#

# Crash Course in Programming R

#

# 1 Preliminaries

#

# (a) About the Context of this material

#

# This assumes that you are sitting at your computer

# with R (version 2.5 or later) open and that your tutor has explained

# some very elementary ideas on the main R console window

# and command file and libraries (ie packages)

#

# The R Console window acts as an output (results) window

# and as command interface window -ie you can type R commands

# at the prompt

#

# This crash course will expose you to many key concepts

# but it will take some time to learn R

#

# NB: be careful about typing, R is case sensitive so R and r and t and T etc

# are different objects

#

# (b) About this file

#

# This is script file produced in R: the # symbol is a comment symbol

# lines beginning with # are ignored in processing.

#

# Script files are used to store sets of R commands and programmes called functions.

# Lines of a script file can be sent to the main r console window for execution.

# Consider the following commands, highlight these with the mouse and use the menu

# edit command in the RGui to despatch them to R console (Run line or selection}

#

#

library (foreign) # make available data import/export commands to other packages

library (graphics) # makes graphics available

x<-rnorm(100) # generates 100 x's ~ N(0,1) and stores in object x

x # display x's in R console

hist(x) # make a histogram of x - appears in a graphic device window

mx<-mean(x) # compute mean and store in scaler mx

mx # display mx in R console

vx<-var(x) # compute variance and store in vx

x # display variance

sx<-sqrt(vx) # compute std and store in sx

sx # display sx

#

# Notice that the symbol <- stands for assignment and replaces the = sign

# prevalent in other languages.

#

# Notice that you are free to name the object on the LHS of <- .

#

# (c) About R and how we use it

#

# R is a statistical package based on the object orientated functional langauge

# S. There are many commands to learn if you wish to use the software merely

# as a statistical package. R has about 2000 statistical functions inbuilt and so

# and because it based on the S langauge we tend to use R has a development

# language ie a tool-kit for developing new statistical techniques or for programming

# complicated statistical calculations. One nice a feature of the language is that

# one can usually some details of the in-built functions.

var

# So we can generally see what is going on and can pick up programming tips by

# following the functions through - there are inevitably basic internal functions whose

# structure we cannot see.

#

# Another nice feature is that the language is vectorized - the basic mathematical object

# is a vector and many of the procedures are designed to exploit this fact. People coming

# from procedural languages like Fortran & C+ find this one of the most difficult aspects

# of R. It takes a while to get used to but take care and persevere.

#

# Nested Looping gives rise to most problems where the efficient procedures in R can

# look strange (at first). Recall that R is interpreted, it is not a complied language,

# so heavily nested loops programmed in conventional ways will be slow compared with

# compiled code, especially recursion. So one needs to learn new programming skills to

# write efficient R code - particularly important are good function construction

# and mastering the programme flow control features.

# However, we are jumping too far ahead.

#

# Yet another nice feature of the language is that one can generally get things working

# qickly and so R is often referred to a templating tool-kit and procedures once working

# in R may be translated into Fortran relatively easily to improve speed - especially

# in large statistical simulation problems. Computations which may take

# weeks in R may take minutes or hours in Fortran.

#

# This crash course concentrates on the programming aspects of R ie on how to

# use the S language language to implement new statistical procedures - it

# therefore deals with programming rather than statistics.

# Naturally these two aspects blend together to solve problems.

#

# The key to writing good R programmes lies in good function contruction and

# ultilisation. A function is a subroutine ie a re-useable piece of code.

# Typically the subroutine takes arguments as input and produces results as output.

# If given new arguments it will produce new results and so on.

# The nice thing about R is that the results at each stage can be stored in an

# object and may act as input to a second function.

# Thus the covariance matrix of the betas in a regression analysis may be input

# into another function which will compute variances of linear cominations of the

# betas. It is this flexibility which makes R so appealing compared to other packages

# which are typically very restricted in this area.

#

# Well enough about background ideas in R: to understand how R works you have to

# first learn how it does vectorised arithmetic and learn the default conventions

# about creating vectors and how to avoid ambiguous (vectorized) calculations.

#

# 2. Vectors & Matrices

#

# (a) Arithmetic with Scalars, Vectors & Matrices

#

# In this section we shall be concerned about the class,

# mode and type and the attributes of the objects created.

# We concentrate on numeric objects but give some examples

# of vectors in other modes such as logical or character.

# We are often need the class of the object such

# as vector, matrix, array, etc . Scalars do not have a class

# even though they are vectors of length 1

# First consider the following scalar assigments

a<-1 # vector of length 1

b<-2 # ditto

c<-3 # ditto

a;b;c # display in R Console

y<-rbind(a,b,c) # create a columm vector

y # display in R console

z<-cbind(a,b,c) # create a row vector

z # display in R console

zc<-c(a,b,c) # concatenate into a numeric vector

zc # display

d<-seq(1:4) # create a vector - ?type

d # display

#

# commands for discovering the properties of an object

#

class(a)

