Appendix A

R Quick Start

Here we present a quick introduction to the R data/statistical programming language. Further learning resources are listed at `http://heather.cs.ucdavis.edu/~/matloff/r.html`.

R syntax is similar to that of C. It is object-oriented (in the sense of encapsulation, polymorphism and everything being an object) and is a functional language (i.e. almost no side effects, every action is a function call, etc.).

A.1 Correspondences

aspect	C/C++	R
assignment	=	`<-` (or =)
array terminology	array	vector, matrix, array
subscripts	start at 0	start at 1
array notation	m[2][3]	m[2,3]
2-D array storage	row-major order	column-major order
mixed container	struct, members accessed by .	list, members acessed by $ or [[ ]]
return mechanism	return	return() or last value computed
primitive types	int, float, double, char, bool	integer, float, double, character, logical
logical values	true, false	TRUE, FALSE (abbreviated T, F)
mechanism for combining modules	include, link	library()
run method	batch	interactive, batch

A.2 Starting R

To invoke R, just type "R" into a terminal window. On a Windows machine, you probably have an R icon to click.

If you prefer to run from an IDE, you may wish to consider ESS for Emacs, StatET for Eclipse or RStudio, all open source. ESS is the favorite among the "hard core coder" types, while the colorful, easy-to-use, RStudio is a big general crowd pleaser. If you are already an Eclipse user, StatET will be just what you need.

R is normally run in interactive mode, with > as the prompt. Among other things, that makes it easy to try little experiments to learn from; remember my slogan, "When in doubt, try it out!"

A.3 First Sample Programming Session

Below is a commented R session, to introduce the concepts. I had a text editor open in another window, constantly changing my code, then loading it via R's **source()** command. The original contents of the file **odd.R** were:

```
oddcount <- function(x) {
   k <- 0  # assign 0 to k
   for (n in x) {
      if (n %% 2 == 1) k <- k+1  # %% is the modulo operator
   }
   return(k)
}
```

By the way, we could have written that last statement as simply

```
   k
```

because the last computed value of an R function is returned automatically.

The R session is shown below. You may wish to type it yourself as you go along, trying little experiments of your own along the way.[1]

```
> source("odd.R")  # load code from the given file
> ls()  # what objects do we have?
[1] "oddcount"
> # what kind of object is oddcount (well, we already know)?
```

[1] The source code for this file is at `http://heather.cs.ucdavis.edu/~matloff/MiscPLN/R5MinIntro.tex`. You can download the file, and copy/paste the text from there.

```
> class(oddcount)
[1] "function"
> # while in interactive mode, and not inside a function, can print
> # any object by typing its name; otherwise use print(), e.g. print(x+y)
> oddcount  # a function is an object, so can print it
function(x)  {
   k <- 0  # assign 0 to k
   for (n in x)  {
      if (n %% 2 == 1) k <- k+1  # %% is the modulo operator
   }
   return(k)
}

> # let's test oddcount(), but look at some properties of vectors first
> y <- c(5,12,13,8,88)  # c() is the concatenate function
> y
[1]  5 12 13  8 88
> y[2]  # R subscripts begin at 1, not 0
[1] 12
> y[2:4]  # extract elements 2, 3 and 4 of y
[1] 12 13  8
> y[c(1,3:5)]  # elements 1, 3, 4 and 5
[1]  5 13  8 88
> oddcount(y)  # should report 2 odd numbers
[1] 2

> # change code (in the other window) to vectorize the count operation,
> # for much faster execution
> source("odd.R")
> oddcount
function(x)  {
   x1 <- (x %% 2 == 1)  # x1 now a vector of TRUEs and FALSEs
   x2 <- x[x1]  # x2 now has the elements of x that were TRUE in x1
   return(length(x2))
}

> # try it on subset of y, elements 2 through 3
> oddcount(y[2:3])
[1] 1
> # try it on subset of y, elements 2, 4 and 5
```

```
> oddcount(y[c(2,4,5)])
[1] 0

> # further compactify the code
> source("odd.R")
> oddcount
function(x) {
   length(x[x %% 2 == 1]) # last value computed is auto returned
}
> oddcount(y) # test it
[1] 2

# and even more compactification, making use of the fact that TRUE and
# FALSE are treated as 1 and 0
> oddcount <- function(x) sum(x %% 2 == 1)
# make sure you understand the steps that that involves: x is a vector,
# and thus x %% 2 is a new vector, the result of applying the mod 2
# operation to every element of x; then x %% 2 == 1 applies the == 1
# operation to each element of that result, yielding a new vector of TRUE
# and FALSE values; sum() then adds them (as 1s and 0s)

# we can also determine which elements are odd
> which(y %% 2 == 1)
[1] 1 3

> # now have ftn return odd count AND the odd numbers themselves, using
> # the R list type
> source("odd.R")
> oddcount
function(x) {
   x1 <- x[x %% 2 == 1]
   return(list(odds=x1, numodds=length(x1)))
}
> # R's list type can contain any type; components delineated by $
> oddcount(y)
$odds
[1]  5 13

$numodds
[1] 2
```

```

> ocy <- oddcount(y)  # save the output in ocy, which will be a list
> ocy
$odds
[1]  5 13

$numodds
[1] 2

> ocy$odds
[1]  5 13
> ocy[[1]]  # can get list elements using [[ ]] instead of $
[1]  5 13
> ocy[[2]]
[1] 2
```

