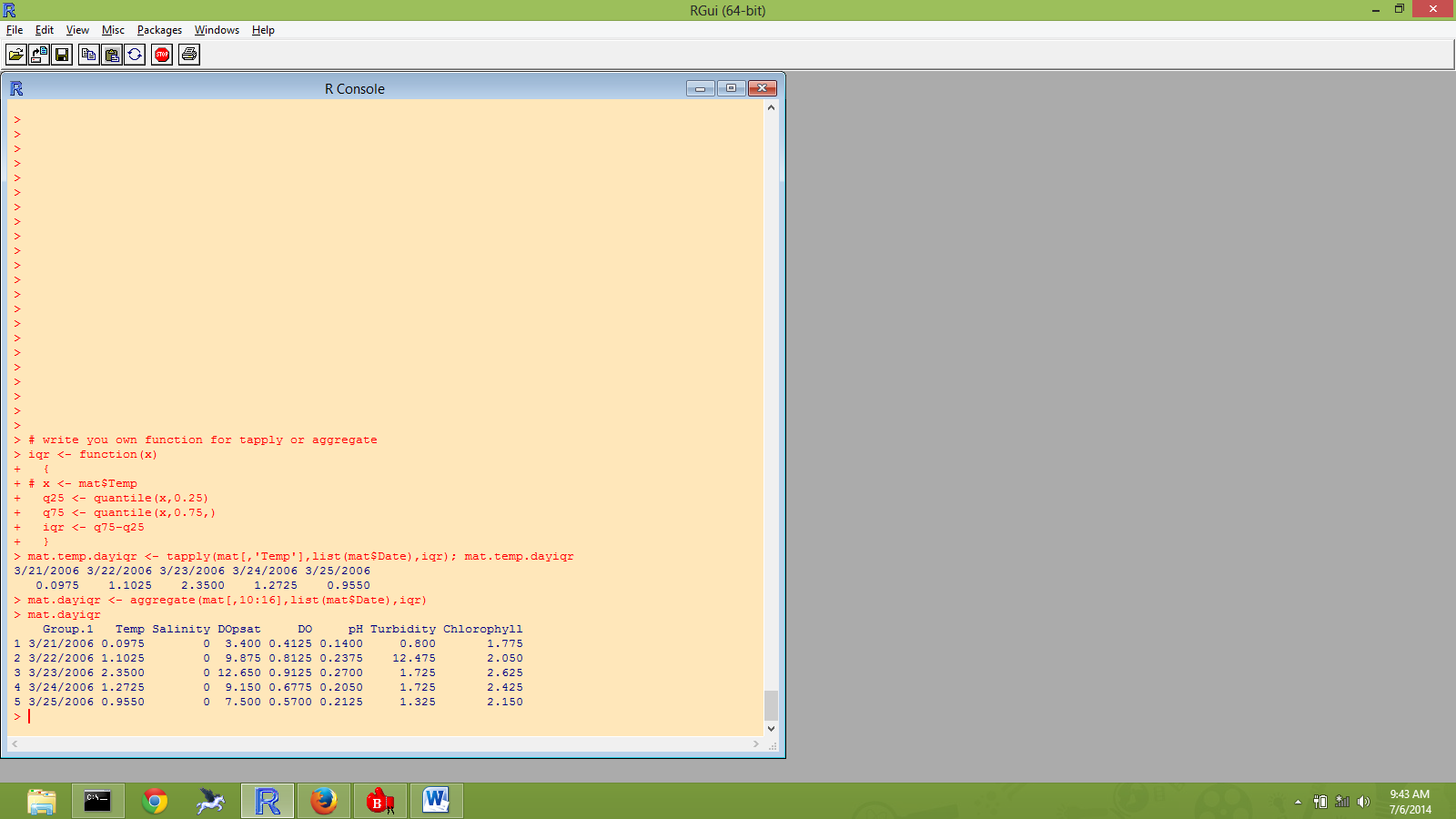
iqr <- q75-q25

}

mat.temp.dayiqr <- tapply(mat[,'Temp'],list(mat$Date),iqr); mat.temp.dayiqr

mat.dayiqr <- aggregate(mat[,10:16],list(mat$Date),iqr)

