Rbit006

6/22/2014

Elgin Perry

eperry@chesapeake.net

**User Defined Functions**

In this Rbit, we look at user defined functions. A large part of the power of R is made available through functions. Functions allow you to leverage your time by creating tools you can use over and over. Functions will add structure to your code making it easier to follow. This is very helpful as you get older and especially when you have to go back and figure out what you did when you were younger. Functions will save you from Carpal Tunnel Syndrome by doing boring repetitive tasks automatically rather having to point and click your way through them.

In mathematics, a function is a rule that assigns members of one set to members of another set. In programming it is better to think of a function as just a collection of commands that you assign to a name so you can call them all at once. These commands produce a specific result. It might be a mathematic calculation; it might be plot; or it might be a table of results. You might write functions to each of these three things and then write a wrapper function that calls these three functions to do the analysis, make the plot, and tabulate the results all with one function call. Further, you might put this wrapper function is a loop that asks R to do this for every station in the Chesapeake Bay. At this point I would go to play my guitar. Since you work for CBP, you would get to go to a meeting.

You can feed data in the form of function arguments to a function that it can use in performing it’s task. These arguments might be individual numbers it needs to perform a calculation, data frames for data analysis, character strings that control options, or any of these that one function might need to pass to another function. The R-programming language makes it particularly easy to set defaults for function arguments so that you do not have to explicitly set each argument for each function call.

We begin by writing a function that does a mathematical calculation because this is quite similar in concept to a mathematical function. Subsequently we will look at some user defined functions that I have written and use frequently.

The syntax for a function definition is shown here where the function name is ‘myfunct’, optional arguments are enclosed in (), and the function statements are enclosed in {}.

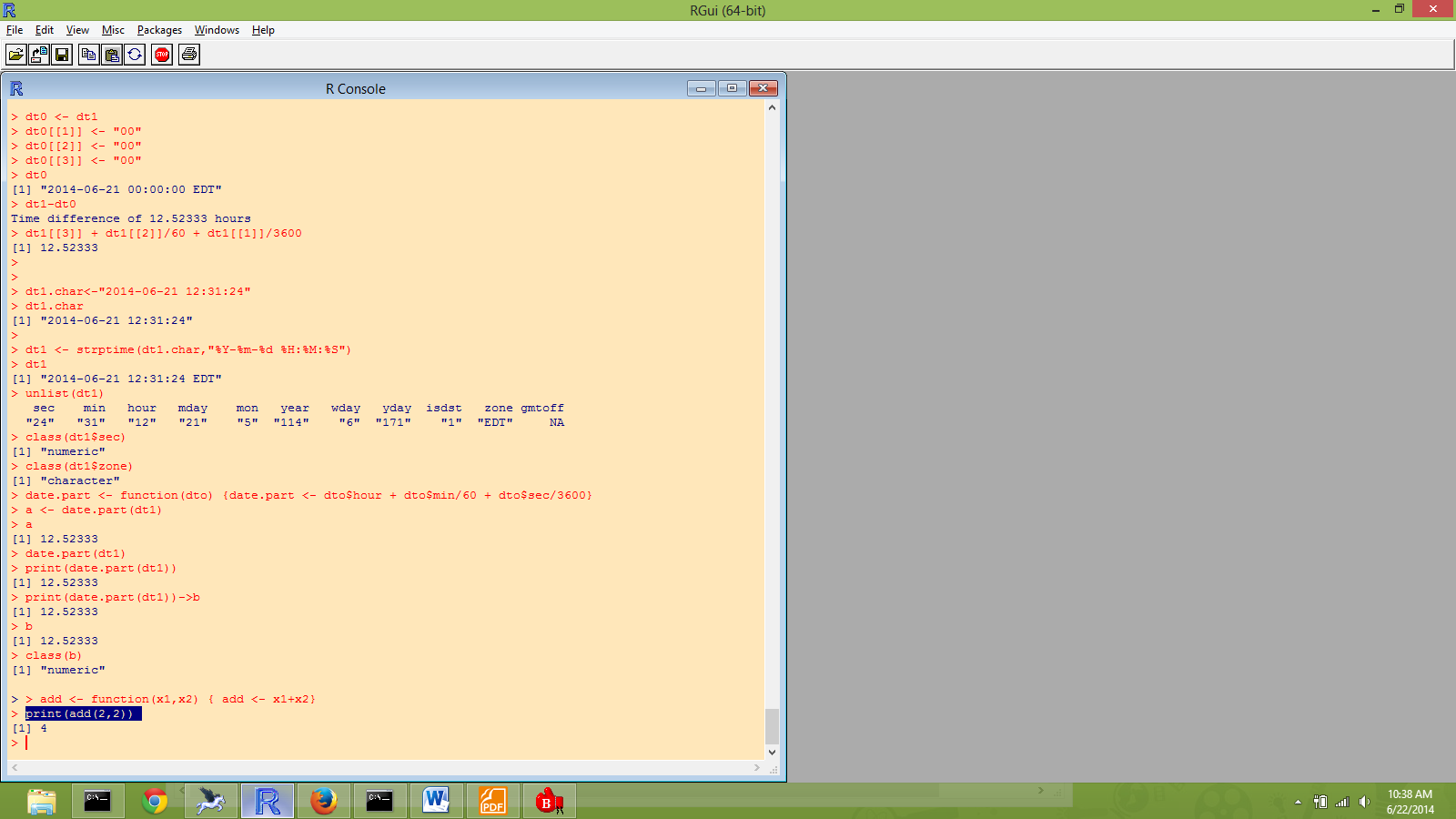
Myfunct <- function(arg1, arg2, . . . ) {statement\_1, statement\_2, . . . }