ributes(a)

length(a)

dim(a)

mode(a)

is.vector(a)

is.matrix(a)

#

# So we discover that a is a numeric vector of length 1

# What about the "vectors"

#

class(y)

attributes(y)

dim(y)

mode(y)

is.vector(y)

is.matrix(y)

length(y)

is.numeric(y)

#

# So we discover y is a matrix not a vector!: y was made by rbind

# which bound the 3 scalars into the first row of a 3 x 1 matrix.

# Its attributes are $dim [1] 3,1 and

# $dimnames - note it has two dimensions & that the row names 'a', 'b', 'c'

# are stored but there are no column names - none were specified.

# These properties can be retrieved from the matrix object y and stored

# in another object - method comes later.

# To summarise y is a typical numeric column vector of class matrix

# with row names assigned

#

#So what about z

#

class(z)

attributes(z)

dim(z)

is.vector(z)

is.matrix(z)

is.numeric(z)

length(z)

#

# So again z is a matrix of order 1 x 3 is not of class vector.

# It is a numeric row vector with columm names assigned - notice

# it is the second component of dimnames which contain the

# column names.

#

# So what about zc?

#

class(zc)

attributes(zc)

dim(zc)

is.vector(zc)

is.numeric(zc)

length(zc)

#

# So zc is a numeric vector

#

# So what about d

#

class(d)

attributes(d)

dim(d)

is.vector(d)

is.numeric(d)

length(d)

#

# Notice (a) that we have a new class called integer which is not

# helpful (b) attributes are null - again not helpful (c) the dimension

# is null - not helpful. The other information is useful - and we learn

# that it is a numeric vector of length 4. But is it a row vector or

# a column vector - no use if we do not know.

# consider

#

d # careful - it prints like a row

t(d) # take the transpose - the transpose is a row! - so d is a column

t(d)%\*%d # computes the inner product

e<-d%\*%t(d) # computes the outer product - store in e

e # display outer product

f<-solve(e) # store inverse in f

f # display inverse - this is singular - why?

g<-diag(rnorm(4)) # make a random diagonal matrix 4 x 4

g # display

h<-solve(e+g) # jitter principal diagonal of e and invert

h # display inverse

#

# Shows use of some matrix techniques for you to follow-up.

# Notice the matrix multiplication sign %\*%

# ?solve #gets details of usage and examples of the function

# ?diag #ditto

# ?class #ditto

# ?seq #ditto

# But the key thing here is that the function seq produces a column

# vector by default unlike cbind which produces a row vector and

# rbind which produces a column vector (both of class matrix).

#

# We are not finished yet there are many other ways to produce vectors

# and you need to know the defaults

#

#What about x = rnorm(100 ) = 100 N(0,1) random variables?

#

x

class(x)

attributes(x)

dim(x)

is.vector(x)

is.numeric(x)

length(x)

t(x) # is a row vector so x is column vector

#

# Now consider some other ways of making vectors

#

i<- rep(1:4, 2) # use rep function to generate 2 sets of 1:4

i #display

is.vector(i)

j<- rep(1:4, each = 2) # use rep again but not the same => 1 1 2 2 3 3 etc.

j #display

is.vector(j)

k<-runif(10) # generates 10 uniform random numbers

k #display

is.vector(k)

l<-t(i)%\*%i # inner product

l # display

m<-length(l) # compute length = 1 so i is column vector

m # display

#

# Similar for j produced by rep so we are on the brink of

# deducing that the R default is to produce column vectors as in mathematics

#Lets check k

#

n<-t(k)%\*%k # inner product

n # display

o<-length(n) # compute length = 1 so i is column vector

o # display

#

# Consider the logical vector

p<-rep(cbind(T,T,F), 2) # a logical vector - but it is not a row vector!!!

