Important: Note distinction between figures in the book and figures in the online chapters

Table of contents

Online Appendix 3.A: R code demonstration of sensitivity/specificity concepts

In the folder corresponding to this chapter, open the **software** folder, click on the file **software.Rproj**, which launches **RStudio** in the correct folder. [You do not have to worry about setting the default directory, path, etc., as the project file usually takes care of all these details.] Under the **Files** menu find the file **mainBinaryRatings.R** and open it (i.e., click on it. A listing of the file follows:

Appendix 3.A.1: Code Listing

```
# mainBinaryRatings.R
rm( list = ls() ) # clear all variables
mu <- 1.5;zeta <- mu/2
seed <- 100;K1 <- 9;K2 <- 11
set.seed(seed)
z1 <- rnorm(K1)
z2 <- rnorm(K2) + mu
nTN <- length(z1[z1 < zeta])
nTP <- length(z2[z2 >= zeta])
Sp <- nTN/K1;Se <- nTP/K2
cat("seed = ", seed, ", K1 = ", K1, ", K2 = ", K2,
    "Specificity = ", Sp, ", Sensitivity = ", Se, "\n")
```

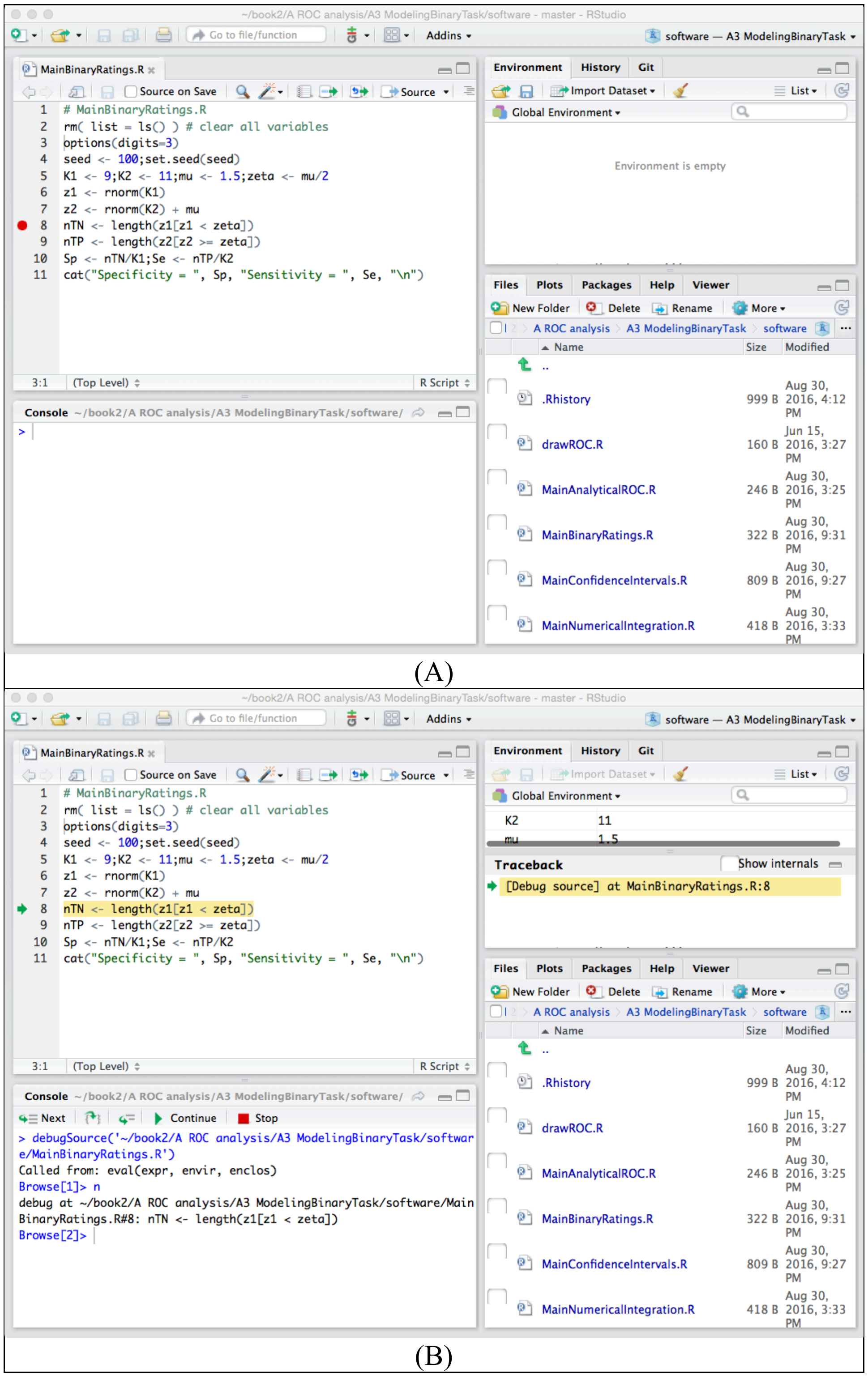
Line 4 sets the seed of the random number generator to 100: this causes the random number generator to yield the same sequence of "random" numbers every time it is run. This is useful during initial code development and for showing the various steps of the example (if **`seed <- NULL`** the random numbers would be different every time, making it harder for me, from a pedagogical point of view, to illustrate the steps). Line 5 initializes variables **`K1`** and **`K2`**, which represent K_1, the number of non-diseased cases and K_2, the number of diseased cases, respectively. In this example 9 non-diseased and 11 diseased cases are simulated. Line 5 also initializes the parameter **`mu <- 1.5`** (**`mu`** corresponds to the μ parameter of the simulation model). Finally, this line initializes **`zeta`**, which corresponds to the threshold ζ defined previously, to **`mu/2`**, i.e., halfway between the means of the two distributions defining the binormal model. Later one can experiment with other values. [Note that multiple statements can be put on a single line as long as semi-colons separate them. The author prefers the "vertical length" of the program to be short, a personal preference that gives the author a better perspective of the code.]

Line 6 calls the built-in function **`rnorm()`** – for random sample(s) from a normal distribution - with argument **`K1`**, which yields **`K1`** (9 in our example) samples from a unit normal distribution $N(0,1)$. *Arguments to a function are always comma separated and contained within enclosing parentheses*. The samples are assigned to the variable **`z1`** (for z-samples for non-diseased cases; $t = 1$). The corresponding samples for the diseased cases, line 7, denoted **`z2`**, were obtained using **`rnorm(K2) + mu`**. [Alternatively one could have used **`rnorm(K2, mean = mu)`**, which cause the value **`mu`** to override the default value - zero - of the mean of the normal distribution.] Since **`mu`** was initialized to 1.5, this line yields 11 samples from a normal distribution with mean zero and unit variance and adds 1.5 to all samples (if one wishes to sample from a distribution with a different variance, for example "3", one needs to also insert the standard deviation argument, e.g., **`sd = sqrt(3)`**, in the call to **`rnorm()`**). The modifications to the default values can be inserted, separated by commas, *in any order*, but the names **`mean`** and **`sd`** must match; try **`typing rnorm(K1, mean1 = 0)`** in the console window, one should see the following error message:

Appendix 3.A.2: Code Snippet

```
> rnorm(K1, mean1 = 0)
Error in rnorm(K1, mean1 = 0) : unused argument (mean1 = 0)
```

This is **R**'s way of telling one that one has supplied an argument **`mean1`** that it does not understand. Click on the grey area of the source code window to the left of the line label 8. This inserts a **`break point`** at line 8 and it show up visually as a red dot to the left of line number 8, Online Fig. 3.1 (A). A break point is a debugging aid that causes program execution to stop at the first encountered break point. **`Source`** the file, Online Fig. 3.1 (B). Tip: to clear a messy **`Console`** window, **`Ctrl-L`**.