Note that the () for arguments are needed even if the function does not use arguments.

Adding two numbers could be defined as a function

add <- function(x1,x2) { add <- x1+x2}

print(add(2,2))



The arguments 2 and 2 define what goes into the function. The last assignment statement inside the {} defines what comes back from the function. It is customary to may this last assignment to the function name, but this is not necessary. If you want to return several objects, you will need to collect them in a vector, dataframe or a list object. When you execute an R-defined function, it will usually display the results. When you execute a user-defined function, R does not display the result. Thus you need to assign the result to an object and the display that object, or use print with the function call.

Now let’s write a function to compute Euclidean Distance between two points, (x1,y1) and (x2,y2). Before we begin, I will share a technique for writing functions that works for me. Error messages that originate from inside a function are hard to decipher in the R-language and every other language that I have used. Therefore I always test the statements I want to use in my function in ‘open code’. ‘Open code’ is a term I use for statements that are not inside of a function or a loop. In the editor, I set up the structure syntax for the function as defined above and then the first line after the “{“ is a comment line that if you skip the “#” will assign all of the arguments. Then I put in all of the executable statements. For this function is will look like this:

Euc.dist <- function(x1,y1,x2,y2)

{

# x1<-1; y1<-1; x2<-5; y2<-4

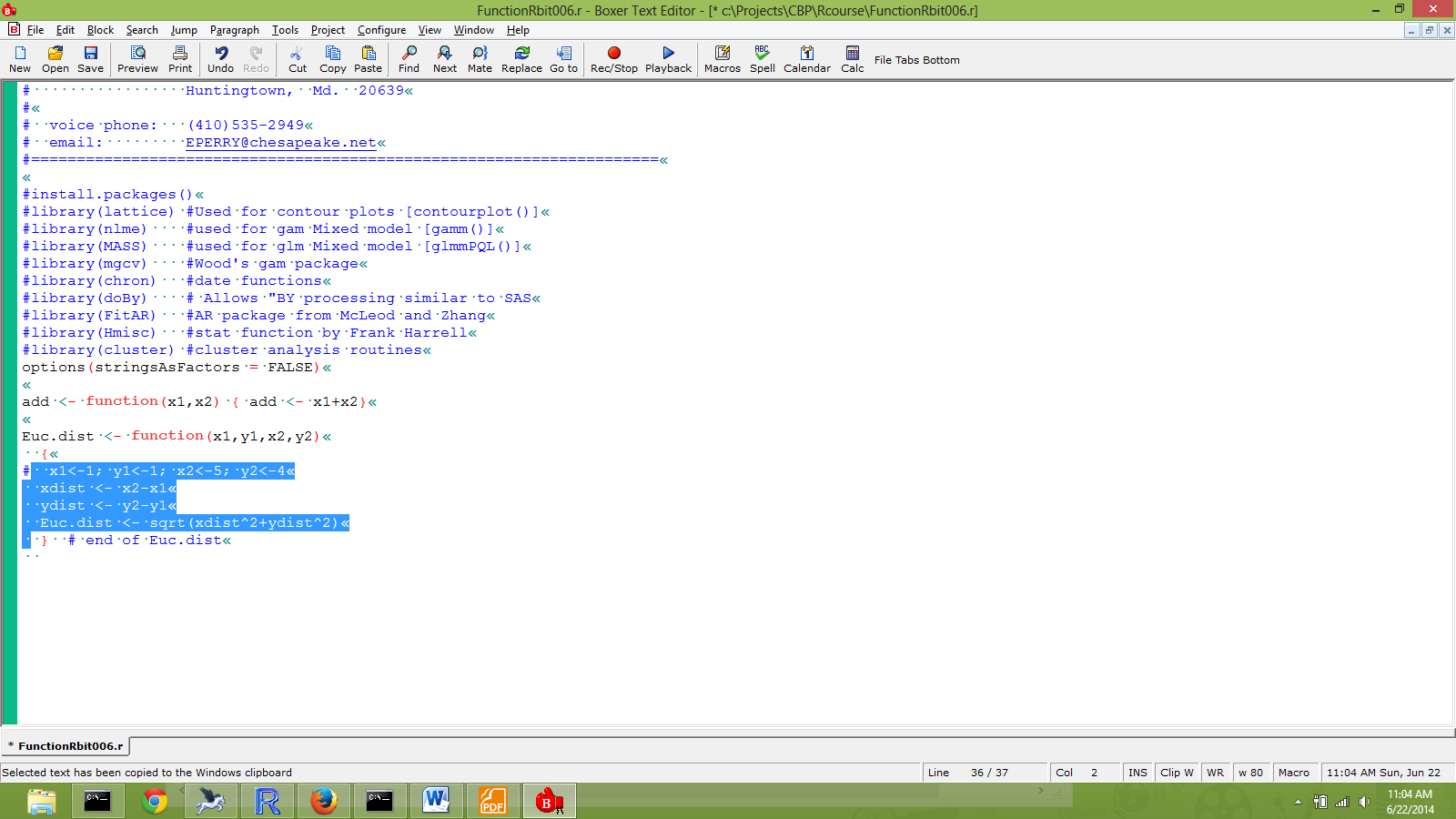
xdist <- x2-x1

ydist <- y2-y1

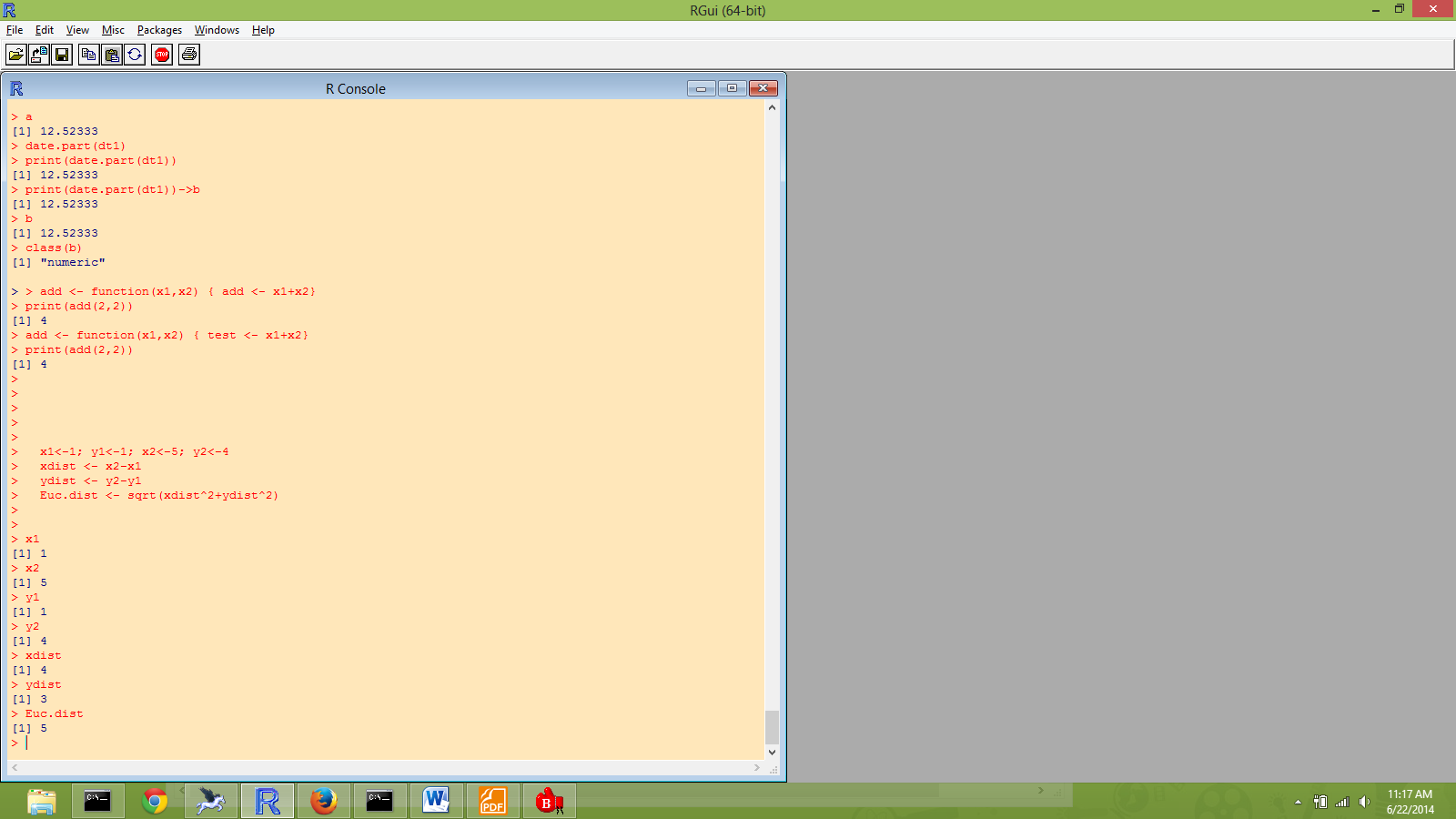
Euc.dist <- sqrt(xdist^2+ydist^2)

