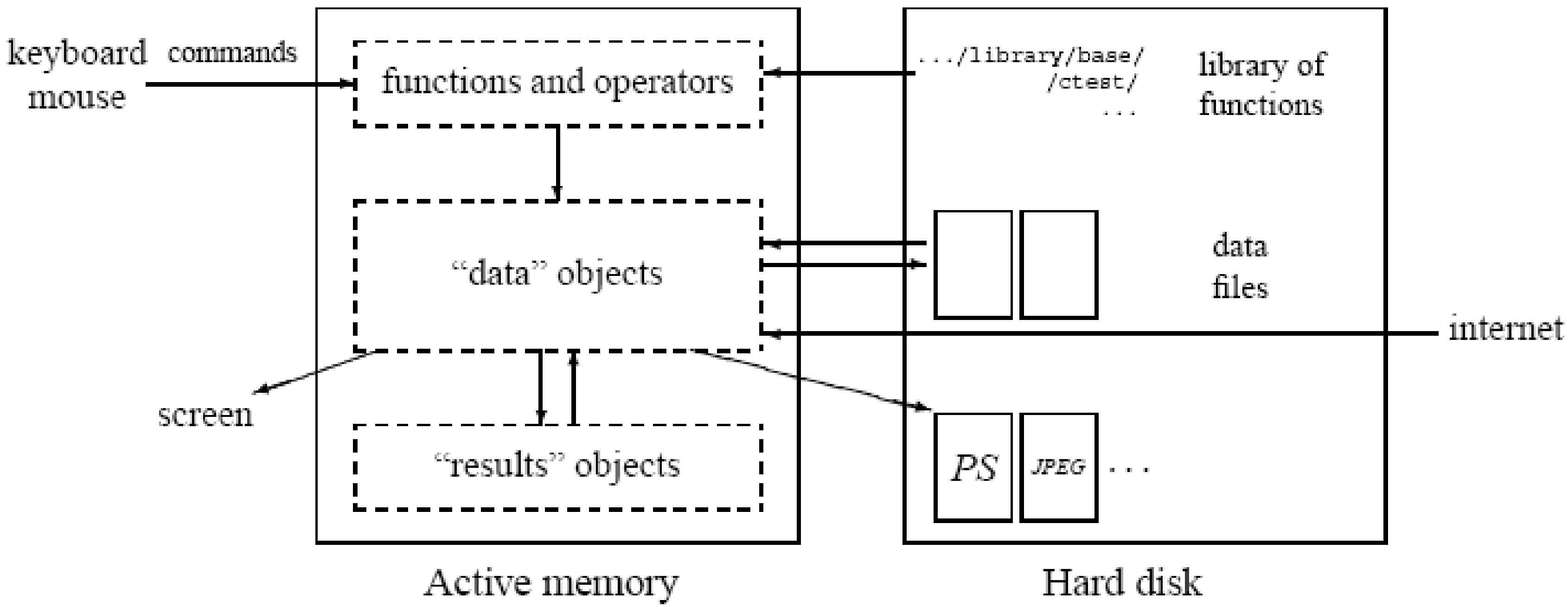
Introduction to R

Also, see cran.**r**-project.org/doc/manuals/**R**-intro.pdf



A schematic view of R (from E. Paridis, R for Beginners, 2002)

Why R?

- free!
- runs on a variety of platforms including Windows, Unix and MacOS.
- provides an unparalleled platform for programming new statistical methods in an easy and straightforward manner.
- contains advanced statistical routines not yet available in other packages.
- has state-of-the-art graphics capabilities.

R Tutorials

1. **P. Kuhnert & B. Venables,** [An Introduction to R: Software for Statistical Modeling & Computing]
2. **J.H. Maindonald,** [Using R for Data Analysis and Graphics]
3. **B. Muenchen,** [R for SAS and SPSS Users]
4. **W.J. Owen,** [The R Guide]
5. **D. Rossiter,** [Introduction to the R Project for Statistical Computing for Use at the ITC]
6. **W.N. Venebles & D. M. Smith,** [An Introduction to R]

- Variables, data, functions, results, etc, are stored in the active memory of the computer in the form of *objects* which have a *name*. The user can do actions on these objects with *operators* (arithmetic, logical, and comparison) and *functions* (which are themselves objects)

- Let’s illustrate by starting with the command line in a new session. You’ll see “>” starting each line
- To quit R, use >q()

R Overview

- Most functionality is provided through built-in and user-created functions and all data objects are kept in memory during an interactive session
- Basic functions are available by default. Other functions are contained in packages that can be attached to a current session as needed