(A)

```
1   # MainBinaryRatings.R
2   rm( list = ls() ) # clear all variables
3   options(digits=3)
4   seed <- 100;set.seed(seed)
5   K1 <- 9;K2 <- 11;mu <- 1.5;zeta <- mu/2
6   z1 <- rnorm(K1)
7   z2 <- rnorm(K2) + mu
8   nTN <- length(z1[z1 < zeta])
9   nTP <- length(z2[z2 >= zeta])
10  Sp <- nTN/K1;Se <- nTP/K2
11  cat("Specificity = ", Sp, "Sensitivity = ", Se, "\n")
```

Environment is empty

(B)

```
K2    11
mu    1.5
```

```
[Debug source] at MainBinaryRatings.R:8
```

```
> debugSource('~/book2/A ROC analysis/A3 ModelingBinaryTask/software/MainBinaryRatings.R')
Called from: eval(expr, envir, enclos)
Browse[1]> n
debug at ~/book2/A ROC analysis/A3 ModelingBinaryTask/software/MainBinaryRatings.R#8: nTN <- length(z1[z1 < zeta])
Browse[2]>
```

~/book2/A ROC analysis/A3 ModelingBinaryTask/software - master - RStudio
MainBinaryRatings.R
Source on Save
Source
Environment
History
Git
Global Environment
Traceback
Show internals
Console ~/book2/A ROC analysis/A3 ModelingBinaryTask/software/
Next
Continue
Stop
Files
Plots
Packages
Help
Viewer
.Rhistory 999 B Aug 30, 2016, 4:12 PM
drawROC.R 160 B Jun 15, 2016, 3:27 PM
MainAnalyticalROC.R 246 B Aug 30, 2016, 3:25 PM
MainBinaryRatings.R 322 B Aug 30, 2016, 9:31 PM
MainConfidenceIntervals.R 809 B Aug 30, 2016, 9:27 PM
MainNumericalIntegration.R 418 B Aug 30, 2016, 3:33 PM

Online Fig. 3.1: Plot A: Screen shot of **RStudio** window after inserting break point at line 8; note the red dot indicating the position of the break point. Plot B: Screen shot after clicking on **Source**; program has halted at line 9 and current values of variables are displayed in the **Environment** window.

Notice the appearance of appropriate values in the **Environment** window. *Line 8 looks complicated but it is important to understand it.* For convenience it is reproduced below:

```
nTN <- length(z1[z1 < zeta])
```

One always starts by trying to understand the right hand side of the assignment operator **`<-`** *and one always work from inside out.* The innermost statement in the "mess" on the right hand side is **`z1 < zeta`**: highlight it (and nothing else!) and click on the **`Run`** button, or simply type **`z1 < zeta`** in the **`Console`** window. In either case one should see the following (if not, check the seed values, numbers of cases etc. to exactly match the above code listing):

```
[1]  TRUE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
```

There are 9 elements in this output corresponding to the 9 non-diseased cases. Now highlight **`z1`** (and nothing else!) and click on the **`Run`** button. You should see the following:

```
[1] -0.5022  0.1315 -0.0789  0.8868  0.1170  0.3186 -0.5818  0.7145 -0.8253
```

Again, there are 9 elements in this output. The starting **`[1]`** is **`R`**'s way of saying that the 1st element of the 9-dimensional array is -0.5022, the 2nd element is 0.1315, etc. Since the value of the first element of the array (i.e., -0.5022) is less than **`zeta`** (**`zeta = mu/2`** = 0.75), the 1st element of the 9-dimensional logical array **`z1 < zeta`** is **`TRUE`**. The same is true for all other elements of the array with one exception: the 4th element of the **`z1`** array is 0.8868, and since this is greater than **`zeta`**, the corresponding element of **`z1 < zeta`** is **`FALSE`**. [One can now appreciate why the author took the precaution of "freezing" the seed variable at 100 at line 4. Otherwise randomness would make it impossible for the author to explain the code.]

Working outwards, one sees that **`z1 < zeta`** is surrounded by square brackets. *Square brackets* **`[]`** *are used to index (or subscript) the variable to the left of the opening square bracket*, which happens to be **`z1`**. So **`z1[1]`** would be -0.5022, the 2nd element **`z1[2]`** would be 0.1315 and so on (unlike C++, IDL and some other languages, arrays in **`R`** are indexed starting with unity, not zero). Interestingly, **`R`** allows the indexing variable to be a logical (i.e., **`TRUE`**/**`FALSE`**) array. **`R`** keeps those elements whose index is **`TRUE`** and discards those elements whose index is **`FALSE`**. With this knowledge under one's belt, highlight the variable **`z1[z1 < zeta]`** and click on the **`Run`** button; one should see the following:

```
> z1[z1 < zeta]
[1] -0.5022  0.1315 -0.0789  0.1170  0.3186 -0.5818  0.7145 -0.8253
```

This time there are only 8 elements in this output corresponding to the 8 non-diseased cases which yielded TRUE in the logical comparison. Notice that the value **`0.8868`**, which did not satisfy the inequality **`z1 < zeta`**, has been discarded. The resulting array **`z1[z1 < zeta]`** is 8-dimensional. In summary, *the construct **`z1[z1 < zeta]`** selects all elements of the **`z1`** array that satisfy the inequality **`z1 < zeta`**.*

Working outwards, one sees that **`z1[z1 < zeta]`** is *itself* enclosed in parentheses: **`()`**. In **R** parentheses are used to pass the pass an argument (or multiple arguments, separated by commas) to the **`function`** whose name occurs to the left of the opening parenthesis. This happens to be the **R** function **`length()`**, which as expected yields the length of the enclose array, 8 in our case. Highlight **`length(z1[z1 < zeta])`** and click on **`Run`**:

```
> length(z1[z1 < zeta])
[1] 8
```

As always, the final result on the right hand side is assigned to the left hand side using the assignment operator **`<-`**. The left hand side of line 8 is the variable **`nTN`**, the number of true negative decisions (one cannot use the hash tag symbol as in the book, as this is reserved for comments; there are limitations to any programming language). Thus line 8 results in **`nTN`** being assigned the value 8. Click on **`Next`** in the **`Console`** window to execute line 8 and print out **`nTN`**. Using similar logic, line 9 results in **`nTP`**, the number of true positives decisions, being assigned the value 10. Click **`Next`** in the **`Console`** window. Line 10 calculates specificity by dividing the number of TN decisions by the number of non-diseased cases, and also calculates sensitivity by dividing the number of TP decisions by the number of diseased cases. Line 11 prints out these values with a descriptive message. Click **`Continue`** in the **`Console`** window to execute the rest of the code.

Appendix 3.A.3: Executing the entire source code file

The purpose of the preceding section was to explain the working of the program. *One can always examine how the program works by inserting breakpoints and sourcing the code, or one can run each line in turn by clicking on the **`Run`** button; the cursor will automatically advance to the next line to be executed. The **`Run`** button always executes the highlighted section of code or the line at the position of the cursor. Highlighting can be used to run multiple lines, or even print a variable name.* Once satisfied that the program is working as expected, and one wishes to **`Run`** all the lines, exit debug mode by clicking on **`Stop`** in the **`Console`** window, remove the breakpoint by clicking on it, then click on **`Source`**, which executes all lines in the file.