} # end of Euc.dist

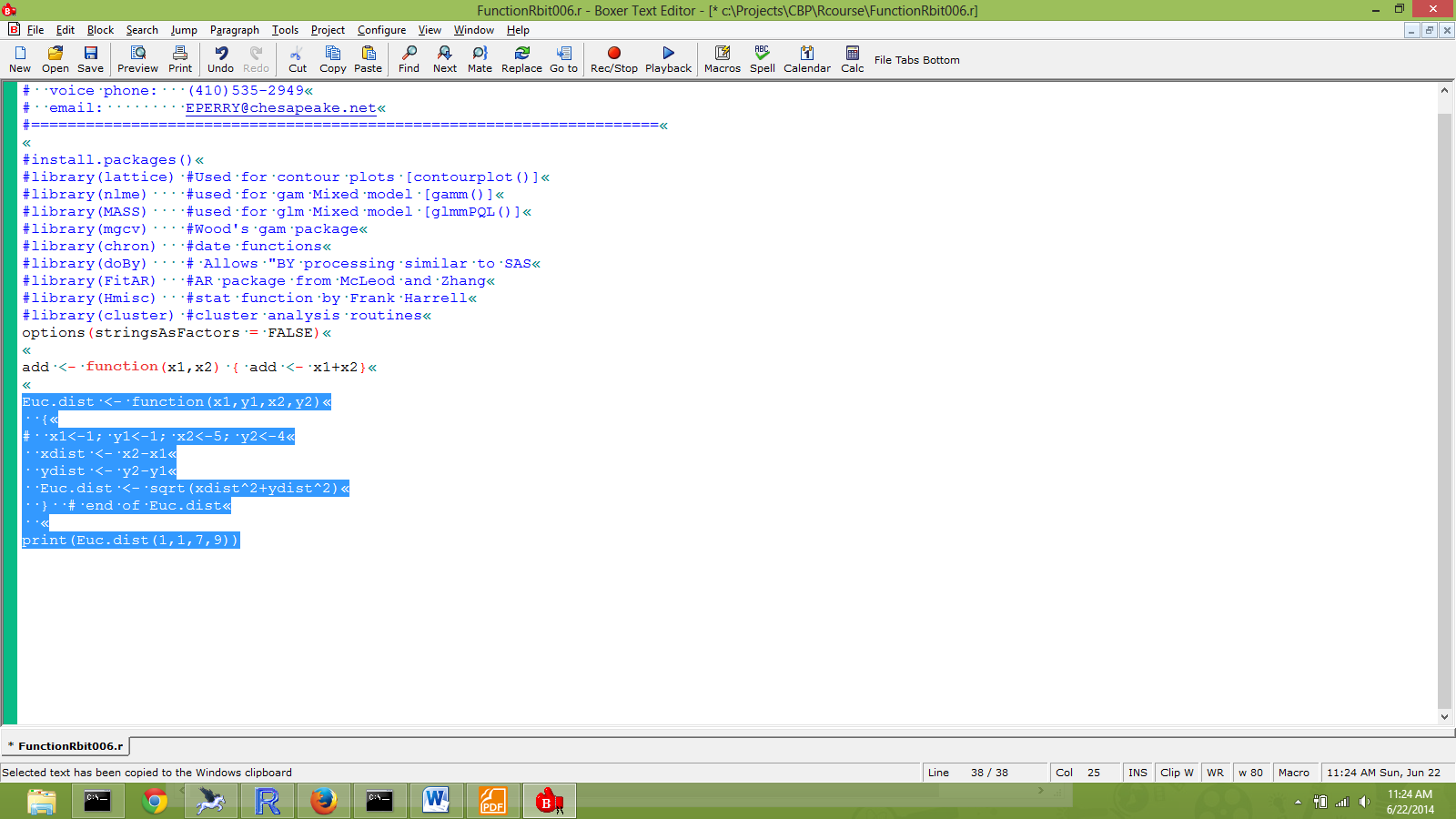
Now my placing the cursor just after the “#” and copying down to just before the “}”, you can drop these statements on the R-console as if they are in open code. Copy like this:

