Data Analysis & Graphics Using R, 2^{nd} edn – Solutions to Exercises (December 13, 2006)

Preliminaries

```
> library(DAAG)
```

Exercise 1
Compare the different outputs from `help.search("print")`, `apropos(print)` and `methods(print)`. Look up the help for each of these three functions, and use what you find to explain the different outputs.

`help.search()` searches the documentation for a match in the name, or alias (i.e., an alternative name for a function or other object) or title or keyword.

`apropos()` searches for object or alias names where there is a partial match. For example, try `help.search("str")`. [Note also the function `find()`, which is an alias for `apropos()` in which the default parameters are set to find "simple words".]

`methods(print)` finds all available print methods, i.e., all the different functions that may, depending on the class of object that is to be printed, be called when the generic print function is used.

Now that the number of functions and associated documentation is so extensive, consider limiting the search by using, e.g., `help.search("print", package="base")`, rather than `help.search("print")`

Exercise 2
Identify as many R functions as possible that are specifically designed for manipulations with text strings.

Try `apropos("str")`. Some objects (e.g., `fitdistr` or `structure`) clearly have nothing to do with strings. Look up the help for those that do seem possible string manipulation functions. Look under See Also: to find other related functions that may not have the letters "str" in their names. Try also `apropos("char")`. Once these steps are complete, this should identify most possibilities.

Another recourse may be to type in `help.start()`, and click on Search Engine & Keywords.

Exercise 3
Test whether `strsplit()` is vectorized, i.e., does it accept a vector of character strings as input, then operating in parallel on all elements of the vector?

Try applying `strsplit()` to a vector of character strings. For example:

```
> strsplit(c("eggs\nbacon", "bacon\neggs"), "\n")

[[1]]
[1] "eggs"  "bacon"

[[2]]
[1] "bacon" "eggs"
```

Notice that `strsplit()` does accept a vector of character strings as input, and that it returns one list element for each character string in the vector.

Exercise 4
For the data frame `Cars93`, get the information provided by `summary()` for each level of Type. (Use `split()`.)

First, note the column names:

```
> names(Cars93)
```

```
 [1] "Manufacturer"       "Model"              "Type"
 [4] "Min.Price"          "Price"              "Max.Price"
 [7] "MPG.city"           "MPG.highway"        "AirBags"
[10] "DriveTrain"         "Cylinders"          "EngineSize"
[13] "Horsepower"         "RPM"                "Rev.per.mile"
[16] "Man.trans.avail"    "Fuel.tank.capacity" "Passengers"
[19] "Length"             "Wheelbase"          "Width"
[22] "Turn.circle"        "Rear.seat.room"     "Luggage.room"
[25] "Weight"             "Origin"             "Make"
```

The code that gives the summaries is:

```
lapply(split(Cars93, Cars93$Type), summary)
```

The output runs over many pages. To present only the first two sets of summaries, for the first five columns of the data frame, specify.

```
> lapply(split(Cars93[, 1:5], Cars93$Type), summary)[1:2]
```

Exercise 5
Determine the number of cars, in the data frame `Cars93`, for each Origin and Type.

```
> table(Cars93$Origin, Cars93$Type)
```

```
         Compact Large Midsize Small Sporty Van
  USA          7    11      10     7      8   5
  non-USA      9     0      12    14      6   4
```

Exercise 6
In the data frame `Insurance` (*MASS* package):

(a) determine the number of rows of information for each age category (`Age`) and car type (`Group`);

(b) determine the total number of claims for each age category and car type;

(a)
```
> library(MASS)
> sapply(Insurance, function(x) sum(is.na(x)))
```

```
District    Group      Age  Holders   Claims
       0        0        0        0        0
```

```
> table(Insurance$Group, Insurance$Age)

         <25 25-29 30-35 >35
  <1l      4     4     4   4
  1-1.5l   4     4     4   4
  1.5-2l   4     4     4   4
  >2l      4     4     4   4
```

As the default for `table()` is to omit mention of `NA`'s, it is good practice to make a check, such as included in the statement above, on the number of `NA`'s in each column.

(b)
```
> attach(Insurance)
> tapply(Claims, list(Group, Age), sum)

       <25 25-29 30-35 >35
<1l     67    70    56 346
1-1.5l 105   169   197 979
1.5-2l  46   124   153 540
>2l     11    41    47 200

> detach(Insurance)
```

Exercise 7
Enter the following, and explain the steps that are performed to obtain the result:

```
## Use of split() and sapply(): data frame science (DAAG)
with(science, sapply(split(school, PrivPub),
                     function(x)length(unique(x))))
```

The data frame `science` becomes, for the duration of the calculation

```
sapply(split(school, PrivPub),
                    function(x)length(unique(x)))
```

a "database" where the objects `school` and `PrivPub` can be found.