Show current *working directory*, where data objects will be stored

```
> getwd()
>setwd("C:\\Users\\feizou\\Desktop\\Teaching\\Bios663-2019")
```

Sets working directory. It's a good idea to manage different projects separately into different working directories.

```
> # comments follow the '#' sign
> a=10 # R ignores everything after '#'
```

```
> n
Error: Object "n" not found
> n=10
> n
[1] 10
> n<-15
> n
[1] 15
> n=15; a=20
```

I typed “n” and hit return

Doesn’t exist yet

Now I assign (using `= or =`) the value 10 to object n

Now it exists

Alternate way of assigning.

Multiple commands on same line using semicolon

If the object already exists, its previous value is erased (the modification affects only the objects
in the active memory, not the data on the disk).
Note: R is a case sensitive language. N and N are two different objects

```
> n==20
[1] FALSE
> n==15
[1] TRUE
> n<10
[1] FALSE
> n>10
[1] TRUE
> n==n
[1] TRUE
> a<-15          An assignment.
> a< -15         A query.
[1] FALSE
> n=a            An assignment.
>n==a            A query.
[1] TRUE
```

Functions need an argument, provided in parentheses

```
> a=10
> sqrt(a)
> [1] 3.162278
```

Object names cannot start with digits, but they can have digits within them. For long names, it's a good idea to use separators such as "." or "_"

Names that are already in use by R (usually as functions) are a bad idea. Single letter names could be a problem, like "t" (used for the transpose function, etc.). But it's easy to check.

Removing/deleting objects from the current working directory

```
> objects()
```

Similar

```
> ls()

> rm(n)
```

Object is removed

```
> objects()
```

```
> d
Error: Object "d" not found
> t
function (x)
UseMethod("t")
<environment: namespace:base>
```

Variable data **types** – character, numeric, logical

```
> a="scooby doo"
> a
[1] "scooby doo"
> is.character(a)
[1] TRUE
> typeof(a)
[1] "character"
```

CHARACTER

functions that start with “is” often can answer a question about an object

```
> a=10
> is.numeric(a)
[1] TRUE
> is.character(a)
[1] FALSE
>
> a=10/sqrt(2)
> a
[1] 7.071068
> is.numeric(a)
[1] TRUE
> a^100
[1] 8.881784e+84
> a^1000
[1] Inf
> a^(-1000)
[1] 0
```

NUMERIC

Too big

Too tiny

```
> a=T
> a
[1] TRUE
> a=TRUE
> a
[1] TRUE
> a+1
[1] 2
```

LOGICAL

Treats TRUE as 1, FALSE as zero when numeric operations performed

R as a calculator

```
> a=2
> b=10
> c=3
>
> (-b+sqrt(b^2-4*a*c))/(2*a)
[1] -0.3205505
> (-b-sqrt(b^2-4*a*c))/(2*a)
[1] -4.679449
```

R object types

- **Vector:**
  - a one-dimensional array of arbitrary length. Subsets of the vector may be referenced. All elements of the vector must be of the same data type--numerical, character, etc.
- **Matrix:**
  - a two-dimensional array with an arbitrary number of rows and columns. Subsets of the matrix may be referenced, and individual rows and columns of the matrix may be handled as vectors. Again all elements of the matrix must be of the same data type.
- **Array:**
  - as a matrix, but of arbitrary dimension.

Source: http://www.ma.hw.ac.uk/~stan/R/Rnotes.pdf

So far we have assigned a single element to each object. This is a special case of a vector.

The “c” function concatenates things into a single vector

Special case of numeric type

```
> a=c(1,2,3,4)
> a
[1] 1 2 3 4
> a=(1:4)
> a
[1] 1 2 3 4
> typeof(a)
[1] "integer"
> is.numeric(a)
[1] TRUE
> is.vector(a)
[1] TRUE
>
```

So far we have assigned a single element to each object. This is a special case of a vector.

```
> a
[1] 1 2 3 4
> a=a+1
> a
[1] 2 3 4 5
> a=c(a, "scooby doo")
> a
[1] "2"            "3"            "4"
"5"            "scooby doo"
> typeof(a)
[1] "character"
>
```

What does this do?

We say that a was “coerced” into the character data type

Simple plots

```
> a=2
> b=10
> c=3
> x=c(-50:50)/10
>
> y=a*x^2+b*x+c
> plot(x,y)
> abline(0,0)
```

Vector from what value to what value?