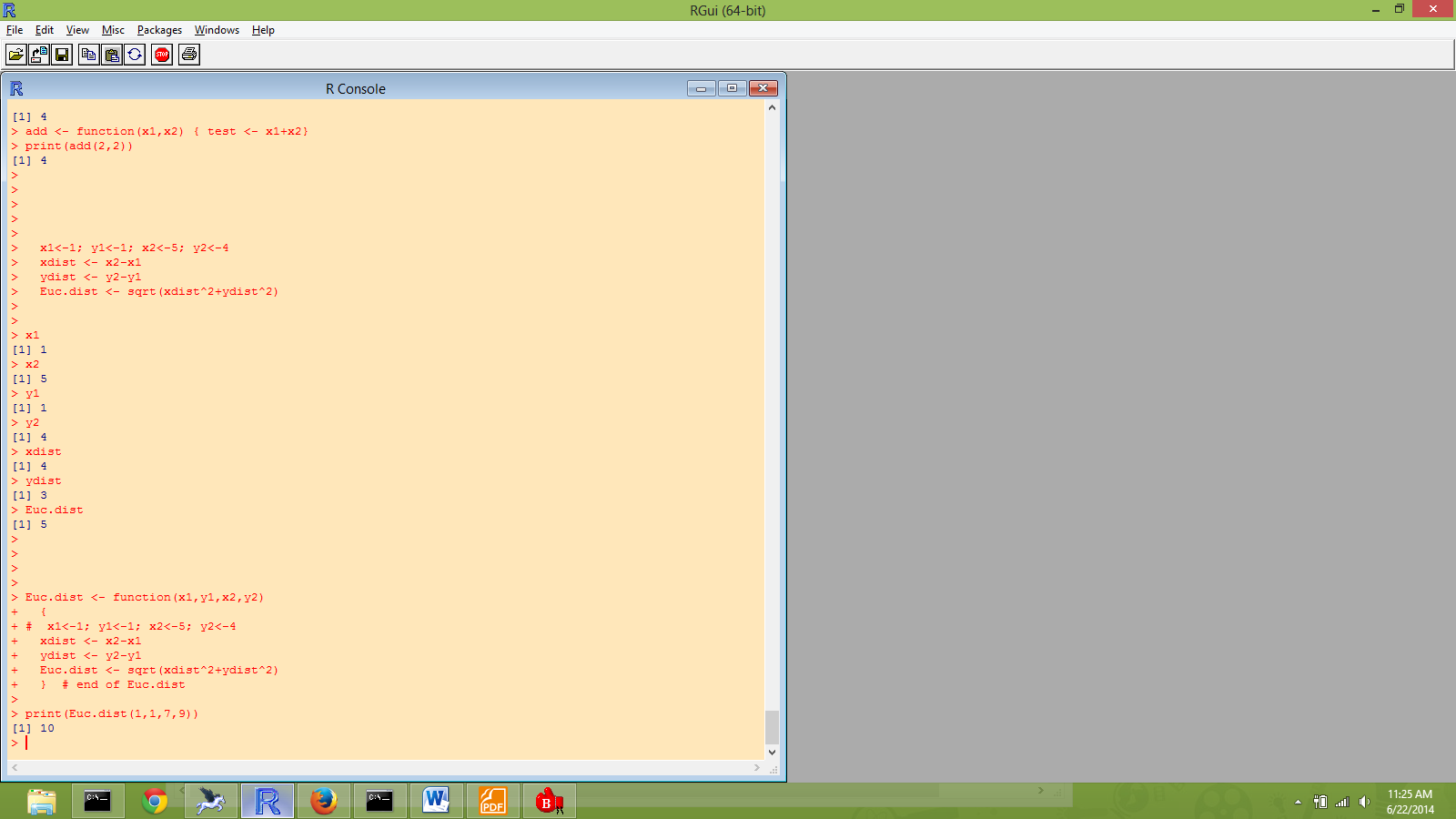


And drop:



You can then easily check if each object has the correct value. Iterate this process until everything works. Then define as test the function like this.





Note that by having the “#” in front of the test arguments, these will now be treated as a comment, and the function will use the arguments that are passed in. There are debugging tools available for R, but by using this trick (which works for lots of languages), I have not needed them.

When working with functions, it is important to understand the concept of scope. Notice that when I did my open code test of statements in Euc.dist(), I created objects x1,x2,y1, and y2. However, when I defined and called Euc.dist (), I used different values for the arguments and Euc.dist() used x1,x2,y1,and y2 to calculate it’s result and it got the answer for the arguments. It did not get confused by the open code x1,x2,y1, and y2 that are still floating around. That is because the arguments are defined within the scope of the function which is defined by {} and not outside. The same is true of the objects xdist and ydist. They are defined inside the function but not available in open code. Only the last assignment to Euc.dist comes back to open code. If you wanted xdist and ydist to come back you could make the last assignment a vector.

# passing back multiple results

Euc.dist <- function(x1,y1,x2,y2)