p

t(p) # check: t(p) is a row so p is a column

class(p)

is.vector(p)

attributes(p)

dim(p)

length(p)

attributes(p)

#

# it usually does not matter whether this is a row or column viz

#

q<-rep(rbind(T,T,F), 2) # a logical vector

q

class(p)

is.vector(q)

attributes(q)

dim(q)

length(q)

#

# You can investigate the properties of logical matrices

#at length

#

# Character vector

#

r<-rep(c("yes","no"), c(4,2)) # note the c(...)construct is a concatenation function

r

class(r)

is.vector(r)

attributes(r)

dim(r)

length(r)

#

# Check these out

#

# So far we have looked at vectors of the same mode: sometimes vectors

# and other structures can be of mixed modes (numeric

# logical and character) depending on the data. These are usally handled

# as dataframes - more later.

# Consider our previous p<- rep(cbind(T,T,F), 2)

#

p

t(p)

s <- as.numeric(p) # coerces p to be numeric

s

#

# Must be careful - coercion is often implicit with some functions

# - to the most "informative" mode- and so one cannot always

# coerce coercion to produce what you want

#

r

tr<-as.numeric(r)

tr

t(tr)

#

# The take home message here is that one must

# be careful when making vectors and one must

# know how to find out what kind of vector one has

# created.

#

# (b) Making Matrices

#

# create a matrix - see ?matrix

# usage

#matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL)

# Example

#

A<-matrix(1, 3, 3) # define matrix - see ?matrix

A

B<-diag(A) # store the diagonal as a vector - see ?diag

B

C<-diag(B) # Make a diagonal matrix with vector B

C

D<-A+C # Add togetehr

D

E<-solve(D) # Invert - see ?solve

E

I<-D%\*%E

I

#

# We have created a number of objects as we went along

# Here they are

#

objects() # Lists the working directory of objects

remove (c(p,q,r,s)) #removes p q r s - see ?rm

ls () # Lists the working directory of objects -?ls

#

#

# 3. Basics about functions

#

# A function is a piece of re-usable code.

# It has a specific structure.

# It has an argument list of input objects.

# It has an results list of output objects.

# The function may be assigned to another object.

# This object may in turn be interrogated to recover

# the results objects for input into another function.

# The function must be initiated ie made known to R

# before it can be used.

# The function is used by calling it - the format of

# the call must be understood.

#

# Here is the general format of the structure of a function

# myfun1<- function (a1,a2,a3) # arg list goes in (...)

# { # open the function body

# # body of function contains R commands

# x<-sqrt(a1\*a1+a2\*a2+a3\*a3) # last assignment is the value of the function

# } # close the function body

#

# The name of this function is myfun1, the input arguments are 3

# scalars, the last assignment is the value of the function ie

# myfun1 has value x

# The function is introduced to R by highlighting the code

# and running the function viz

#

 myfun1<- function (a1,a2,a3) # arg list goes in (...)

 { # open the function body

# # body of function ie R commands

 x<-sqrt(a1\*a1+a2\*a2+a3\*a3) # last assignment is the value of the function

 } # close the function body

#

# The function is called using

# myfun1(a1,a2,a3) viz

# Let

#

a1<-1

a2<-2

a3<-3

myres1<-myfun1(a1,a2,a3)

myres1

#

# Notice that x <- sqrt(...) is local to the function and

# it did not overwrite our original x = rnorm(100) in

# the master working directory

#

x # checking x in master working directory

#

# This function format no use of course if we need more

# than one item out of the funtion.

#

# Here is a more general version of the function

# which uses a return list to return the values

# length of the input vector and the Euclidean Distance

#

dvec<-cbind(a1,a2,a3) # creates a row vector 1 x 3

dvec # display

myfun2<- function (dvec)