Appendix 3.B.1: Code Listing

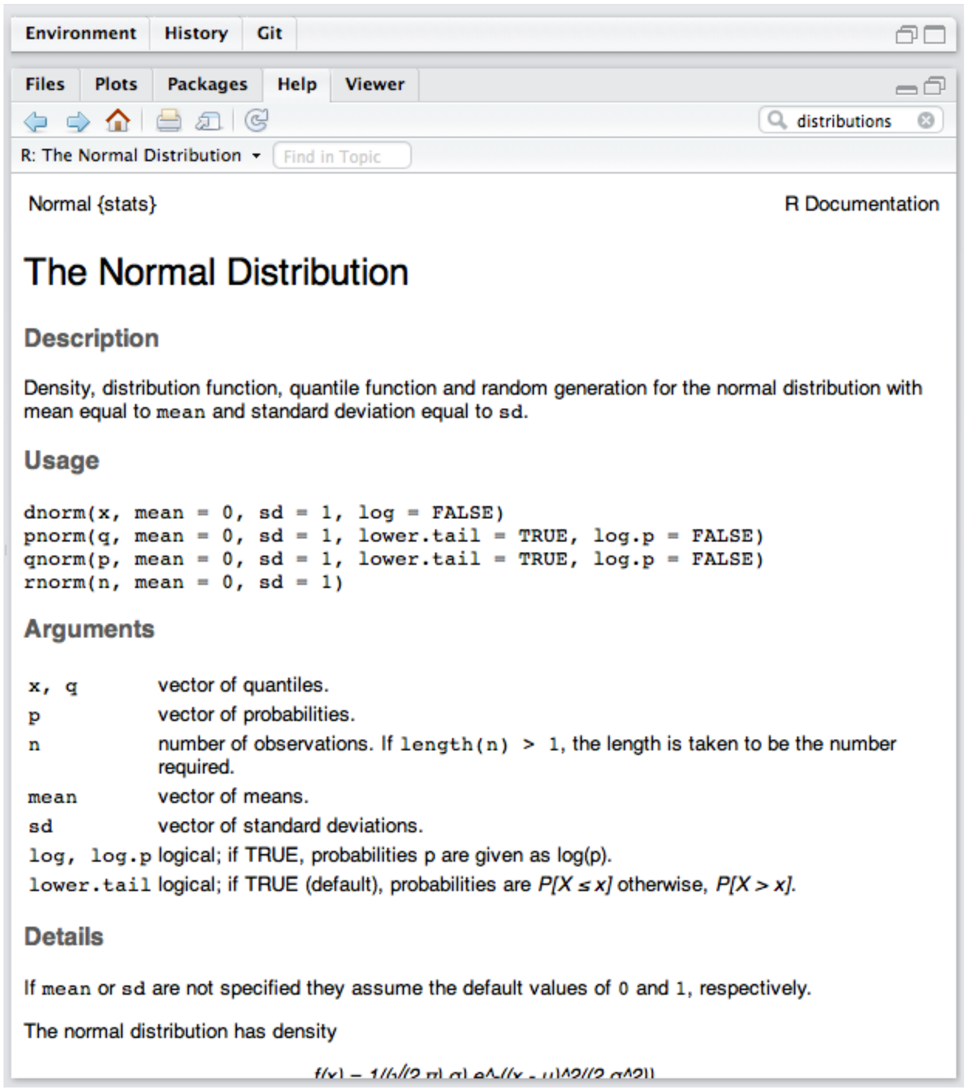
```
# mainConfidenceIntervals.R
rm( list = ls() )
options(digits=3)
seed <- 100;set.seed(seed)
K1 <- 99;K2 <- 111;mu <- 1.5;zeta <- mu/2
z1 <- rnorm(K1)
z2 <- rnorm(K2) + mu
nTN <- length(z1[z1 < zeta])
nTP <- length(z2[z2 >= zeta])
Sp <- nTN/K1;Se <- nTP/K2
alpha <- 0.05
cat("Specificity = ", Sp, "Sensitivity = ", Se, "\n")
# Approx binomial tests
cat("approx 95% CI on Sp = ",
    -abs(qnorm(alpha/2))*sqrt(Sp*(1-Sp)/K1)+Sp,
    +abs(qnorm(alpha/2))*sqrt(Sp*(1-Sp)/K1)+Sp,"\n")

# Exact binomial test
ret <- binom.test(nTN, K1, p = nTN/K1)
cat("Exact 95% CI on Sp = ", as.numeric(ret$conf.int),"\n")

# Approx binomial tests
cat("approx 95% CI on Se = ",
    -abs(qnorm(alpha/2))*sqrt(Se*(1-Se)/K2)+Se,
    +abs(qnorm(alpha/2))*sqrt(Se*(1-Se)/K2)+Se,"\n")

# Exact binomial test
ret <- binom.test(nTP, K2, p = nTP/K2)
cat("Exact 95% CI on S2 = ", as.numeric(ret$conf.int),"\n")
```

The 1st 12 lines are almost identical to those in **`mainBinaryRatings.R`**, explained above. Lines 14-17 calculates the approximate 95% CI for FPF using Eqn. (3.46) and (3.49). Note the usage of the absolute value of the **`qnorm()`** function; **`qnorm`** is the *lower* quantile function for the unit normal distribution, identical to Φ^{-1} and $z_{\alpha/2}$ is the *upper* quantile function. Line 19 – 21 calculates and prints the corresponding exact confidence interval, using the function **`binom.test()`**; one should look up the documentation on this function for further details (in the **`Help`** panel – lower right window - start typing in the function name and **`RStudio`** should complete it) and examine the structure of the returned variable **`ret`**. The remaining code repeats these calculations for TPF.

Part I of the introduction is in the Online Appendix to **Chapter 01**.

Online Appendix 3.C.1: The normal distribution in R

A quick way to learn about available distributions in **R** is to click on the **Help** tab in the bottom-right **RStudio** window and start typing "**distributions**" in the search box (indicated by the magnifying glass in the Mac operating system). By the time the author typed "**dis**", **RStudio** popped-up up some matches, ranging from "**discoveries**" to "**distributions**". When the author selected "distributions" it shows a page of available statistical distributions ranging from the *beta* distribution to the *Weibull* distribution. The one the author is interested in is the *normal* distribution, and the entry **dnorm** is color coded as a link to a webpage (**RStudio** is actually displaying pages from the official **R** help site: http://www.r-project.org/). Clicking on **dnorm** gives me the following, Online Fig. 3.2 (the author is only showing the first few lines; one can look at the whole page on one's computer):



Normal {stats} R Documentation

The Normal Distribution

Description

Density, distribution function, quantile function and random generation for the normal distribution with mean equal to `mean` and standard deviation equal to `sd`.

Usage

```
dnorm(x, mean = 0, sd = 1, log = FALSE)
pnorm(q, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
qnorm(p, mean = 0, sd = 1, lower.tail = TRUE, log.p = FALSE)
rnorm(n, mean = 0, sd = 1)
```

Arguments

`x, q`	vector of quantiles.
`p`	vector of probabilities.
`n`	number of observations. If `length(n) > 1`, the length is taken to be the number required.
`mean`	vector of means.
`sd`	vector of standard deviations.
`log, log.p`	logical; if TRUE, probabilities p are given as log(p).
`lower.tail`	logical; if TRUE (default), probabilities are *P[X ≤ x]* otherwise, *P[X > x]*.

Details

If `mean` or `sd` are not specified they assume the default values of `0` and `1`, respectively.