{

# x1<-1; y1<-1; x2<-5; y2<-4

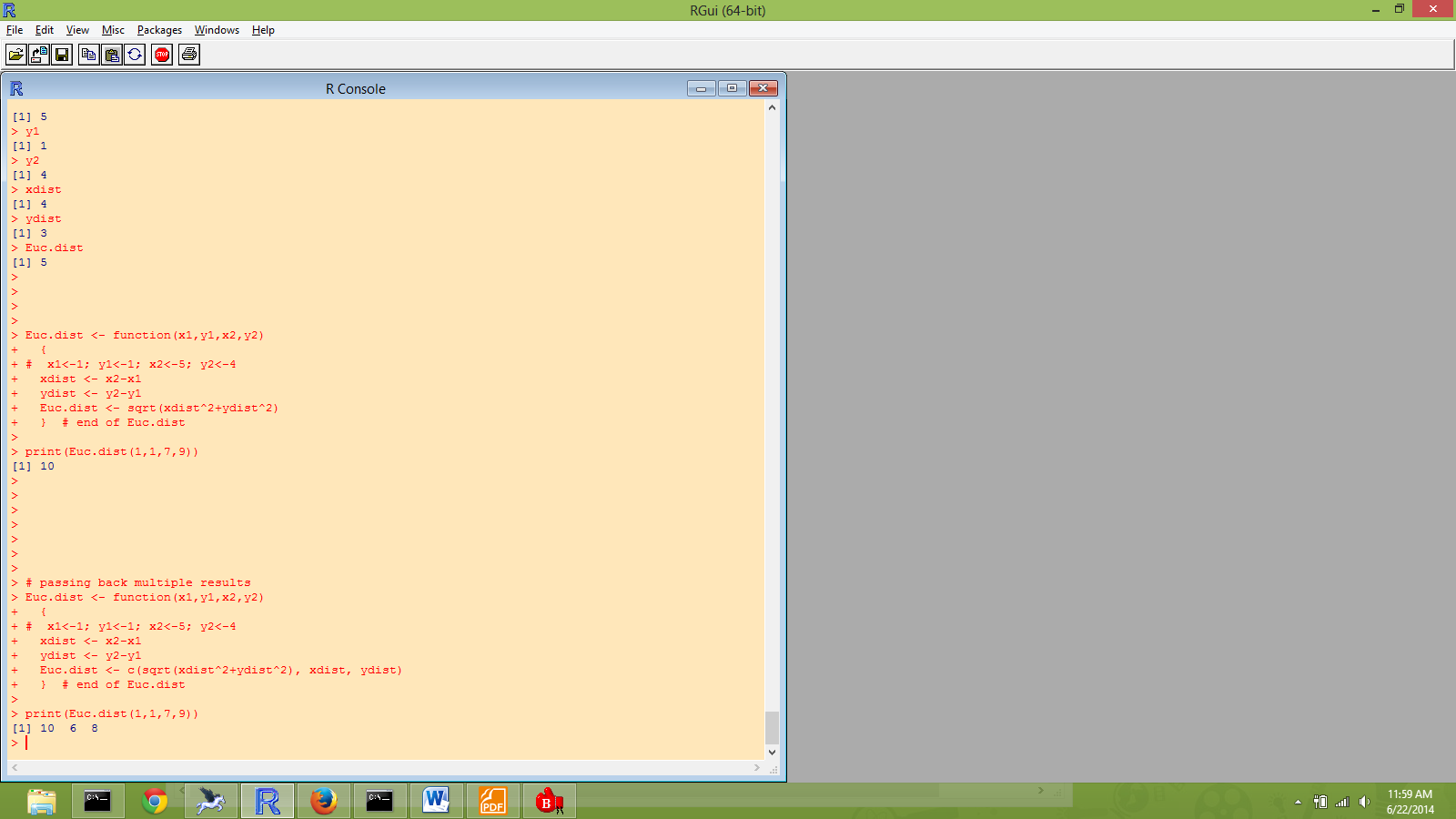
xdist <- x2-x1

ydist <- y2-y1

Euc.dist <- c(sqrt(xdist^2+ydist^2), xdist, ydist)

} # end of Euc.dist

print(Euc.dist(1,1,7,9))



If a statement refers to an object that is not defined as an argument or result inside of the function, the function will look to see if this object is defined in open code. If it finds it, it will use it.