 {

 n<-length(dvec) # length of dvec

 x<-sqrt(dvec%\*%t(dvec)) # euclidean distance

 return( list(Len=n, Edist=x) ) # return 2 quantities

 }

#

# Introduce the function before running

# ie highlight it and send to r console

#

myres2<-myfun2(dvec)

myres2

#

# Now notice that the two returned values are preceded

# by $ signs ie in object myres2 we have $Len and $Edist

# The $signs mean that this information is retrievable

# from this object using the extraction method - look

#

mylen<-myres2$Len

mylen

myedist<-myres2$Edist

myedist

#

# This is the method of returning values from a function

# whence the returned values are now back in the master

# working directory or the master segment of the R programme

# Functions execute in their own "frames" - not in this course.

#

# Now Recall the object e and observe how to use the

# extraction method to interrogate and store its

# attributes ie the dimension of the matrix.

#

e

myeatt<-attributes (e)

myeatt

edim<-myeatt$dim

edim

#

#

# 4 Basics about programme flow and control

#

#

# You need to know how to do loops and conditional

# branching and generally how to organise the

# flow of computation in a prgramme.

#

# (a) The if statement is of the form:

#

# if (condition) {expression}

#

# note the different usees of parentheses

#

# Example

#

hist(x)

myifx <-function (x)

 {

 if (!is.numeric(x)) {stop("Gil says ~ Data must be numeric")}

 n=length(x)

 x<-x^2 #notice this squaring technique using vectorized arith

 c<-sum(x) # could have used t(x)%\*%x which is faster

 return(list(df=n,chi=c))

 }

myifxres<-myifx(x)

myifxres

#

# create a character vector

#

testc<-c(T, "char", pi)

testc

class (testc)

attributes(testc)

is.vector(testc)

length(testc)

#

# Give character vector to function myifx

#

myifxres2<-myifx(testc)

myifxres2

#

# myifxres2 does not exist since we did not do

# an assignment within the function.

#

#

# (b) For loops

#

# Basic specification

#

# for (i in 1:n )

# {

# body of loop

# }

#

# Specification of double ie nested loop

#

# for (i in 1:n) # begin first loop

# { # open body of first loop

# r commands

# for( j in 1: m) { body of second loop} # specify second loop - open and close

# } # close first loop

#

# Example 1: of Loop recoding col\_2 into a binary variable based on col\_1

#

m1<-matrix(nrow=20,ncol=4) # create matrix 20 x 2 matrix

m1 # display

m1[,1]<-runif(20) # put 20 uniform random number in col 1

# # notice use of comma in indexing 1st column

for (i in 1:20)

{

if (m1[i,1] <= 0.5) m1[i,2]<-0

else if (m1[i,1] > 0.5) m1[i,2]<-1

}

m1

#

# Example 2: Doing it again for 3 categories in col\_3

# (Note that this is also an example of multiple if function - see later)

#

for (i in 1:20)

{

mi1<-m1[i,1] # reusable code - 4 times

if (mi1 <= 0.33) m1[i,3]<-1 # assignment to col\_3

else if (mi1 > 0.33 && mi1 <=0.66 ) m1[i,3]<-2 # && means "and" for scalers

else if (mi1 > 0.66) m1[i,3]<-3 # nothing new

}

m1

#

# Notice here that we are indexing the rows one at a time via the

# subscript i. But R is a vectorised language not meant for this.

# In general "for" loops are slow in R - especially nested loops

# and recursion. Special procedures for speeding things up

# 1. resusable code

# 2. making every loop a function - must read about

# 3. using the apply function - must read about

#

#

# Example 3: Doing Example 1 without a loop - the ifelse function

#

# Basic structure

# ifelse( test condition, true expr, false expr)

#

# here is the code for Example 1 - very much faster and no loop

#

m1[,4]<-ifelse ( m1[,1]<= 0.5 , 0, 1)

m1

#

# but this is basically only available for a binary split.

# Note that one can have expressions or function in the true and false

# slots in the ifelse functionm

#

#

# (c) The While loop

#

# For use when you do not know the number of iterations in the loop

# in advance

#

# Example 4 Find a x ~ N(0,1) >= 2 using a function norm2

#

myr<-rnorm(1) # generate a single rnorm(1)

norm2<-function (myr) # name function

{ # open function body

while (myr < 2) # open define & end while loop

{myr<-rnorm(1)} # replace myr until myr >= 2

myr # define myr as the return value of norm2

}

norm2(myr) # display

norm2(rnorm(1)) # display illustrating function substiution

#

# Example 5: calculate a vector of norm2 s using a loop

#

xran<-rnorm(100)

xran

for (i in 1:length(xran))

{

xran[i]<-norm2(xran[i])

}

xran

#

# (d) Break & Next Commands

#

# Example 6: drawing four aces from a pack of 52 cards

#

draw.aces<-function()