The statement `split(school, PrivPub)` creates a list that has two elements, one for each of the two levels of `PrivPub`. Each list element holds the codes that identifies the schools. The function `sapply()` operates on each of these list elements in turn. It replaces the vector of codes by a vector of unique codes. The length of that vector is then the number of schools, and of course this is done separately for `Private` and `Public` schools.

Exercise 8
Save the objects in your workspace, into an image (`.RData`) file, with the name **archive.RData**. Then remove all objects from the workspace. Demonstrate how, without loading the image file, it is possible to list the objects that were included in **archive.RData** and to recover a deleted object that is again required.

To save the workspace contents into the file **archive.RData**, type

```
> save.image(file = "archive.RData")
```

We can now type

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

The following will again make available all objects that were in the workspace:

```
> attach("archive.RData", warn.conflicts = FALSE)
```

To see the contents of this "database", type

```
> ls(name = "file:archive.RData")
```

```
 [1] "count.neighbours" "cpu"              "d"                "enoise"
 [5] "generate.data"    "h"                "iperm"            "m"
 [9] "maxys"            "minys"            "n"                "nn"
[13] "nnear"            "ns"               "nx"               "plot.signal"
[17] "sig"              "x"                "xn"               "xsignal"
[21] "xy"               "y"                "yn"               "ysignal"
```

Providing no other databases have been attached in the meantime, an alternative is `ls(pos=2)`.

Type the name of an object that is in the database (choose one that is not too large!) to demonstrate that all such objects are now available.

Note the use of `detach("file:archive.RData")` to detach the database.

Exercise 9
Determine the number of days, according to R, between the following dates:

(a) January 1 in the year 1700, and January 1 in the year 1800

(b) January 1 in the year 1998, and January 1 in the year 2007

```
> as.Date("1/1/1800", "%d/%m/%Y") - as.Date("1/1/1700", "%d/%m/%Y")
```

```
Time difference of 36524 days
```

```
> as.Date("1/1/2007", "%d/%m/%Y") - as.Date("1/1/1998", "%d/%m/%Y")
```

```
Time difference of 3287 days
```

Exercise 10
*The following code concatenates (x, y) data values that are random noise to data pairs that contain a 'signal', randomly permutes the pairs of data values, and finally attempts to reconstruct the signal:

```
### Thanks to Markus Hegland (ANU), who wrote the initial version
##1 Generate the data
# . . . .
# Code is displayed below (with annotations),
# and is therefore omitted here.
# . . . .
##1 End

##2 determine number of neighbors within
#   a distance <= h = 1/sqrt(length(xn))
# . . . .
# Annotated code is shown below
# . . . .
##2 End

##3 Plot data, with reconstructed signal overlaid.
# . . . .
# Annotated code is shown below
# . . . .
##3 End
```

(a) Run the code and observe the graph that results.

(b) Work through the code, and write notes on what each line does.
[The key idea is that points that are part of the signal will, on average, have more near neighbours than points that are noise.]

(c) Split the code into three functions, bracketed respectively between lines that begin `##1`, lines that begin `##2`, and lines that begin `##3`. The first function should take parameters `m` and `n`, and return a list `xy` that holds data that will be used subsequently. The second function should take vectors `xn` and `yn` as parameters, and return values of `nnear`, i.e., for each point, it will give the number of other points that lie within a circle with the point as center and with radius `h`. The third function will take as parameters `x`, `y`, `nnear` and the constant `ns` such that points with more than `ns` near neighbours will be identified as part of the signal. Run the first function, and store the output list of data values in `xy`.

(d) Run the second and third functions with various different settings of `h` and `ns`. Comment on the effect of varying `h`. Comment on the effect of varying `ns`.

(e) Which part of the calculation is most computationally intensive? Which makes the heaviest demands on computer memory?