The normal distribution has density

f(x) = 1/(√(2 π) σ) e^-((x - μ)^2/(2 σ^2))

Online Fig. 3.2: Help page for functions pertaining to the normal distribution.

There are four functions related to the normal distribution, all of which end with **norm** (for normal distribution). The prefixes are **d**, for density, **p**, for probability, **q** for quantile and **r** for random. The same convention is followed for the other distributions. You already have some familiarity with **rnorm()**, so let us start with it. The complete function call is:

```
rnorm(n, mean = 0, sd = 1)
```

Here **rnorm()** is a function (the opening parenthesis signals that the variable preceding it is the name of a function). Think of a function as a black box that takes some input variables, contained within the parentheses, termed *arguments*, and returning, or getting, something useful. With **rnorm** the minimum input is the desired number of samples, which is what the argument **n** stands for. You could enter **rnorm(n = 10)** to get 10 samples from $\mathrm{N}(0,1)$ or one could omit the **n =** and simply enter **rnorm(10)** which would work just as well, but **rnorm(x = 10)** would generate an error:

```
Error in rnorm(x = 10) : unused argument (x = 10)
```

This is because the function is not expecting an argument named *x*. If one does not supply the remaining quantities, the default values denoted by the **=** signs in Online Fig. 3.2 are used. Therefore, **rnorm(10)** is equivalent to **rnorm(n = 10, mean = 0, sd = 1)**. Now if one types (or copies and pastes) these commands conscientiously into the **Console** window, one sees different numbers, because the random number generator produces different sequences of numbers every time it is called. To prevent this from happening, try running the following commands:

```
seed <- 1;set.seed(seed); rnorm(10)
set.seed(seed); rnorm(n = 10, mean = 0, sd = 1)
```

Try it! Copy and paste the above two lines, one line at a time, into the **Console** window, and press the **enter**/**return** key. This time one sees the same sequence of 10 random numbers. The author stresses that this pedagogical device is used only to explain how things work; *one should never reset the seed more than once when conducting actual simulations.* To start with a completely random seed (determined from the system time) use

```
seed <- NULL;set.seed(seed);
```

The author trusts by now one is seeing at least one distinction between the assignment operator **<-** and the **=** operator. The **=** operator is used to override default values of function arguments. To sample from a normal distribution with mean 1.5 and variance 2, denoted $N(1.5,2)$ use the following equivalent forms:

```
seed <- 1;set.seed(seed); rnorm(10, 1.5, sqrt(2))
seed <- 1;set.seed(seed); rnorm(10, mean = 1.5, sd = sqrt(2))
```

The first form involves less typing, but one has to be sure that the arguments are in the correct order, i.e., the first one must correspond to **n**, the second to **mean**, etc. If one has not guessed it, **sqrt()** stands for the square root function [the standard deviation is the square root of the variance; try typing **sqrt(49)** in the **Console** window; one should get **7**]. The reader should study the examples below:

```
> set.seed(seed = 1);rnorm(n = 10, mean = 1.5, sd = sqrt(2)) # example 1
 [1] 0.614 1.760 0.318 3.756 1.966 0.340 2.189 2.544 2.314 1.068
> set.seed(seed = 1);rnorm(10, 1.5, sqrt(2))                 # example 2
 [1] 0.614 1.760 0.318 3.756 1.966 0.340 2.189 2.544 2.314 1.068
> set.seed(seed = 1);rnorm(1.5, 10, sqrt(2))                 # example 3
[1] 9.11
> set.seed(seed = 1);rnorm(mean = 1.5, n = 10, sd = sqrt(2)) # example 4
 [1] 0.614 1.760 0.318 3.756 1.966 0.340 2.189 2.544 2.314 1.068
```

The third example, where the order is not what one intended, **rnorm(1.5, 10, sqrt(2))**, yields one sample from the normal distribution with mean 10 and variance 2. You got one sample, as it is difficult for **R** to satisfy one's request to provide 1.5 samples. The fourth example shows that as long as one also specifies the parameter names, shuffling the order of the arguments has no effect. The style of programming is entirely up to the reader. As the saying goes, “there are more than one ways of skinning a cat”.

Online Appendix 3.C.2: The mean, variance, and standard deviation functions

Click on the file menu in the lower-right **RStudio** window and select the file **mainSamplingNormalDistributions.R**, which is reproduced below.

Online Appendix 3.C.3: Code Listing

```
#mainSamplingNormalDist.R
rm(list = ls()); set.seed(1); options(digits=3)
# get 2 sets of 10 random samples from the unit normal distribution
```

```
x <- rnorm(10);cat("1st set of 10 random samples are \n",x,"\n\n")
x <- rnorm(10);cat("2nd set of 10 random samples are \n",x,"\n\n")
# mean and standard deviation of 10,000 new samples from the unit normal distribution
cat("mean of 10,000 new samples =", mean(x1 <- rnorm(10000)),"\n")
cat("std. of above 10,000 samples =", sqrt(var(x1)),"\n")
```

Line 2 does three things (always in left-to-right order): it clears all variables, sets the seed to unity, and limits displayed numbers to 3 decimal places. Line 3 is a comment explaining what lines 4 and 5 do. Line 4 has two statements separated by a semi-colon. The first statement generates 10 random samples from a unit normal distribution and assigns the values to the variable **`x`**. The second statement prints the helpful remark "**`1st set of 10 random samples are \n`**" where the newline character assures that the numbers themselves will appear on a separate line. The values are printed because of the presence of the variable **`x`** in the **`cat`** argument list. Specifically, the call to cat was:

```
cat("1st set of 10 random samples are \n",x,"\n\n")
```

As noted earlier, **`cat`** stands for concatenate and print. So what is there to concatenate and print? One starts with the string **`("1st set  of 10 random samples are \n"`** which is the helpful message followed by the newline character; the next argument is **`x`** (notice that arguments are *always* comma separated), which in our case is an array with 10 elements; the final argument is two consecutive newline characters, which is a convenient way to achieve clear separation between the outputs of line 4 and 5. [A subtle point at this stage: the assignment in Line 5 overwrites the previously assigned values to the variable **`x`** in line 4.] Line 7 is:

```
cat("mean of 10,000 new samples) =", mean(x1 <- rnorm(10000)),"\n")
```

It prints out a helpful message followed by a number. The number is the value shown below:

```
mean(x1 <- rnorm(10000))
```

Again, starting from inside out, **`rnorm(10000)`**yields ten-thousand samples from the unit normal distribution. This huge array is assigned to the variable **`x1`**. The only reason for doing so is one needs the huge array **`x1`** in line 8 where one calculates the standard deviation of this array - so one needs to save it. Finally this array is passed to the **`mean()`** function, which as one may have guessed, calculates the mean (average) of the array. Since this is the mean of a large number of samples from a unit normal distribution, one expects a value close to zero. Line 8 prints a helpful message followed by the square root **`sqrt()`** of the variance **`var()`** of the huge array **x1**. So one sees that it is possible to calculate the variance and pass the result directly to the square root function. Since this is the standard deviation of a large number of samples from a unit normal distribution, one expects a value close to unity. [In case one is wondering, **R** does have a standard deviation function, **sd()**, but the author chose not to use it.] Sourcing this file ("sourcing" is **RStudio**-speak for clicking on the **Source** button with the file in question in the foreground) yields the following output.