Note that the function of the R function **function()** is to produce functions! Thus assignment is used. For example, here is what **odd.R** looked like at the end of the above session:

```
oddcount <- function(x)  {
   x1 <- x[x %% 2 == 1]
   return(list(odds=x1, numodds=length(x1)))
}
```

We created some code, and then used **function()** to create a function object, which we assigned to **oddcount**.

Note that we eventually **vectorized** our function **oddcount()**. This means taking advantage of the vector-based, functional language nature of R, exploiting R's built-in functions instead of loops. This changes the venue from interpreted R to C level, with a potentially large increase in speed. For example:

```
> x <- runif(1000000)  # 1000000 random numbers from the interval (0,1)
> system.time(sum(x))
   user  system elapsed
  0.008   0.000   0.006
> system.time({s <- 0; for (i in 1:1000000) s <- s + x[i]})
   user  system elapsed
  2.776   0.004   2.859
```

A.4 Second Sample Programming Session

A matrix is a special case of a vector, with added class attributes, the numbers of rows and columns.

```
> # "rowbind() function combines rows of matrices; there's a cbind() too
> m1 <- rbind(1:2,c(5,8))
> m1
     [,1] [,2]
[1,]    1    2
[2,]    5    8
> rbind(m1,c(6,-1))
     [,1] [,2]
[1,]    1    2
[2,]    5    8
[3,]    6   -1

> # form matrix from 1,2,3,4,5,6, in 2 rows; R uses column-major storage
> m2 <- matrix(1:6,nrow=2)
> m2
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
> ncol(m2)
[1] 3
> nrow(m2)
[1] 2
> m2[2,3]  # extract element in row 2, col 3
[1] 6
# get submatrix of m2, cols 2 and 3, any row
> m3 <- m2[,2:3]
> m3
     [,1] [,2]
[1,]    3    5
[2,]    4    6

> m1 * m3  # elementwise multiplication
     [,1] [,2]
[1,]    3   10
[2,]   20   48
> 2.5 * m3  # scalar multiplication (but see below)
     [,1] [,2]
```

```
[1,]    7.5  12.5
[2,]   10.0  15.0
> m1 %*% m3  # linear algebra matrix multiplication
      [,1] [,2]
[1,]    11   17
[2,]    47   73

> # matrices are special cases of vectors, so can treat them as vectors
> sum(m1)
[1] 16
> ifelse(m2 %%3 == 1,0,m2) # (see below)
      [,1] [,2] [,3]
[1,]     0    3    5
[2,]     2    0    6
```

The "scalar multiplication" above is not quite what you may think, even though the result may be. Here's why:

In R, scalars don't really exist; they are just one-element vectors. However, R usually uses **recycling**, i.e. replication, to make vector sizes match. In the example above in which we evaluated the express `2.5 * m3`, the number 2.5 was recycled to the matrix

$$\begin{pmatrix} 2.5 & 2.5 \\ 2.5 & 2.5 \end{pmatrix} \tag{A.1}$$

in order to conform with **m3** for (elementwise) multiplication.