Another vector

Simple scatterplot – two vectors must have elements in the same order

Draws a line on a plot with intercept 0 and slope 0

Matrix outer product

```
> a=c(1:11)
> b=c(-5:5)
> d=outer(a,b,FUN="*")
> d
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11]
 [1,]   -5   -4   -3   -2   -1    0    1    2    3     4     5
 [2,]  -10   -8   -6   -4   -2    0    2    4    6     8    10
 [3,]  -15  -12   -9   -6   -3    0    3    6    9    12    15
 [4,]  -20  -16  -12   -8   -4    0    4    8   12    16    20
 [5,]  -25  -20  -15  -10   -5    0    5   10   15    20    25
 [6,]  -30  -24  -18  -12   -6    0    6   12   18    24    30
 [7,]  -35  -28  -21  -14   -7    0    7   14   21    28    35
 [8,]  -40  -32  -24  -16   -8    0    8   16   24    32    40
 [9,]  -45  -36  -27  -18   -9    0    9   18   27    36    45
[10,]  -50  -40  -30  -20  -10    0   10   20   30    40    50
[11,]  -55  -44  -33  -22  -11    0   11   22   33    44    55
```

A single element

```
> d[3,2]
[1] -12
```

Both are vectors

A column

```
> d[,2]
 [1]  -4  -8 -12 -16 -20 -24 -28 -32 -36 -40
-44
```

A row

```
> d[3,]
 [1] -15 -12  -9  -6  -3   0   3   6   9  12
15
```

```
> image(d)
```

Image of a matrix, color corresponds to matrix elements

```
> smaller.d=d[2:3,4:8]
> smaller.d
     [,1] [,2] [,3] [,4] [,5]
[1,]   -4   -2    0    2    4
[2,]   -6   -3    0    3    6
> smaller.d^2
     [,1] [,2] [,3] [,4] [,5]
[1,]   16    4    0    4   16
[2,]   36    9    0    9   36
> temp.matrix=matrix(c(1:10),5,2)
> temp.matrix
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

Subset of matrix is a matrix (unless one of the dimensions is 1, in which case it's a vector)

Many operations operate element-wise

rows

columns

Q: what is `temp.matrix[2,]`?

The apply function

```
> temp.matrix
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
> apply(temp.matrix,1,mean)
[1] 3.5 4.5 5.5 6.5 7.5
> rowMeans(temp.matrix)
[1] 3.5 4.5 5.5 6.5 7.5
> rowSums(temp.matrix)
[1]  7  9 11 13 15
> apply(temp.matrix,1,var)
[1] 12.5 12.5 12.5 12.5 12.5
> apply(temp.matrix,2,var)
[1] 2.5 2.5
```

Faster way to do the same thing

No analogous function rowVars to speed things up

Matrix

```
A <- matrix(c(1,2,4,1), ncol=2)
B<- matrix(c(1,4,5,6), ncol=2)
A*B
A%*%B
t(A)
det(A)
solve(A)
eigen(A)
```

```
> temp.array=array(1:24,c(2,3,4))
> temp.array
, , 1

     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

, , 2

     [,1] [,2] [,3]
[1,]    7    9   11
[2,]    8   10   12

, , 3

     [,1] [,2] [,3]
[1,]   13   15   17
[2,]   14   16   18

, , 4

     [,1] [,2] [,3]
[1,]   19   21   23
[2,]   20   22   24
```

Q: what is `temp.array[2,2,2:3]`?

The apply function on arrays

```
> temp.array=array(1:24,c(2,3,4))
> apply(temp.array,1,mean)
[1] 12 13
> apply(temp.array,2,mean)
[1] 10.5 12.5 14.5
> apply(temp.array,c(1,2),mean)
     [,1] [,2] [,3]
[1,]   10   12   14
[2,]   11   13   15
> apply(temp.array,c(1,3),mean)
     [,1] [,2] [,3] [,4]
[1,]    3    9   15   21
[2,]    4   10   16   22
>
```

Factors and tapply

Apply a function to "ragged" arrays

```
>a=c("NC","NC","NY","NC","GA","GA","NY","FL","GA","GA",
"SC","SC","NY","NY")
> scores=c(0.33,0.00,-1.75,0.74,-0.30,-0.36,-
0.57,0.83,0.75,-1.42,-0.83,-0.19,-0.10,1.68)