Online Appendix 3.C.4: Code Output

```
> source('~/book/3 R_RStudio/Software/mainSamplingNormalDistribution1.R')
1st set of 10 random samples are
 -0.626 0.184 -0.836 1.6 0.33 -0.82 0.487 0.738 0.576 -0.305

2nd set of 10 random samples are
 1.51 0.39 -0.621 -2.21 1.12 -0.0449 -0.0162 0.944 0.821 0.594

mean of 10,000 new samples = -0.00731
std. of above 10,000 samples = 1.01
```

Online Appendix 3.C.5: Quantiles and CDF of the normal distribution

Open the file **mainPropertiesUnitNormal.R** whose listing follows:

Online Appendix 3.C.6: Code Listing

```
#mainPropertiesUnitNormal.R
rm(list = ls()); set.seed(1); #options(digits=3)
cat("2.5 percentile of distribution =", qnorm(0.025), "\n")
cat("97.5 percentile of distribution =", qnorm(1-0.025), "\n")
cat("P(X<0) =", pnorm(0),"\n")
cat("P(X<-1.96) =", pnorm(-1.96),"\n")
cat("P(X<-Infinity) =", pnorm(-Inf),"\n")
cat("P(X<Infinity) =", pnorm(Inf),"\n")
# plot the CDF
curve(pnorm, xlim=c(-3,3), ylab = "CDF", xlab = "z")
```

Line 3 illustrates the usage of the quantile function of the normal distribution, called **qnorm()**. This is the Φ^{-1} function defined in Eqn. (3.9). Again, **cat()** is used to print a helpful message regarding the variable being printed. Line 3 gives the 2.5th percentile of the unit normal distribution, which is approximately -1.96, i.e., 2.5 percent of the distribution is below -1.96. If one highlighted (or simply put the cursor on the line) and clicked **Run**, one sees:

```
> cat("2.5 percentile of distribution =", qnorm(0.025), "\n")
2.5 percentile of distribution = -1.96
```

Likewise, Line 4 yields the 97.5 percentile of the unit normal distribution, roughly +1.96. Line 6 illustrates the usage of the cumulative distribution function (CDF), the Φ function of the unit normal distribution, and denoted **`pnorm()`** in **R**. To demonstrate that **`pnorm()`** and **`qnorm()`** are indeed inverse functions, try

```
> p <- 0.2; pnorm(qnorm(p))
[1] 0.2
> x <- 2.5111; qnorm(pnorm(x))
[1] 2.5111
```

The second form may not always work: if the magnitude of **`x`** is large, then **`pnorm`** can be so close to unity or zero that due to finite precision it is exactly unity or zero, and the **`qnorm()`** function will give plus or minus infinity, respectively:

```
> x <- 25111; qnorm(pnorm(x))
[1] Inf
> x <- -25111; qnorm(pnorm(x))
[1] -Inf
```

If one **`source`**s the file **`mainPropertiesUnitNormal.R`** one gets the following output:

Online Appendix 3.C.7: Code Output

```
> source('~/book/3 R_RStudio/Software/mainPropertiesUnitNormal.R ')
2.5 percentile of distribution = -1.96
97.5 percentile of distribution = 1.96
P(X<0) = 0.5
P(X<-1.96) = 0.025
P(X<-Infinity) = 0
P(X<Infinity) = 1
```

Online Appendix 3.D: Plotting in R

Open the file **`mainPlotPdfCdf.R`** in the source-code window. The listing follows:

Online Appendix 3.D.1: Code Listing

```
# mainPlotPdfCdf.R
rm(list = ls()); set.seed(1); options(digits=3)
# plot probability density functions (pdfs)
curve(dnorm(x),xlim=c(-2,4),col='blue', ylab = "pdfs", xlab ="z-sample")
curve(dnorm(x, mean = 1.5),col='red', add = TRUE )
```

Line 4 uses the plotting function called **`curve()`**. The first argument to curve has to be a function of **`x`** (in case one missed it, the variable has to be a function of **`x`**, not **`y`** or **`z`**, or any other favorite variable name one might happen to have). In our example the function is **`dnorm(x)`**, which is the density or *pdf* function, Eqn. (3.6). So the function to be plotted by the **`curve()`** function is the unit normal density function $\phi(x)$. The second argument is **`xlim=c(-2,4)`** which sets the internal parameter **`xlim`** of the **`curve()`** function to a vector with elements -2 and 4. This is the first time one are seeing it, but the purpose of the **R** function **`c()`** is to *collect* the values inside the parenthesis (recall, a function always requires a parenthesis to enclose any arguments) into an array (The author has confused **`cat()`** with **`c()`**, with obvious negative consequences). For example, **`c(1,2,4)`** yields a one-dimensional array with elements 1, 2 and 4, respectively. If one assigns them to a variable **`t`**, then **`t[1]`** has the value 1, **`t[2]`** has the value 2 and **`t[3]`** has value 4. Here is the example in ready to copy and paste form for direct use in one's **`Console`** window:

Online Appendix 3.D.2: Code snippet

```
> t <- c(1,2,4)
> t[1];t[2];t[3];t[4]
[1] 1
[1] 2
[1] 4
[1] NA
```

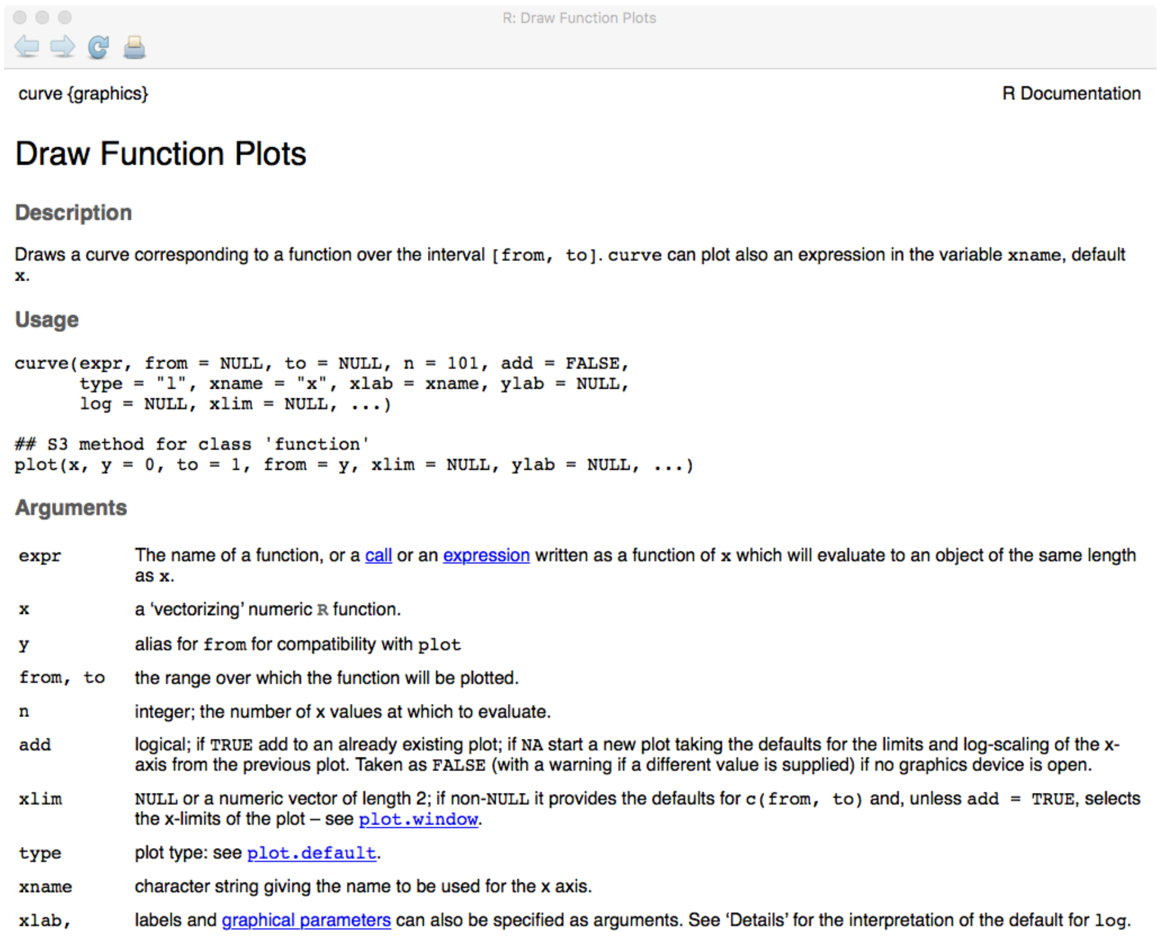
[Notice that the attempt to get an element that was not defined yielded a **NA**, for not available. This is a useful feature of **R** that some other languages (ex. IDL, C) lack.] Returning to the **`curve()`** function, the parameter **`xlim`** stands for "limits for x-axis". So one is asking the **`curve()`** function to plot values in the x-axis range $-2 \le x \le 4$. The next argument is **`col = ‘blue’`**, which sets the color of the plot. The next argument is **`ylab = “pdfs”`**, the label for the y-axis, and the final argument is **`xlab = "z-sample"`**, the label for the x-axis.