(f) Suggest ways in which the calculation might be made more efficient.

```
generate data

produce plots
```

(a)

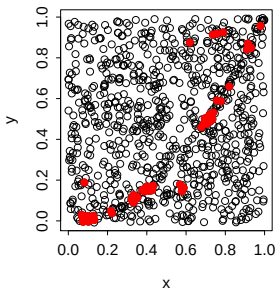

Figure 1: Graph obtained from running the code of Exercise 10

(b)

```
### Thanks to Markus Hegland (ANU), who wrote the initial version
##1 Generate the data
  cat("generate data \n")
  n <- 800      # length of noise vector
  m <- 100      # length of signal vector
## Samples 100 values that will be x-values for the signal
  xsignal <- runif(m)
  sig <- 0.01
  enoise <- rnorm(m)*sig
  ysignal <- xsignal**2+enoise   # y = x^2 + noise
## Determine the range of x- and y-values for the signal
  maxys <- max(ysignal)
  minys <- min(ysignal)
## Precede signal x-values with 800 x-values for points
## that will be entirely noise
  x <- c(runif(n), xsignal)
## Generate y-values for noise; follow with signal values.
## y-values for noise are sampled from a uniform distribution,
## with the same limits as the y-values for the signal.
  y <- c(runif(n)*(maxys-minys)+minys, ysignal)
  # random permutation of the data vectors
## Randomly permute the points, so that points that are signal
## are mixed in with points that are noise.
  iperm <- sample(seq(x))
  x <- x[iperm]
  y <- y[iperm]
  # normalise the data, i.e., scale x & y values to lie between 0 & 1
  xn <- (x - min(x))/(max(x) - min(x))
  yn <- (y - min(y))/(max(y) - min(y))
## The above has generated data, from which to recover the signal.
##1 End

##2 determine number of neighbors within
#   a distance <= h = 1/sqrt(length(xn))
## These distances will be available for all points
  nx <- length(xn)
  # determine distance matrix
## The following is a clever way to calculate
```

```
## sqrt((xi-xj)^2 + (yi-yj)^2), and store the result in the (i,j)
## position of d.
  d  <- sqrt( (matrix(xn, nx, nx) - t(matrix(xn, nx, nx)) )**2 +
              (matrix(yn, nx, nx) - t(matrix(yn, nx, nx)) )**2 )
## Next, we need a threshold, such that most random points are
## will not be closer than this.  Detailed investigation will
## require examination of the distribution of d.  Here we choose
## 1/sqrt(nx); if this does not seem to work, it can be varied.
## Better (and here is a starting point for further exercises),
## the distribution of d can be examined empirically and/or
## theoretically.
  h <- 1/sqrt(nx)
## Count the number of points that lie closer than this threshold
  nnear <- apply(d <= h, 1, sum)
##2 End

##3 Plot data, with reconstructed signal overlaid.
  cat("produce plots \n")
  plot(x, y)
  # plot only the points which have many such neighbors
## ns is another tuning constant.
  ns <- 8
  points(x[nnear > ns], y[nnear > ns], col="red", pch=16)
##3 End
```

(c) Next, the code will be split between three functions:

```
> generate.data <- function(m = 100, n = 800) {
+     xsignal <- runif(m)
+     sig <- 0.01
+     enoise <- rnorm(m) * sig
+     ysignal <- xsignal^2 + enoise
+     maxys <- max(ysignal)
+     minys <- min(ysignal)
+     x <- c(runif(n), xsignal)
+     y <- c(runif(n) * (maxys - minys) + minys, ysignal)
+     iperm <- sample(seq(x))
+     x <- x[iperm]
+     y <- y[iperm]
+     xn <- (x - min(x))/(max(x) - min(x))
+     yn <- (y - min(y))/(max(y) - min(y))
+     list(x = x, y = y, xn = xn, yn = yn)
+ }
> count.neighbours <- function(xn, yn, h = 1/sqrt(length(xn))) {
+     nx <- length(xn)
+     d <- sqrt((matrix(xn, nx, nx) - t(matrix(xn, nx, nx)))^2 +
+         (matrix(yn, nx, nx) - t(matrix(yn, nx, nx)))^2)
+     nnear <- apply(d <= h, 1, sum)
+     nnear
+ }
> plot.signal <- function(x, y, nnear, ns = 8) {
+     plot(x, y)
+     ns <- 8
```

```
+     points(x[nnear > ns], y[nnear > ns], col = "red", pch = 16)
+ }
```

Here then is a sequence of calls:

```
> xy <- generate.data(m = 100, n = 800)
> nnear <- count.neighbours(xn = xy[["xn"]], yn = xy[["yn"]])
> plot.signal(x = xy[["x"]], y = xy[["y"]], nnear = nnear, ns = 8)
```