 {

 n<-0

 aces<-rep(F,4)

 repeat{

 draw<-sample(1:52, 1, replace=T)

 n<-n+1

 if (draw %% 13 != 1)

 next

 aces[draw %/% 13 +1]<-T

 if (all(aces))

 break

 }

 cat("It took", n, "draws to pick all 4 aces!\n")

 }

#

draw.aces()

#

# Notice several things here:

#(i) how to call a function with no arguments

#(ii) use of sample function - see ?sample

#(iii) use of %% modulus, %/% for integer divide & != for ne

# (iv) use of all function - see ?all

# (v) use & positioning of repeat next and break

# (vi) use of the cat (alogue) to output message - see ?cat

#

# Thats enough to get you started on a crash course

# You will find more in the manuals

# As mentioned earlier For loops and recursion can be

# slow - so we try to avoid these where possible -

# but gnerally they have to be used somewhere

#

#

# 5 Maximum likelihood in R using functions &

# numerical optimisation & the finding hessian

# numerically etc

#

# Recall the hessian is the negative of the observed

# information

#

# we use the non-linear minimisatiom function in R - nlm

# which lives in the library(stats)

#

library (stats) # load library

?nlm # look at help for nlm

#

# Example 1: random sample of 100 Xs ~ N(100, 9)

# ?rnorm

#

myx <- rnorm(100, 100, 3)

hist(myx)

#

# need to find mles by numerical methods using nlm

#

# Step 1 write a function to compute the

# value of the log likelihood function at

# the pair of (mux, sigx)

#

# Since nlm is a minimisation routine we minimise -loglike

# which is the same as maximising loglike

#

loglikenorm<-function(x, myx) # always use x to hold parameter values

{

mu<-x[1]

sig<-x[2]

n<-length(myx)

loglike<- -n\*log(sig\*sqrt(2\*pi))- sum((1/(2\*sig^2))\*(myx-mu)^2) # note use of sum

loglike<- -loglike

}

#

# check second term has length(x) terms because x is a vector

#

# but here is a faster way

loglikenorm2<-function(x, myx)

{

mu<-x[1]

sig<-x[2]

loglike<- sum(-log(sig\*sqrt(2\*pi))- (1/(2\*sig^2))\*(myx-mu)^2) # note use of sum

loglike<- -loglike

}

#

# sum applied to the whole expression gives correct answer

# the constant part (first term) is repeated n times since x

# is vector of length n. This is the way to write log likelihoods

# the sum acts as a big sigma and the expression inside is the

# kernal for the ith component - must that the arithmetic is correct.

#

# check the arithmetic out for say mu=100 & sig=3

#

mu<-100

sig<-3

x=c(mu,sig)

hist(myx)

ans<-loglikenorm(x, myx)

ans

ans2<-loglikenorm2(x, myx)

ans2

#

# define starting values - this supposes

# we do not know the truth (the usual case) and we

# guess initial values and maximise

#

mu<-98

sig<-2

theta=c(mu,sig) # store starting values

hist(myx) # check x ~ N(100,9) on way into nlm

theta # check theta on way in to nlm

#

# Now call nlm

# Pay attention to the order of the arguments

# function, starting values, second parameter myx =>

# things needed in the function to calculate the likelihood,

# hessian =TRUE , print.level 1 => some output.

# Other arguments possible - see ?nlm and examples therein

#

result<-nlm(loglikenorm2, theta , myx=myx, hessian=TRUE, print.level=1)

result # display contents of object result

#

# Extract goodies : note hess from nlm = observed information

# ie already # multiplied by minus one (Americans again!!)

#

mle<-result$estimate #extract and store mles

mle #display mles

hess<-result$hessian #extract and store observed information

hess #display mles

cov<-solve(hess) #invert hess to get cov(mles)

cov #display cov.

stderr<-sqrt(diag(cov))# compute standard errors of mles

stderr

#

# Some functions are more sensitive to starting values

# than others - usually we have to start from different

# values and detrmine whether we converge to the same estimates

# and that these define a global maximum.

#

#

# Example 2: Exponential Survival Model

#

mytime<-rexp(100)

hist(mytime)

delta<-rep(NA,100) # censoring indicator crude

delta[1:80]<-1 # define events

delta[81:100]<-0 # define 20% censored

delta # display

surdat<-matrix(NA,100,4)#

surdat

surdat[,1]<-mytime

surdat[,2]<-delta

surdat[,3]<-rep(1,100)

surdat[,4]<-sqrt(mytime)

surdat

loglikexp<-function(x,surdat)