> tapply(scores,a,FUN=mean)
        FL         GA         NC         NY         SC 
 0.8300000 -0.3325000  0.3566667 -0.1850000 -0.5100000 
> tapply(scores,a,FUN=sd)
       FL        GA        NC        NY        SC 
       NA 0.8862421 0.3707200 1.4239499 0.4525483 
>
```

Missing value

Simple histograms and boxplots

```
> hist(scores)
> boxplot(scores)
> boxplot(scores~a.f)


> plot(scores,a.f)
> plot(a.f,scores)
```

R object types, cont.

- **Data frame:**
  - a set of **data** organized similarly to a matrix. However, each column of the data frame may contain its own type of data. Columns typically correspond to variables in a statistical study, while rows correspond to observations of these variables. A data frame may be handled similarly to a matrix, and individual columns of the data frame may be handled as vectors.
- **List:**

Powerful feature of R

  - an arbitrary collection of other **R** objects (which may include other lists).

Source: http://www.ma.hw.ac.uk/~stan/R/Rnotes.pdf

R object types, cont.

- **function:**

  - Doesn't have a data type. This is an object that takes other objects as arguments and does something with them.

Random variables and distributions

Distribution	R name	additional arguments
beta	`beta`	`shape1, shape2, ncp`
binomial	`binom`	`size, prob`
Cauchy	`cauchy`	`location, scale`
chi-squared	`chisq`	`df, ncp`
exponential	`exp`	`rate`
F	`f`	`df1, df2, ncp`
gamma	`gamma`	`shape, scale`
geometric	`geom`	`prob`
hypergeometric	`hyper`	`m, n, k`
log-normal	`lnorm`	`meanlog, sdlog`
logistic	`logis`	`location, scale`
negative binomial	`nbinom`	`size, prob`
normal	`norm`	`mean, sd`
Poisson	`pois`	`lambda`
Student's t	`t`	`df, ncp`
uniform	`unif`	`min, max`
Weibull	`weibull`	`shape, scale`
Wilcoxon	`wilcox`	`m, n`

Random variables and distributions, cont.

Random variable   Quantile   Density

Example: rnorm, qnorm, dnorm, etc.

```
> plot(rnorm(100,0,1),rnorm(100,0,1))

> x=rnorm(100,0,1)

> y=x+rnorm(100,0,1)

> x=c(-50:50)/10

> y=dnorm(x,mean=0,sd=1)

> plot(x,y)

> y=pnorm(x,mean=0,sd=1)

> plot(x,y)
```

Searching for information/functions

```
> help(regression)
No documentation for 'regression' in
specified packages and libraries:
you could try 'help.search("regression")'
>
>
> help("lsfit")
> help.search("regression")
```

Now we’re getting somewhere (see popup)

Searches for anything with `regression’ in it

R hints from Karl Broman's website, with additions

* Use the "recording" feature of the graphics window.

After making a plot, select (from the menu bar) History:Recording. After making further plots, you may use the Page Up and Page Dn keys to go back and forth through the previous plots you've made.

* Save your analysis commands in a text file.

Use a text editor (such as Notepad or an enhanced version such as EditPad) to edit a file containing your commands. Use the Windows keyboard shortcut Alt:Tab to quickly switch between R and your text editor.

Alternatively, use File->New Script for an R Editor.

Note that commands you type in R are saved (temporarily) in the file .Rhistory (in the same directory as your .RData file). You may wish to peruse (or copy and paste from) this file and keep a permanent record your analysis commands.

* To delete all objects in your workspace, type (from the R prompt)

```
rm(list=ls())
```

This is useful in debugging to ensure that your code does not rely on existing values for data objects, so that you can mimic what a naïve user would encounter by using your code. However, it's also very dangerous, as it removes all objects from your workspace.

Additional Resources

http://cran.r-project.org/doc/manuals/R-intro.html
http://cran.r-project.org/doc/contrib/Paradis-rdebuts_en.pdf
Quick reference card: http://people.musc.edu/~slateeh/Rintro/RQuickReference.pdf

```
# VERY BASIC THINGS
x = 1:10
x
y = 2
z = x + y
logi = x < 4
logi
mode(x)
mode(y)
mode(logi)
as.numeric(logi)
mat = matrix(z, ncol = 2)
mat
mat2 = matrix(z, ncol = 2, byrow = TRUE)
dim(mat)
mat[1,2]
mat[,1]
```