The **ifelse()** function is another example of vectorization. Its call has the form

```
ifelse(boolean vectorexpression1, vectorexpression2, vectorexpression3)
```

All three vector expressions must be the same length, though R will lengthen some via recycling. The action will be to return a vector of the same length (and if matrices are involved, then the result also has the same shape). Each element of the result will be set to its corresponding element in **vectorexpression2** or **vectorexpression3**, depending on whether the corresponding element in **vectorexpression1** is TRUE or FALSE.

In our example above,

```
> ifelse(m2 %%3 == 1,0,m2) # (see below)
```

the expression `m2 %%3 == 1` evaluated to the boolean matrix

$$\begin{pmatrix} T & F & F \\ F & T & F \end{pmatrix} \tag{A.2}$$

(TRUE and FALSE may be abbreviated to T and F.)

The 0 was recycled to the matrix

$$\begin{pmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{A.3}$$

while **vectorexpression3**, **m2**, evaluated to itself.

A.5 Third Sample Programming Session

This time, we focus on vectors and matrices.

```
> m <- rbind(1:3,c(5,12,13))  # "row bind," combine rows
> m
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    5   12   13
> t(m)  # transpose
     [,1] [,2]
[1,]    1    5
[2,]    2   12
[3,]    3   13
> ma <- m[,1:2]
> ma
     [,1] [,2]
[1,]    1    2
[2,]    5   12
> rep(1,2)  # "repeat," make multiple copies
[1] 1 1
> ma %*% rep(1,2)  # matrix multiply
     [,1]
[1,]    3
[2,]   17
> solve(ma,c(3,17))  # solve linear system
```

```
[1] 1 1
> solve(ma)  # matrix inverse
     [,1] [,2]
[1,]  6.0 -1.0
[2,] -2.5  0.5
```

A.6 The R List Type

The R **list** type is, after vectors, the most important R construct. A list is like a vector, except that the components are generally of mixed types.

A.6.1 The Basics

Here is example usage:

```
> g <- list(x = 4:6, s = "abc")
> g
$x
[1] 4 5 6

$s
[1] "abc"

> g$x  # can reference by component name
[1] 4 5 6
> g$s
[1] "abc"
> g[[1]]  # can reference by index, but note double brackets
[1] 4 5 6
> g[[2]]
[1] "abc"
> for (i in 1:length(g)) print(g[[i]])
[1] 4 5 6
[1] "abc"
```

A.6.2 The Reduce() Function

One often needs to combine elements of a list in some way. One approach to this is to use **Reduce()**:

```
> x <- list(4:6,c(1,6,8))
> x
[[1]]
[1] 4 5 6

[[2]]
[1] 1 6 8

> sum(x)
Error in sum(x) : invalid 'type' (list) of argument
> Reduce(sum,x)
[1] 30
```

Here **Reduce()** cumulatively applied R's **sum()** to **x**. Of course, you can use it with functions you write yourself too.

Continuing the above example:

```
> Reduce(c,x)
[1] 4 5 6 1 6 8
```

A.6.3 S3 Classes

R is an object-oriented (and functional) language. It features two types of classes, S3 and S4. I'll introduce S3 here.

An S3 object is simply a list, with a class name added as an *attribute*:

```
> j <- list(name="Joe", salary=55000, union=T)
> class(j) <- "employee"
> m <- list(name="Joe", salary=55000, union=F)
> class(m) <- "employee"
```

So now we have two objects of a class we've chosen to name **"employee"**. Note the quotation marks.