Before we finish with Euc.dist(), I would like to point out that the above it simple, it does not take advantage of R’s matrix language. If you become adept at thinking in terms of vectors, matrices, and matrix algebra calculations, your R code will be more efficient. Here is a revision of Euc.dist() using a vector approach.

# a more R-like version

Euc.dist <- function(x1,x2)

{

# x1<-c(1,1); x2<-c(5,4)

xdist <- x2-x1

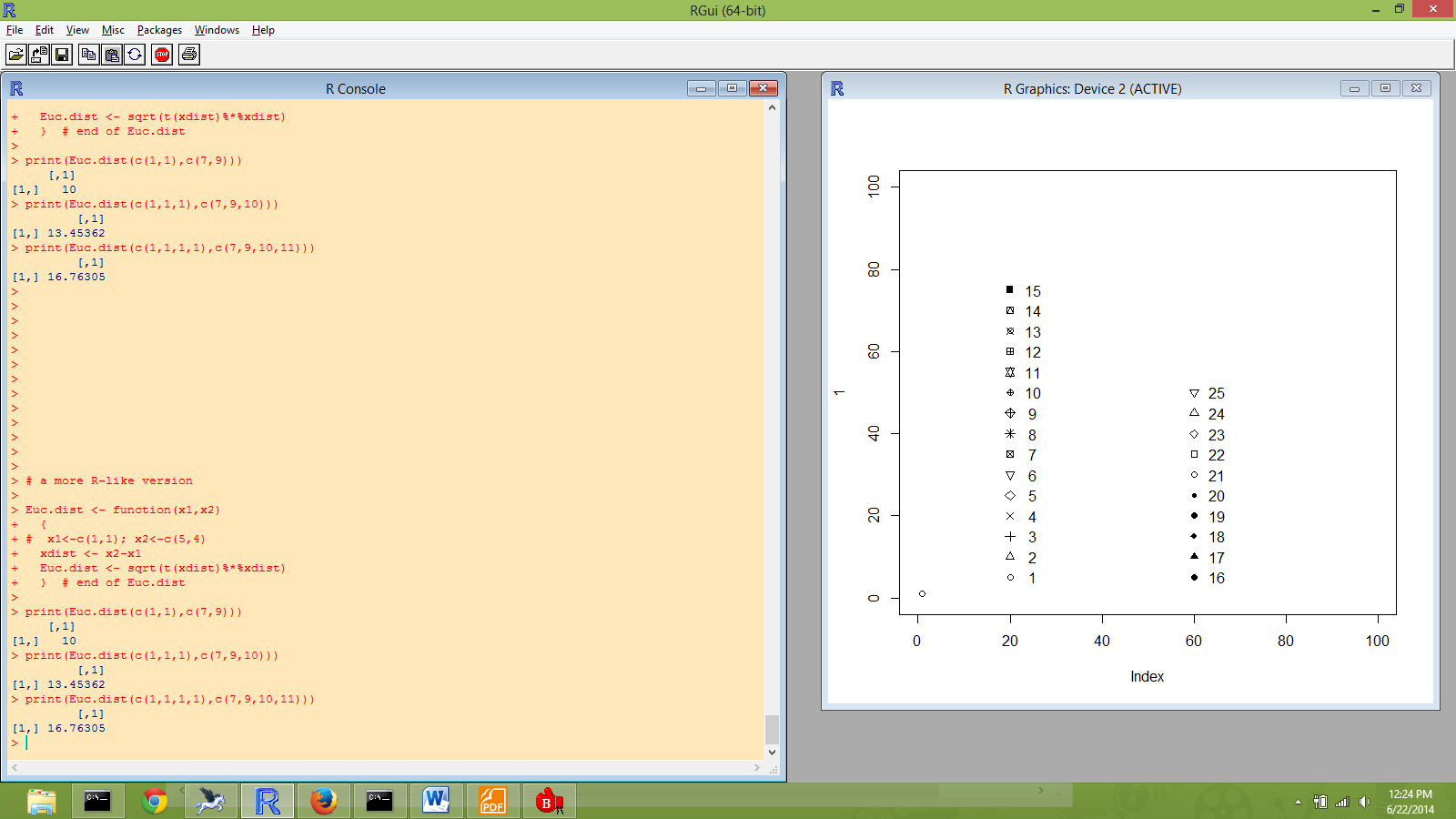
Euc.dist <- sqrt(t(xdist)%\*%xdist)