```
# VERY BASIC THINGS, cont.
mat[2,]
colnames(mat) = paste("col", 1:2)
colnames(mat)
mat
mat[,"col 2"]
t(mat)
arraydat = array(1:27, dim = c(3,3,3))
arraydat
mylist = list(x = x, y = y, z = z, mat = mat, arrayd
= arraydat)
mylist[["arrayd"]]
mylist$arrayd
mylist$x
mylist[["y"]]
mydata = data.frame(x = x, y = y, logi = logi)
names(mydata)
```

transpose

Creating a list

Double brackets used to indicate list “elements”

Name

Sub-object assigned to that name

```
# CREATING AND MANIPULATING DATA
a = c(1,2,3)
a^2
a + 23
b = 4:6
d = seq(8, 12, by = 2)
e = c(a,b)
f = rep(a, 3)
g = rep(a, b)
m1 = cbind(a, b)
m2 = rbind(a, b)
matrix(e, ncol = 2)
matrix(e, ncol = 2, byrow = TRUE)
h = list(x = a, y = d)
apply(m1, 2, sum)  ### sum the columns
lapply(f, sum)### sum the elements in each list item
lapply(h, sum)
colors = c("blue", "green", "red")
paste("My favorite color is", colors)
```

A sequence of values

Repeating values

Column and row “bind” statements make matrices

```
# CREATING AND MANIPULATING DATA, CONT.
mydata = data.frame(aa = a, bb = b, dd = d)
mydata$aa
mydata["aa"]
mydata$zz = mydata$aa + mydata$bb
mydata$xx    ### NULL, doesn't exist
mydata$ff = g    ### wrong because g is length 15
mydata    ### error given above, nothing has been changed
a < 2
a < c(0,4,2)
a < 2 & a > 1
z = c(17, 2, 4, 23, 40, 4)
z + c(0,1,2)
sum(z)
prod(z)
max(z)
z[z > 4]
z[z==max(z)]
which.max(z)
unique(z)
```

Logical "and" – see ?Logic

Subsetting by logicals

Returns index of max

Unique set of elements within object

```r
#EXAMPLES OF PLOTS

x=rnorm(100)
y=rnorm(100)+x
plot(x,y)

x=rnorm(1e5,0,2)
y=rnorm(1e5,1,3)+x
plot(x,y,pch=".")

x=seq(1,20,by=.5)
y=2^x-x^2
plot(x,y,type="b")
plot(x,log(y))
plot(log(x),log(y))
z=log(y)
print(is.na(z))
grep(T,is.na(z))
```

Change print character

is.na returns logical vector for missing or not

return indices for which the statement is true

#EXAMPLES OF PLOTS, CONT.

```
z = lm(dist ~ speed, data = cars)
cars
plot(cars)
plot(cars$speed,cars$dist)
abline(z) # equivalent to abline(reg = z) or
abline(coef = coef(z))
cars.loess=loess(dist ~ speed, data = cars)
lines(cars$speed,cars.loess$fitted,lwd=3)
# what is the effect of dropping the maximum observation?
which.biggest=which.max(cars$dist)
z.new=lm(dist[-which.biggest] ~ speed[-which.biggest], data
= cars)
abline(coef = coef(z.new),col="green")
cars.loess.new=loess(dist[-which.biggest] ~ speed[-
which.biggest], data = cars)
lines(cars$speed[-
which.biggest],cars.loess.new$fitted,col="green",lwd=3)
```

#PLOTS, CONT.

Identify "n" points by clicking

```
plot(cars)
plot(cars$speed,cars$dist)
abline(z)
which.identified=identify(cars,n=1)
z.new=lm(dist[-which.identified] ~ speed[-
which.identified], data = cars)
abline(coef = coef(z.new),lty=2)
points(cars[which.identified,],pch="O",col=2,cex=2)

plot(rnorm(100),rexp(100,1),xlab=expression(hat(mu)[x]),
ylab=expression(eta^tau),
main=expression(paste("Scatterplot of ", eta^tau, "
versus ", hat(mu)[x])))
```

PLOTS, CONT.

Putting text on plot

```
x=rnorm(100);y=x+rnorm(100)
plot(x,y,axes=F)
axis(1)
mtext("I can put text on this side",3)
mtext("I can put text on this side too",4,col=2)
text(0,1,"text1",col="blue")
text(.5,1.5,"text2",srt=90,col="green")
text(.5,-1.5,"text3",col="purple",cex=2)
```

Using “pairs” to explore two-way relationships

```
?swiss
pairs(swiss)
cor(swiss)
swiss.pch=rep("X",dim(swiss)[1])
swiss.pch[swiss$Catholic>50]="O"
swiss.colors=rep("red",dim(swiss)[1])
swiss.colors[swiss$Education>median(swiss$Education)]="
blue"
pairs(swiss,pch=swiss.pch,col=swiss.colors)
```

REGRESSION MODELS

(http://people.musc.edu/~slateeh/Rintro/Rdemo.htm)