mat.dayiqr



I believe it is possible to do complex function analysis by groups using lapply() and sapply(), but I have not been very successful at this. For complex analyses, I resort to using loops. The R-language provides several looping structures such as for-loops and while-loops. I typically use a for-loop. In Rbit006, we looked at using a loop to create day by day plots of the diel DO cycle in the Mattawoman ConMon data. These plots were saved to a \*.pdf file. In this example, the diel.gam() function of Rbit006 is modified to produce some other tabulated results. The tables and the plot are saved in an \*.rtf file. Saving results in \*.rtf format is accomplished using functions from the file RTF.r which are user defined functions that I have written. You are welcome to use these, but I should warn you that because I have written these for my use, I have not made them user friendly. Look at the comments in the new diel.gam() function below to get an understanding of what this function is doing. Open the file ‘ProcessingGroupsRbit007.rtf’ to see the results.

diel.gam <- function(day)

{

# day <- '3/21/2006'

# select data for specified date into temporary data frame tdta

tdta <- mat[day==mat$Date,]

# put at title for this set of output in rtf file

RTFtext(paste("Diel Analysis results for",day))

# create data summary

num.var <- c( "Temp","Salinity","DOpsat","DO","pH","Turbidity","Chlorophyll")

mat.halfdaymean <- aggregate(tdta[,num.var],list(tdta$Date,tdta$time.split),mean)

mat.halfdaymean

# call RTFtab() with defaults

RTFtab(mat.halfdaymean)

# make table a little nicer

mat.halfdaymean[,num.var] <- round(mat.halfdaymean[,num.var],2)

RTFtab(mat.halfdaymean,

TableTitle=paste('Means of numeric data by half-day for',day),

vr= c("Group.1","Group.2","Temp","Salinity","DO","pH","Turbidity","Chlorophyll"),

ch= c("Date","am/pm","Temp","Salinity","DO","pH","Turbidity","Chlorophyll"),

cw = c(rep(1000,6),1200,1500),

)

# fit gam model to selected data

dogam <- gam(DO ~ s(time,bs='cc'),data=tdta)

# get predicted values from gam and add to tdta

tdta$pred <- predict(dogam)

# plot data, label with day

plot(DO~time,data=tdta,main=day)

# overlay predicted line

lines(pred~time,data=tdta,col='red',lwd=2)

# get max and min predictions

range.do <- range(tdta$pred)

# locate times associated with max and min

min.pt <- tdta[range.do[1]==tdta$pred,c('time','pred')]

max.pt <- tdta[range.do[2]==tdta$pred,c('time','pred')]

min.time <- tdta[range.do[1]==tdta$pred,'Time']

max.time <- tdta[range.do[2]==tdta$pred,'Time']

# label max and min on plot

text(min.pt[1],min.pt[2],'min',cex=1.5,col='red',pos=1)

text(max.pt[1],max.pt[2],'max',cex=1.5,col='red',pos=3)

# put plot in RTF file

RTFput.plt(tmpfile='c:/projects/rtp/TempPng.png')

# write min/max summary in rtf file

RTFtext(paste("The maximum DO of", round(max.pt[2],2),"occurred at",max.time))

RTFtext(paste("The minimum DO of", round(min.pt[2],2),"occurred at",min.time))

# put a blank line is rtf file

RTFtext('')

# write a gam summary anova table in rtf file

RTFgam.anova(dogam)

# put the standard r-summary of a gam in the rtf file

RTFsummary(dogam)

# put a page break in the rtf file

RTFpage()

} #end diel.gam

# call function for each date in mat

daylist <- unique(mat$Date)

# initialize an RTF file

RTFout <- "c:/Projects/CBP/Rcourse/ProcessingGroupsRbit007.rtf"

RTFinit()

# call function for each date in mat using loop and save results to rtf

for (day in daylist) { diel.gam(day) }

RTFtext('end of file')

RTFclose()

To run this part of the code from ProcessingGroupsRbit007.r in your own environment, you will need to modify the path name in the definition of RTFout in the line

RTFout <- "c:/Projects/CBP/Rcourse/ProcessingGroupsRbit007.rtf"

and also modify the pathname in the line

RTFput.plt(tmpfile='c:/projects/rtp/TempPng.png')

. Be careful about the direction of the ‘/’.