Line 5 is a similar call to **`curve()`**, but this time the function being plotted is **`dnorm(x, mean = 1.5)`**, the *pdf* for the diseased cases. An additional argument to curve, namely **`add = TRUE`**, ensures that this plot appears superposed on the one just created by line 4. When **`add = TRUE`** the existing plotting parameters (specified in line 4) are reused. That is why one does not need to re-specify **`xlim=c(-2,4)`**. Without it the second plot appears in a different window and **R** would use the default values for the **`xlim`** parameter, yielding a strange looking *pdf* curve (try removing the **`add = TRUE`** part; the reader should be convinced that the plot is correct, it simply looks strange because the relevant range of the x-axis is not being shown). Also notice that multiple plots are conveniently organized under the **`Plots`** tab, and one can use the arrows under the **`Plots`** tab to navigate through them. When the plots keep piling up one may wish to discard them all by clicking on the **`Clear All`** button (the broom symbol); or one can delete individual plots using the red-cross button next to the **`Export`** tab.

Enough said. Time to **`Source`** the code. The result is Online Fig. 3.2, which appears under the **`Plots`** menu. Using the **`Export`** tab under the **`Plots`** menu, the author copied the plot to the clipboard and pasted it into my manuscript.

Online Appendix 3.E: Getting help in R – Part I

How did I know what argument names to use (e.g., **`xlim`**)? I did not. The author entered **`curve`** in the search bar of the **`Help`** panel to find the help file for this function. Try it, Online Fig. 3.3.



curve {graphics} R Documentation

Draw Function Plots

Description

Draws a curve corresponding to a function over the interval `[from, to]`. `curve` can plot also an expression in the variable `xname`, default `x`.

Usage

```
curve(expr, from = NULL, to = NULL, n = 101, add = FALSE,
      type = "l", xname = "x", xlab = xname, ylab = NULL,
      log = NULL, xlim = NULL, ...)

## S3 method for class 'function'
plot(x, y = 0, to = 1, from = y, xlim = NULL, ylab = NULL, ...)
```

Arguments

`expr`	The name of a function, or a call or an expression written as a function of `x` which will evaluate to an object of the same length as `x`.
`x`	a 'vectorizing' numeric R function.
`y`	alias for `from` for compatibility with `plot`
`from, to`	the range over which the function will be plotted.
`n`	integer; the number of x values at which to evaluate.
`add`	logical; if `TRUE` add to an already existing plot; if `NA` start a new plot taking the defaults for the limits and log-scaling of the x-axis from the previous plot. Taken as `FALSE` (with a warning if a different value is supplied) if no graphics device is open.
`xlim`	`NULL` or a numeric vector of length 2; if non-`NULL` it provides the defaults for `c(from, to)` and, unless `add = TRUE`, selects the x-limits of the plot – see `plot.window`.
`type`	plot type: see `plot.default`.
`xname`	character string giving the name to be used for the x axis.
`xlab,`	labels and graphical parameters can also be specified as arguments. See 'Details' for the interpretation of the default for `log`.

Online Fig. 3.3: The help page for the **`curve`** function.

How did the author know to use **`curve`** in the first place? He did not. This is where a terse well-phrased Google search helps. Preceding the search with [R] can help narrow the search, and then enter one's phrase. For example, entering **`[R] plotting a function`** in the search bar of Google yielded as the top "hit" the help file for the **`curve()`** function:

```
http://stat.ethz.ch/R-manual/R-devel/library/graphics/html/curve.html
```

The ability to get help quickly is very important in the learning curve. The **R** documentation is terse, and many times the author has benefited from a quick search on the Internet. There are literally thousands of people using **R**, and chances are one's question has already been answered.

Online Appendix 3.F: Getting help in R – Part II

The previous method works only if the function one is seeking help on is already available on disk and compiled by **R**. As part of its standard distribution **R** includes a lot of functions, but sometimes one needs to load a **R**-package, which is a collection of functions, not available in the basic distribution. Packages are extension to **R** developed by other researchers and made available on the CRAN website, in open-source fashion, to all users. One example the reader will see often in this book is the **`RJafroc`** package. This package, available on the CRAN website, https://cran.r-project.org/web/packages/RJafroc/index.html, encapsulates most of the analyses techniques described in this book, hiding unnecessary details from the user. This section gives an example of how to get help regarding this package, and by extension, enabling the reader to get help about any package.

Online Appendix 3.F.1: Installing a package

Click **`packages`** in the lower right **`RStudio`** window; click **`Install`** and start typing "**`RJa`**..", before one gets far, one should see **`RJafroc`** in the pop-up menu. Select it: one should see some activity like:

Online Appendix 3.F.2: Code Output

```
> install.packages("RJafroc")
trying URL 'https://cran.rstudio.com/bin/macosx/mavericks/contrib/3.3/RJafroc_0.1.1.tgz'
Content type 'application/x-gzip' length 1072212 bytes (1.0 MB)
==================================================
downloaded 1.0 MB


The downloaded binary packages are in
	/var/folders/d1/mx6dcbzx3v39r260458z2b200000gn/T//Rtmp4XwA1R/downloaded_packages
```

This shows that the required package files have been *downloaded* to one's disk. One also needs to insert a **`library`** command, as in the code below, which causes the package to be read from disk and *compiled* to binary instructions; also, required help files are loaded. The following file **`mainPackage.R`** is an example:

Online Appendix 3.F.3: Code Listing

```
rm(list = ls()) #mainPackage.R
library(RJafroc)
fileName <- "VanDyke.lrc"
rocData <- ReadDataFile(fileName, format = "MRMC")
```

Line 2 compiles the package so that all of its functions are available from within **R**. **`Source`** the code.

Online Appendix 3.F.4: Code output

```
> source('~/book/02 A ROC analysis/A3 ModelingBinaryParadigm/software/mainPackage.R')
Loading required package: tools
Loading required package: xlsx
Loading required package: rJava
Loading required package: xlsxjars
Loading required package: ggplot2
Loading required package: stringr
Loading required package: shiny
```

The multiple "**`Loading required package:`**" come from packages that **`RJafroc`** needs; for example **`ggplot2`** allows one to generate exquisite plots; **`ggplot2`** is a sophisticated **R** plotting package; **`gg`** stands for Grammar of Graphics[1], which is a whole book in itself that one does not need to understand in depth; one just needs to know enough to use it. The package **`xlsx`** allows reading and writing to Excel files. You may not see the above detailed listing on subsequent **`source`** commands. If a package is already loaded, i.e., in compiled form in working memory, then it is not loaded again, so the **`library`** command can be kept permanently, just in case it is needed.