```
plot(faithful$eruptions, faithful$waiting, main = "Eruptions
of Old Faithful",
        xlab = "Eruption time (min)", ylab = "Waiting time
to next eruption (min)")
lines(lowess(faithful$eruptions, faithful$waiting, f = 2/3,
iter = 3),
           col = "red", lwd = 3)
fit = lm(waiting ~ eruptions, data = faithful)
abline(fit, col = "green", lwd = 3)
plot(fit, which = 1:3)
```

```
# SIMPLE SYMBOLIC DERIVATIVES AND NUMERIC INTEGRALS
D(expression( x^2), "x")
D(expression( x^2-exp(x)), "x")
D(expression( x^2-log(x)), "x")
D(D(expression(x^2),"x"),"x")
D(D(D(expression(x^2),"x"),"x"),"x")
D(expression( y*x^2), "x")

my.deriv=deriv(~x^2-log(x),"x")
x=1:5
eval(my.deriv)
my.deriv2=deriv(~x^2-log(x)+x*y^2,"x")
y=3:7
eval(my.deriv2)

integrate(dnorm, -1.96, 1.96)
integrate(dnorm, -Inf, Inf)
```

```
# CONDITIONS AND LOOPS ###

# R follows the form  "if (expr_1) expr_2 else expr_3"
# and "{" used to set apart grouped statements

x=1
if (x>1) print("BINGO") else print("BOINGO")
if (x>1) {print("BINGO")} else {print("BOINGO")}
if (x>1) {print("BINGO")} else {print("BOINGO");print("BOINGO
BOINGO")}

# "for" loops: "for (name in expr_1) expr_2"
for (i in (1:10)){
 print(i^2)
 }
# nested loops
for (i in (1:5)){
 for (j in (1:10)){
  print(c(i,j))
  }
 }
```

```
# nested loops
for (i in (1:5)){
 for (j in (1:10)){
  print(c(i,j))
   }
 }

for (i in (1:5)){
 for (j in (i:10)){
  print(c(i,j))
   }
 }
```

 # Working with entire R objects is often much faster than looping

```
start.time=Sys.time()
num.rows=1000
num.cols=1000
big.matrix=matrix(0,num.rows,num.cols)

start.time=Sys.time()
for (i in (1:num.rows)){
 for (j in (1:num.cols)){
  big.matrix[i,j]=i+j
   }
  }
end.time=Sys.time()
end.time-start.time

start.time=Sys.time()
big.matrix=outer(1:num.rows,1:num.cols,FUN="+"
)
end.time=Sys.time()
end.time-start.time
```

Writing functions

```
find.quadratic=function(a,b,c){
 root1=(-b+sqrt(b^2-4*a*c))/(2*a)
 root2=(-b-sqrt(b^2-4*a*c))/(2*a)
 return(list(root1=root1,root2=root2))}

find.quadratic(a=2,b=10,c=3)
find.quadratic(2,10,3)

# Now let's do this completely numerically
error.quad=function(x,a,b,c){
 current=a*x^2+b*x+c
 error=(current-0)^2
 return(error)}

optimize(error.quad,a=2,b=10,c=3,interval=c(-
1,0))
optimize(error.quad,a=2,b=10,c=3,interval=c(-5,-
1))
help(optimize)
```

```
### THIS FUNCTION DOES LINEAR REGRESSION
regression=function(y,x){
 # y is a vector
 # x is the design matrix.
 n=length(y)
 p=dim(x)[2]
 xTx.inv=solve(t(x)%*%x)
 betahat=as.vector(xTx.inv%*%t(x)%*%y)
 resid=y-x%*%betahat
 varhat=(sum(resid^2))*diag(xTx.inv)/(n-p)
 se=sqrt(varhat)
 pval=2*pt(-abs(betahat/se),df=n-p)
 out=list(betahat,se,betahat/se,n-p,pval,resid)
 names(out)=c("betahat","se","t","df","pval","resid")
 return(out)
 }
```

R Workspace

*# save your command history
savehistory(file="*myfile*") # default is ".Rhistory"

recall your command history
loadhistory(file="*myfile*") # default is ".Rhistory“

Graphics

- #Save plots

```
x<-rnorm(100)
y<-2*x+ rnorm(100)
plot(x,y)
pdf("test.pdf")
plot(x,y)
dev.off()
```