{

fi<-x

lam<-exp(fi) # keep scalar hazard lam >0

ti<-surdat[,1] # vector of survival time

deltai<-surdat[,2] # vector of censoring indicators

loglike<- -sum(deltai\*fi-lam\*ti)

}

#

theta<-0

resexp<-nlm(loglikexp, theta , surdat=surdat, hessian=TRUE, print.level=1)

resexp # display contents of object result

fihat<-resexp$estimate #extract and store mle

fihat #display mle

hess<-resexp$hessian #extract and store observed information

hess #display mle

cov<-solve(hess) #invert hess to get cov(ml)

cov #display cov.

stderr<-sqrt(diag(cov)) # compute standard error of mle

stderr

#

# Here the mle is of fi ie log(lambda)

# so the mle of lambda is

#

lamhat<-exp(fihat)

lamhat

#

#recall that rexp(100) ~ Exponential(lam=1)

# so lamhat should be near 1. It would have been

# nearer had there been no censoring.

# Question H0: lam =1 v H1: lam ne 1 ?

# Answer test by equivalent hypothesis on fi

# H0: fi = 0 v H1: fi ne 0

# Test statistic is z = (fihat-0)/se(fihat)

#

hist(mytime)

#

#

# Work To do:

# (a)

# modify the Exponential code above to fit

# an exponential regression model with

# lamdai=exp(alpha.xo + beta.xi) etc where

# xo is surdat[,3] .

# x1 is surdat[,4]

#

# (b)

# Write the code for a Poisson Regression model

# and find the mles

#

myfish<-rpois(100, lam=2) #see ?rpois

myfish

hist(myfish)

xi<-sqrt(myfish+1) # use as covariate

xi

xo<-rep(1,100) # constant for intercept

xo

#

# Assume myfish is ~ Poisson (lambda) where

# lambda = exp( xo\*alpha + x1.beta)

#

# There are many other ways constructing

# functions and passing objects

# needed for computing the likelihood

# through to nlm .

#

# This is an area which is very badly

# written up by the R developers -

# seems almost deliberately so at times

#

# You will learn about these topics

# as you study programming in R

#

# 6 Elementary Simulation

#

# Example 1: Central Limit Theorm for X ~ N(0,1)

#

# Suppose need to show asymptotic distribution

# of sample ave and variance for samples

# of size n= 10, n=20, n=30 to persuade

# students

#

# Let the number of repeated samples be

# nsim=1000 so we consider drawing 1000

# SRSs from x~N(0,1) with n=10, n=20 and

# n=30

#

nsim<-1000

#

# We agree to store sample ave & var and later std

# We decide to do each separately and store

# Then reserve storage for n=10

#

sim.n10<-matrix(NA,nsim,3)

attributes(sim.n10)

for (i in 1:nsim)

{

x<-rnorm(10)

sim.n10[i,1]<-mean(x) # this is the mle

sim.n10[i,2]<-var(x) # this is the non-MOM(MY)est

}

sim.n10[,3]<-sqrt(sim.n10[,2])

#

#look at shape of stochastic distributions

#

hist(sim.n10[,1])

hist(sim.n10[,2])

hist(sim.n10[,3])

#

# Find the stochastic means and variances

# of sample ave, variance and std

#

smxbar<-mean(sim.n10[,1])

smvar<-mean(sim.n10[,2])

smstd<-mean(sim.n10[,3])

smean<-c(smxbar,smvar,smstd)

smean

#

# now Stochastic Variances

svxbar<-var(sim.n10[,1])

svvar<-var(sim.n10[,2])

svstd<-var(sim.n10[,3])

svar<-c(svxbar,svvar,svstd)

svar

#

# You can repeat the whole thing

# for n=20 and n=30 or write a

# function to do any sample size

#

#

# 7. Elementary Bootrapping

#

# Say y ~ N(0,1) and we draw 1 sample with

# n =100

#

n<-100

y<-rnorm(n)

hist(y)