(d) In an initial simulation, the range of values of `nnear`, obtained from `range(nnear)`, was from 1 to 13. Hence, we will try setting `nnear = 6` and `nnear=10`. For `ns` we will try `2/sqrt(length(xn))` and `0.5/sqrt(length(xn))`.

```
> par(mfrow = c(2, 2))
> nx <- length(xy[["xn"]])
> nnear <- count.neighbours(xn = xy[["xn"]], yn = xy[["yn"]], h = sqrt(0.5/nx))
> plot.signal(x = xy[["x"]], y = xy[["y"]], nnear = nnear, ns = 6)
> title(main = "h=sqrt(0.5/nx); ns=6")
> plot.signal(x = xy[["x"]], y = xy[["y"]], nnear = nnear, ns = 10)
> title(main = "h=sqrt(0.5/nx); ns=10")
> nnear <- count.neighbours(xn = xy[["xn"]], yn = xy[["yn"]], h = sqrt(2/nx))
> plot.signal(x = xy[["x"]], y = xy[["y"]], nnear = nnear, ns = 6)
> title(main = "h=sqrt(2/nx); ns=6")
> plot.signal(x = xy[["x"]], y = xy[["y"]], nnear = nnear, ns = 10)
> title(main = "h=sqrt(2/nx); ns=10")
```

The result is sensitive to the choice of `h`. Therefore, repeat the exercise with `h=sqrt(0.75/nx)` and `h=sqrt(1/nx)` The result is relatively insensitive to variation in `ns`.

(e) The most computationally intensive part of the calculations is the determination of the distances. This is done for all `nx^2` pairs (x,y), though actually we only need the `nx*(nx+1)/2` points in the upper triangle of the matrix. This makes, if `nx` is large, heavy demands on computer memory. Calculation of `nnear`, as done above, requires `nx` comparisons for each point, i.e., a total of `nx^2` comparisons, with the result stored in a vector of length `nx`. These should be much cheaper than multiplications.

We now examine the costs in an actual machine run.

```
> system.time(xy <- generate.data(m = 100, n = 800))

[1] 0.002 0.000 0.002 0.000 0.000

> system.time(nnear <- count.neighbours(xn = xy[["xn"]], yn = xy[["yn"]]))

[1] 0.341 0.097 0.439 0.000 0.000

> system.time(plot.signal(x = xy[["x"]], y = xy[["y"]], nnear = nnear,
+     ns = 8))

[1] 0.004 0.000 0.012 0.000 0.000
```

The function `count.neighbours()` has taken most of the time, on my system 3.10 seconds. We now break this down further.

```
> xn <- xy[["xn"]]
> yn <- xy[["yn"]]
> nx <- length(xn)
> h <- 1/sqrt(nx)
> system.time(d <- sqrt((matrix(xn, nx, nx) - t(matrix(xn, nx,
+     nx)))^2 + (matrix(yn, nx, nx) - t(matrix(yn, nx, nx)))^2))

[1] 0.167 0.091 0.258 0.000 0.000

> system.time(nnear <- apply(d <= h, 1, sum))

[1] 0.175 0.008 0.182 0.000 0.000
```

Calculation of `d` took 1.62 seconds, whereas calculation of `nnear` tool 0.67 seconds.

(f) The focus should be on those calculations that are computationally intensive, i.e., the calculation of the distances. There are `nx*(nx-1)/2` distances that need be calculated, where the code has calculated `nx^2` distances, i.e. the distance from point 2 to point 1 as well as the distance from point 1 to point 2.

Exercise 11
This question has been reworded
Try the following, for a range of values of `n` between, e.g., 2×10^5 and 10^7. (On systems that are unable to cope with such large numbers of values, adjust the range of numbers of values accordingly.)

```
n <- 10000; system.time(sd(rnorm(n)))
```

The first output number is the user cpu time, while the third output number is the elapsed time. Plot each of these numbers, separately, against `n`. Comment on the graphs. Is the elapsed time roughly linear with `n`? Try the computations both for an otherwise empty workspace, and with large data objects (e.g., with 10^7 or more elements) in the workspace.

On a 1.2MHz Macintosh G4 PowerBook with half a gigabyte of memory, results were:

```
> nn <- 2e+06 * (1:5)
> cpu <- numeric(5)
> cpu[1] <- system.time(sd(rnorm(n = nn[1])))[1]
> cpu[2] <- system.time(sd(rnorm(n = nn[2])))[1]
> cpu[3] <- system.time(sd(rnorm(n = nn[3])))[1]
> cpu[4] <- system.time(sd(rnorm(n = nn[4])))[1]
> cpu[5] <- system.time(sd(rnorm(n = nn[5])))[1]
```

Here is a graph:

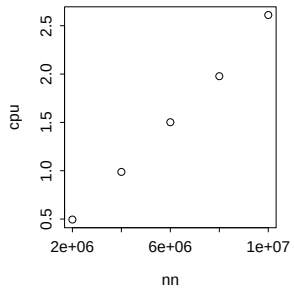

Figure 2: Cpu time, versus number of elements.

On my system, the response was remarkably linear with time. The increase in time with increasing values of `nn` reduced slightly as `nn` increased.