} # end of Euc.dist

print(Euc.dist(c(1,1),c(7,9)))

print(Euc.dist(c(1,1,1),c(7,9,10)))

print(Euc.dist(c(1,1,1,1),c(7,9,10,11)))



This version can calculate Euclidean Distance in higher dimensional space.

From here I look at a few user defined function I routinely use.

dfsum<-function(df,Nprt=10)

{

print(paste("number of rows = ",nrow(df)))

print("Column Names")

print(names(df))

print(summary(df))

Nprt <- min(nrow(df),10)

print(df[1:Nprt,])

}

This function reports a simple summary of a data frame. The summary includes the number of observations, the column names, the information reported by the Base-R function summary(), and prints the 1st 10 rows or all rows if there are fewer than 10. Note that this example shows how to set the default number of rows to 10, but allows the user to choose a different number.

This function emulates the first. Function of SAS

first.dot <- function(x) { first.dot <- c(TRUE,x[2:length(x)] != x[1:(length(x)-1)])}

Here is a function that does a repeated analysis on the 5-day Mattawoman ConMon data similar to the wrapper function I described in the beginning.

# load libraries

library(chron)

library(mgcv)

# set working directory

ProjRoot <- 'c:/Projects/CBP/Rcourse/'

setwd(ProjRoot);

# load some user defined functions

source("C:/Projects/Rtp/dfsum.r")

source("C:/Projects/Rtp/RTF.r")

# read data as usual

datafile <- paste(ProjRoot,"MAT\_5day.csv",sep='');

mat <- read.table(datafile, header=TRUE, sep=",", na.strings="NA", dec=".", strip.white=TRUE,stringsAsFactors = FALSE)

dfsum(mat)

# define a time variable (this uses chron library)

mat$time <- times(paste(mat$Time,'00',sep=':'))

# define an analysis function

diel.gam <- function(day)

{

# day <- '3/21/2006'

# select data for specified date

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

# fit gam model to selected data

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

# get predicted values from gam

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')]

# 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)

} #end diel.gam

# test for one date

diel.gam('3/22/2006')

# call function for each date in mat

daylist <- unique(mat$Date)

diel.gam(daylist[1])

diel.gam(daylist[2])

diel.gam(daylist[3])

diel.gam(daylist[4])

diel.gam(daylist[5])

# call function for each date in mat using loop and save graphics output to pdf

pdf(paste(ProjRoot,"AllDielplots",".pdf",sep=''),width=9,height=6.5)

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

dev.off()