We can write class *generic functions*:

```
> print.employee <- function(wrkr) {
+     cat(wrkr$name,"\n")
+     cat("salary",wrkr$salary,"\n")
+     cat("union member",wrkr$union,"\n")
+ }
```

```
> print(j)
Joe
salary 55000
union member TRUE
> j
Joe
salary 55000
union member TRUE
```

What just happened? Well, **print()** in R is a *generic* function, meaning that it is just a placeholder for a function specific to a given class. When we printed **j** above, the R interpreter searched for a function **print.employee()**, which we had indeed created, and that is what was executed. Lacking this, R would have used the print function for R lists, as before:

```
> rm(print.employee)  # remove the function, to see what happens with print
> j
$name
[1] "Joe"

$salary
[1] 55000

$union
[1] TRUE

attr(,"class")
[1] "employee"
```

A.6.4 Handy Utilities

R functions written by others, e.g. in base R or in the CRAN repository for user-contributed code, often return values which are class objects. It is common, for instance, to have lists within lists. In many cases these objects are quite intricate, and not thoroughly documented. In order to explore the contents of an object—even one you write yourself—here are some handy utilities:

- **names()**: Returns the names of a list.
- **str()**: Shows the first few elements of each component.
- **summary()**: General function. The author of a class **x** can write a version specific to **x**, i.e. **summary.x()**, to print out the important parts; otherwise the default will print some

bare-bones information.

For example:

```
> z <- list(a = runif(50), b = list(u=sample(1:100,25), v="blue sky"))
> z
$a
 [1] 0.301676229 0.679918518 0.208713522 0.510032893 0.405027042
0.412388038
 [7] 0.900498062 0.119936222 0.154996457 0.251126218 0.928304164
0.979945937
[13] 0.902377363 0.941813898 0.027964137 0.992137908 0.207571134
0.049504986
[19] 0.092011899 0.564024424 0.247162004 0.730086786 0.530251779
0.562163986
[25] 0.360718988 0.392522242 0.830468427 0.883086752 0.009853107
0.148819125
[31] 0.381143870 0.027740959 0.173798926 0.338813042 0.371025885
0.417984331
[37] 0.777219084 0.588650413 0.916212011 0.181104510 0.377617399
0.856198893
[43] 0.629269146 0.921698394 0.878412398 0.771662408 0.595483477
0.940457376
[49] 0.228829858 0.700500359

$b
$b$u
 [1] 33 67 32 76 29  3 42 54 97 41 57 87 36 92 81 31 78 12 85 73 26 44
86 40 43

$b$v
[1] "blue sky"
> names(z)
[1] "a" "b"
> str(z)
List of 2
 $ a: num [1:50] 0.302 0.68 0.209 0.51 0.405 ...
 $ b:List of 2
  ..$ u: int [1:25] 33 67 32 76 29 3 42 54 97 41 ...
  ..$ v: chr "blue sky"
> names(z$b)
```

```
[1] "u" "v"
> summary(z)
  Length Class  Mode
a 50     -none- numeric
b  2     -none- list
```

A.7 Data Frames

Another workhorse in R is the *data frame*. A data frame works in many ways like a matrix, but differs from a matrix in that it can mix data of different modes. One column may consist of integers, while another can consist of character strings and so on. Within a column, though, all elements must be of the same mode, and all columns must have the same length.

We might have a 4-column data frame on people, for instance, with columns for height, weight, age and name—3 numeric columns and 1 character string column.

Technically, a data frame is an R list, with one list element per column; each column is a vector. Thus columns can be referred to by name, using the **$** symbol as with all lists, or by column number, as with matrices. The matrix **a[i,j]** notation for the element of **a** in row **i**, column **j**, applies to data frames. So do the **rbind()** and **cbind()** functions, and various other matrix operations, such as filtering.