Online Appendix 3.G: What if a package is missing

Sometimes when one **`source`**s a file one might get a listing like this:

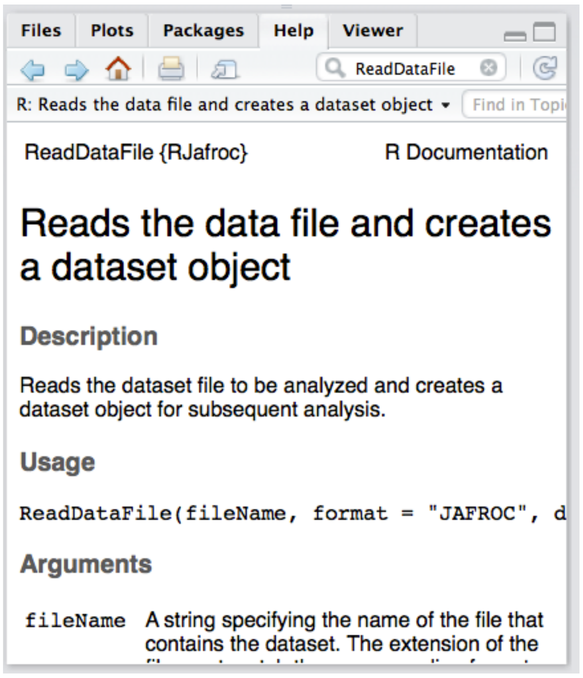
Online Appendix 3.G.1: Code Output

```
> source('~/xzwps/CorCBM/software/mainCorCBM.R')
Find out what's changed in ggplot2 at
http://github.com/hadley/ggplot2/releases.
Loading required package: shiny
Loading required package: xlsx
Loading required package: rJava
Loading required package: xlsxjars
Loading required package: stats4
Error in library(binom) : there is no package called 'binom'
```

The last line is an error message saying, effectively, "can't find a **package** called **binom**". The solution is to install the **package**; one can do it in one line, just type in **install.packages("binom")** into the **Console** window and hit **return**, or go through the **RStudio** interface, as described in Online Appendix 3.F.1.

Online Appendix 3.G.2: Getting help on a function

Line 4 of file **mainPackage.R** uses the **ReadDataFile()** function. To get help on this function proceed as follows. Click on the **Help** tab in the lower right **RStudio** window. Start typing in "**ReadDa**…" and before one gets far **RStudio** will show the full function name, possibly with other near matches, in a pop-up window; select it to obtain Online Fig. 3.4.



ReadDataFile {RJafroc} R Documentation

Reads the data file and creates a dataset object

Description

Reads the dataset file to be analyzed and creates a dataset object for subsequent analysis.

Usage

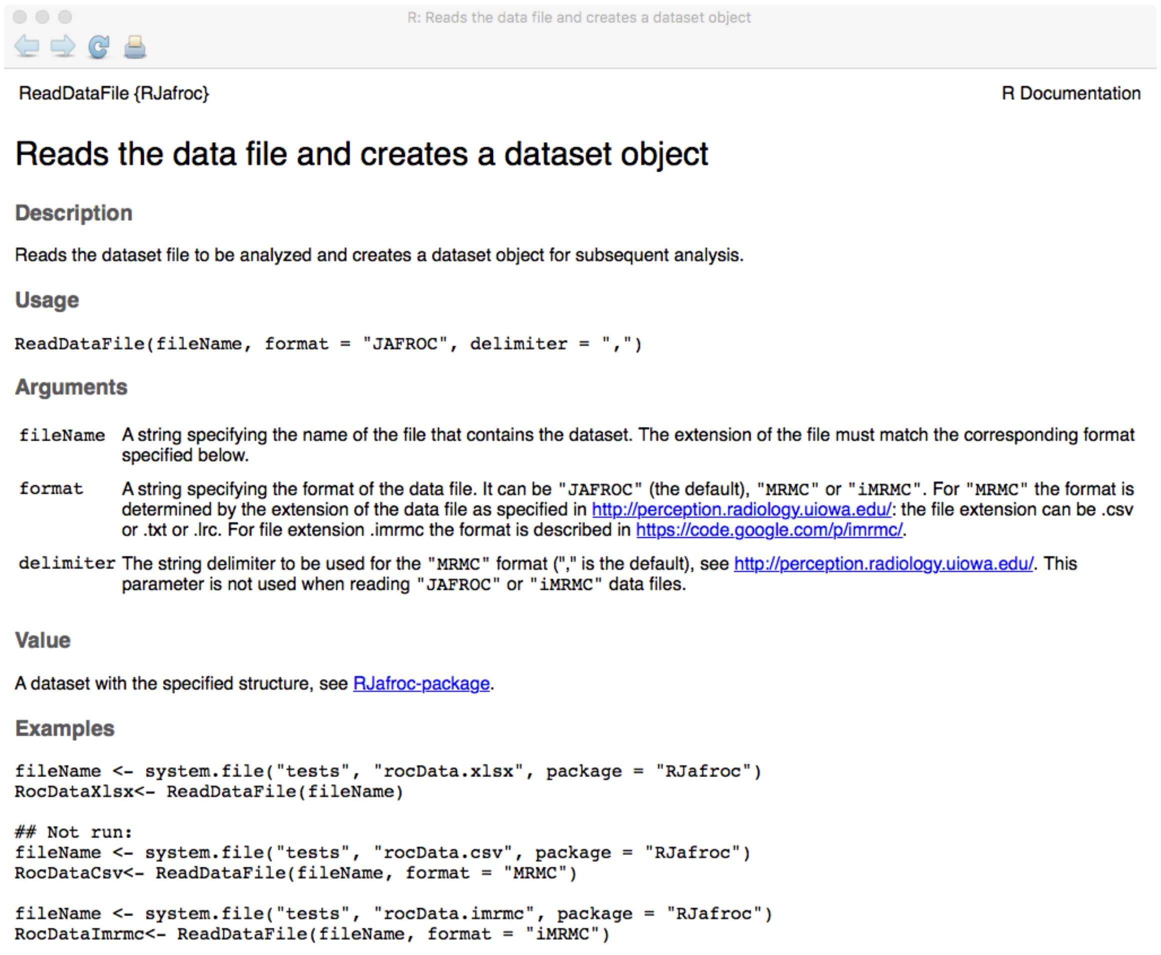
```
ReadDataFile(fileName, format = "JAFROC", d
```

Arguments

fileName A string specifying the name of the file that contains the dataset. The extension of the

Online Fig. 3.4: Viewing the help file for the **ReadDataFile()** function.

Since this display is confined to a small window, there is only so much one can show. Look carefully at the above figure, and just below **Packages**, next the printer icon one sees an icon perhaps best described as "an outward pointing arrow on a document". Hovering the mouse pointer over this icon pops-up the message "**Show in new window**". Click in this icon to get Online Fig. 3.5, the listing of the help file in an independent window, that can be re-sized for ease of reading.



R: Reads the data file and creates a dataset object

ReadDataFile {RJafroc} R Documentation

Reads the data file and creates a dataset object

Description

Reads the dataset file to be analyzed and creates a dataset object for subsequent analysis.