#

# Now draw nb=1000 bootstrap samples (with replacement)

# from y and

# look at the distribution of the estimators of

# the mean and variance & std in the boot-strapped samples.

# reserve storage

#

nb<-1000

boot<-matrix(NA,nb,3)

attributes(boot)

for (i in 1:nb)

{

bs<-sample(y,n,replace =T)

boot[i,1]<-mean(bs)

boot[i,2]<-var(bs)

}

boot[,3]<-sqrt(boot[,2])

#

#look at shape of bootstrap distributions

#

hist(boot[,1])

hist(boot[,2])

hist(boot[,3])

#

# get the Bootstrap Dist means and variances

# Find the bootstrap means and variances

# of sample ave, variance and std

#

bmxbar<-mean(boot[,1])

bmvar<-mean(boot[,2])

bmstd<-mean(boot[,3])

bmean<-c(bmxbar,bmvar,bmstd)

bmean

# now the Bootstrap Variances

bvxbar<-var(boot[,1])

bvvar<-var(boot[,2])

bvstd<-var(boot[,3])

bvar<-c(bvxbar,bvvar,bvstd)

bvar

#

# 8. Data and dataframes

#

# We have seen how to create random sample

# data and sequences etc. But we need yo

# know to get real data from other files

# into R

#

# (a) Importing from SPSS

#

# The read.spss function is used. It is in the

# foreign library.

# Usage is: mydata<-read.spss("datafile") = see ?read.spss for more

# where data file is the file reference

#

# Example 1: Read Lung Cancer Data in from spss.sav file

#

lung<-read.spss("c:\\data\\spssdata\\auckland\\lung\_cancer.sav")

#

# Notice the special format of the file reference

# clearly a particular file is specific to your machine

# you should load lung\_cancer.sav into a directory on

# your machine and change the reference above accordingly.

#

class(lung) # list - a list of objects - see ?list

dim(lung) # 855 rows 13 columns

lung<-data.frame(lung) # coerce lung to be data-frame

class(lung) #check class

lung[1:10,] # show 1st 10 rows

attributes(lung)# check attributes

#

# A dataframe - is simply a matrix structure which

# can hold data of mixed modes ie numeric, logical &

# character variables - it is like the spread sheet

# in SPSS or in Excel.

#

hist(SURTIM) # cannot find SURTIM

attach (lung) # make the column names available as globals

# check

hist(SURTIM) # SURTIM now available - see histogram

#

# the attach function - see ?attach - makes the columns of

# the data frame available directly by name - these names

# become "globals" and are available in functions directly

#

myhist<-function()

{hist(SURTIM);hist(AGE)}

myhist()

#

# Even though the function executes in its own frame

# and SURTIM & AGE are not passed as parameters

#

# Attaching in this way can be tricky especially if

# several data frames are present. Also spoils the programme

# logic which forces one to pass parameters into functions

# -this is generally safer - less chance of confusion.

# However dataframes are extremely useful

#

# Note that the dataframe is not physically attached

# but pointers to the columns are set up using the

# names

#

# It is really convenient when there is only one data frame

# present. Then there is no possibility of a clash of

# names and it is arguably the preferred object of

# choice in statistical modelling problems - typically

# because one can refer to the names of the variables

# directly - as in SPSS

#

#

# (b) Importing Data from text (ASCII) file ie a .dat file

#

# The scan function is very versatile for reading vectors and

# matrices of numbers.

#

# Consider test\_heart.dat n rows x 7 cols in directory auckland

# The numbers in each row are separated by spaces

# This is a matrix so we can use scan inside matrix

#

heart<-matrix(scan("c:\\data\\spssdata\\auckland\\test\_heart.dat"), byrow=T,ncol=7)

heart

class(heart)

heart<-data.frame(heart)

class(heart)

attributes(heart) # notice default column names added

attach(heart) # make column names available

hist(X7)

boxplot(X7)

#

# Can read vectors using scan directly without embedding

# in matrix - see ?scan for many details.

#

#

# Other Examples not done include SAS and Mintab - see

# R manual on importing and exporting - simple function

# work the same way as read.spss. May export in many

# similar ways.

#

#~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

#

# In a crash course it is not possible to

# cover everything - but this should give you a

# reasonable intro.

#

# More advanced topics not covered are

# 1. Advanced functions

# 2. Computing on the language

# 3. Use of apply functions

#

#

#

# Good Luck

# Gilbert MacKenzie

# April 3rd, 2006

#?cat