Here is an example using the dataset **airquality**, built in to R for illustration purposes. You can learn about the data through R's online help, i.e.

```
> ?airquality
```

Let's try a few operations:

```
> names(airquality)
[1] "Ozone"   "Solar.R" "Wind"    "Temp"    "Month"   "Day"
> head(airquality)  # look at the first few rows
  Ozone Solar.R Wind Temp Month Day
1    41     190  7.4   67     5   1
2    36     118  8.0   72     5   2
3    12     149 12.6   74     5   3
4    18     313 11.5   62     5   4
5    NA      NA 14.3   56     5   5
6    28      NA 14.9   66     5   6
> airquality[5,3]  # temp on the 5th day
[1] 14.3
```

```
> airquality$Wind[3]   # same
[1] 12.6
> nrow(airquality)  # number of days observed
[1] 153
> ncol(airquality)  # number of variables
[1] 6
> airquality$Celsius <- (5/9) * (airquality[,4] - 32)  # new variable
> names(airquality)
[1] "Ozone"    "Solar.R"  "Wind"     "Temp"     "Month"    "Day"      "Celsius"
> ncol(airquality)
[1] 7
> airquality[1:3,]
  Ozone Solar.R Wind Temp Month Day  Celsius
1    41     190  7.4   67     5   1 19.44444
2    36     118  8.0   72     5   2 22.22222
3    12     149 12.6   74     5   3 23.33333
> aqjune <- airquality[airquality$Month == 6,]  # filter op
> nrow(aqjune)
[1] 30
> mean(aqjune$Temp)
[1] 79.1
> write.table(aqjune,"AQJune")  # write data frame to file
> aqj <- read.table("AQJune",header=T)  # read it in
```

A.8 Graphics

R excels at graphics, offering a rich set of capabilities, from beginning to advanced. In addition to the functions in base R, extensive graphics packages are available, such as **lattice** and **ggplot2**.

One point of confusion for beginniners involves saving an R graph that is currently displayed on the screen to a file. Here is a function for this, which I include in my R startup file, **.Rprofile**, in my home directory:

```
pr2file
function (filename)
{
    origdev <- dev.cur()
    parts <- strsplit(filename, ".", fixed = TRUE)
    nparts <- length(parts[[1]])
    suff <- parts[[1]][nparts]
```

```
    if (suff == "pdf") {
        pdf(filename)
    }
    else if (suff == "png") {
        png(filename)
    }
    else jpeg(filename)
    devnum <- dev.cur()
    dev.set(origdev)
    dev.copy(which = devnum)
    dev.set(devnum)
    dev.off()
    dev.set(origdev)
}
```

The code, which I won't go into here, mostly involves manipulation of various R graphics devices. I've set it up so that you can save to a file of type either PDF, PNG or JPEG, implied by the file name you give.

A.9 Other Sources for Learning R

There are tons of resources for R on the Web. You may wish to start with the links at `http://heather.cs.ucdavis.edu/~matloff/r.html`.

A.10 Online Help

R's **help()** function, which can be invoked also with a question mark, gives short descriptions of the R functions. For example, typing

```
> ?rep
```

will give you a description of R's **rep()** function.

An especially nice feature of R is its **example()** function, which gives nice examples of whatever function you wish to query. For instance, typing

```
> example(wireframe())
```

will show examples—R code and resulting pictures—of **wireframe()**, one of R's 3-dimensional graphics functions.

A.11 Debugging in R

The internal debugging tool in R, **debug()**, is usable but rather primitive. Here are some alternatives:

- The RStudio IDE has a built-in debugging tool.
- The StatET IDE for R on Eclipse has a nice debugging tool. Works on all major platforms, but can be tricky to install.
- My own debugging tool, **debugR**, is extensive and easy to install, but for the time being is limited to Linux, Mac and other Unix-family systems. See *http://heather.cs.ucdavis.edu/debugR.html*.

A.12 Complex Numbers

If you have need for complex numbers, R does handle them. Here is a sample of use of the main functions of interest:

```
> za <- complex(real=2,imaginary=3.5)
> za
[1] 2+3.5i
> zb <- complex(real=1,imaginary=-5)
> zb
[1] 1-5i
> za * zb
[1] 19.5-6.5i
> Re(za)
[1] 2
> Im(za)
[1] 3.5
> za^2
[1] -8.25+14i
> abs(za)
[1] 4.031129
> exp(complex(real=0,imaginary=pi/4))
[1] 0.7071068+0.7071068i
> cos(pi/4)
[1] 0.7071068
> sin(pi/4)
[1] 0.7071068
```

Note that operations with complex-valued vectors and matrices work as usual; there are no special complex functions.