Usage

```
ReadDataFile(fileName, format = "JAFROC", delimiter = ",")
```

Arguments

`fileName`	A string specifying the name of the file that contains the dataset. The extension of the file must match the corresponding format specified below.
`format`	A string specifying the format of the data file. It can be `"JAFROC"` (the default), `"MRMC"` or `"iMRMC"`. For `"MRMC"` the format is determined by the extension of the data file as specified in http://perception.radiology.uiowa.edu/: the file extension can be .csv or .txt or .lrc. For file extension .imrmc the format is described in https://code.google.com/p/imrmc/.
`delimiter`	The string delimiter to be used for the `"MRMC"` format ("," is the default), see http://perception.radiology.uiowa.edu/. This parameter is not used when reading `"JAFROC"` or `"iMRMC"` data files.

Value

A dataset with the specified structure, see RJafroc-package.

Examples

```
fileName <- system.file("tests", "rocData.xlsx", package = "RJafroc")
RocDataXlsx<- ReadDataFile(fileName)

## Not run:
fileName <- system.file("tests", "rocData.csv", package = "RJafroc")
RocDataCsv<- ReadDataFile(fileName, format = "MRMC")

fileName <- system.file("tests", "rocData.imrmc", package = "RJafroc")
RocDataImrmc<- ReadDataFile(fileName, format = "iMRMC")
```

Online Fig. 3.5: Viewing the help file in a new window (partial, the bottom of the window can be seen by scrolling).

Online Appendix 3.G.3: Interpreting the help file

The window has a number of hyperlinks, clicking on which takes one to other documentation files. After a brief description of what the function does, under **`Usage`** one sees:

```
ReadDataFile(fileName, format = "JAFROC", delimiter = ",")
```

The function takes a filename, basically the name of an existing disk file containing the data, the default format is "**`JAFROC`**" and the default delimiter is "**`,`**". At line 4 in **`mainPackage.R`**, format is overridden by **`format = "MRMC"`**. Under **`Arguments`** in Online Fig. 3.5 one sees:

For "MRMC" the format is determined by the extension of the data file as specified in http://perception.radiology.uiowa.edu/: the file extension can be .csv or .txt or .lrc. For file extension .imrmc the format is described in https://code.google.com/p/imrmc/.

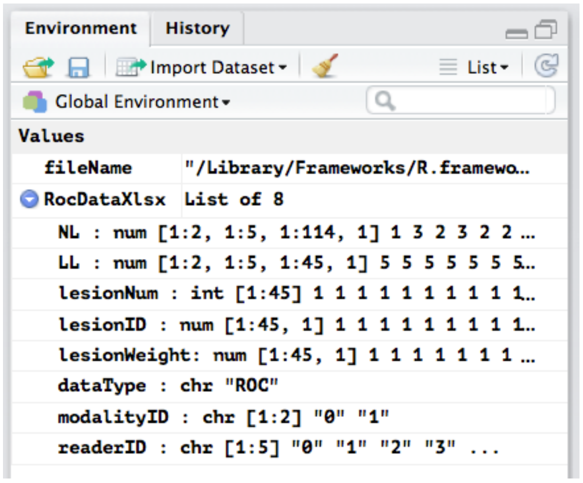
With appropriate specification of the **format** argument, the function can read data in any of the existing formats.

Under **Value** Online Fig. 3.5 lists the quantity returned by the function, in this example an **RJafroc** dataset object. There is a hyperlink to **RJafroc-package** that defines this object. Basically it is a general data format that can accommodate any of the existing data collection paradigms, specifically: ROC, FROC, LROC and ROI.

Under examples are listed some usages of the function. *These can literally be copied and pasted into the* ***Console*** *window to see the function at work*. As an example:

```
fileName <- system.file("tests", "rocData.xlsx", package = "RJafroc")
RocDataXlsx<- ReadDataFile(fileName)
```

On pressing the ⏎ key there is no obvious activity, but if one looks in the **Environment** window, one sees Online Fig. 3.6:



```
Values
  fileName     "/Library/Frameworks/R.framewo…
RocDataXlsx    List of 8
   NL : num [1:2, 1:5, 1:114, 1] 1 3 2 3 2 2 ...
   LL : num [1:2, 1:5, 1:45, 1] 5 5 5 5 5 5 5…
   lesionNum : int [1:45] 1 1 1 1 1 1 1 1 1 1…
   lesionID : num [1:45, 1] 1 1 1 1 1 1 1 1 1…
   lesionWeight: num [1:45, 1] 1 1 1 1 1 1 1 ...
   dataType : chr "ROC"
   modalityID : chr [1:2] "0" "1"
   readerID : chr [1:5] "0" "1" "2" "3" ...
```

Online Fig. 3.6: Result of running the example code in the help file.

To summarize, the function reads a data file and returns a dataset object. The dataset consists of two modalities, five readers, 45 diseased and 114-45 = 69 diseased cases. This is the Van Dyke dataset[2], made famous in the ROC methodology field due to the work of Dorfman, Hillis, Berbaum and colleagues.

Online Appendix 3.H: Shaded distributions in R

Open the file **`mainShadedPlots.R`** and **`Source`** it. You should see Fig. 3.3 in the book. The code is complex and uses the **`R`** package called **`ggplot2`**. A version that is long-winded but simpler to follow is in **`mainShadedPlotsSimple.R`**. *The basic idea is to define the vertices of a polygon that encloses the region to be shaded and use the* **`polygon()`** *function to fill it in with a specified color.* Each polygon vertex requires an *x* and a *y* coordinate stored in two arrays called **`cord.x`** and **`cord.y`**, respectively. [As a past **`C`** programmer, the author cringes at the practice in **`R`** of using the decimal point as part of a variable name, but it is what it is; one can use the underscore character, or even better, good use of upper-case and lower case, termed "**`CamelCase`**" to ensure that one's variable names are more readable, e.g. **`variableName`**.]

If the last vertex is identical to the first vertex the polygon is closed; if not, **`R`** will join the last point to the first point with a straight line. Since the code is a little long, the author shows it in parts. The first 7 lines are:

Online Appendix 3.H.1: Code Listing

```
# mainShadedPlotsSimple.R
rm(list = ls())
mu <- 3;sigma <- 1;zeta <- 1
step <- 0.1

lowerLimit<- -1 # lower limit
upperLimit <- mu + 3*sigma # upper limit
```

Lines 1-3 perform standard initializations: $\mu = 3$, $\sigma = 1$ and $\zeta = 1$. Line 4 initializes the **`step`** variable to 0.1; it determines the spacing of the x-coordinates defining the vertices of the polygon. Lines 6-7 define the variables **`lowerLimit`** (for lower limit) and **`upperLimit`** (for upper limit); these are set to values covering the useful range of z-samples to be plotted (-1 and 6, respectively, see x-axis of Fig. 3.3 in the book). Shown next are lines 9-18:

Online Appendix 3.H.2: Code Listing

```
seqNor <- seq(zeta,upperLimit,step)
cord.x <- c(zeta, seqNor,upperLimit)
# need two y-coords at each end point of range;
# one at zero and one at value of function
```

```
cord.y <- c(0,dnorm(seqNor),0)
curve(dnorm(x,0,1),xlim=c(lowerLimit,upperLimit),col='blue',
      ylab = "pdfs", xlab ="z-sample")
polygon(cord.x,cord.y,col='blue')

curve(dnorm(x,mu,sigma),xlim=c(lowerLimit,upperLimit), add = TRUE, col = 'red')
```

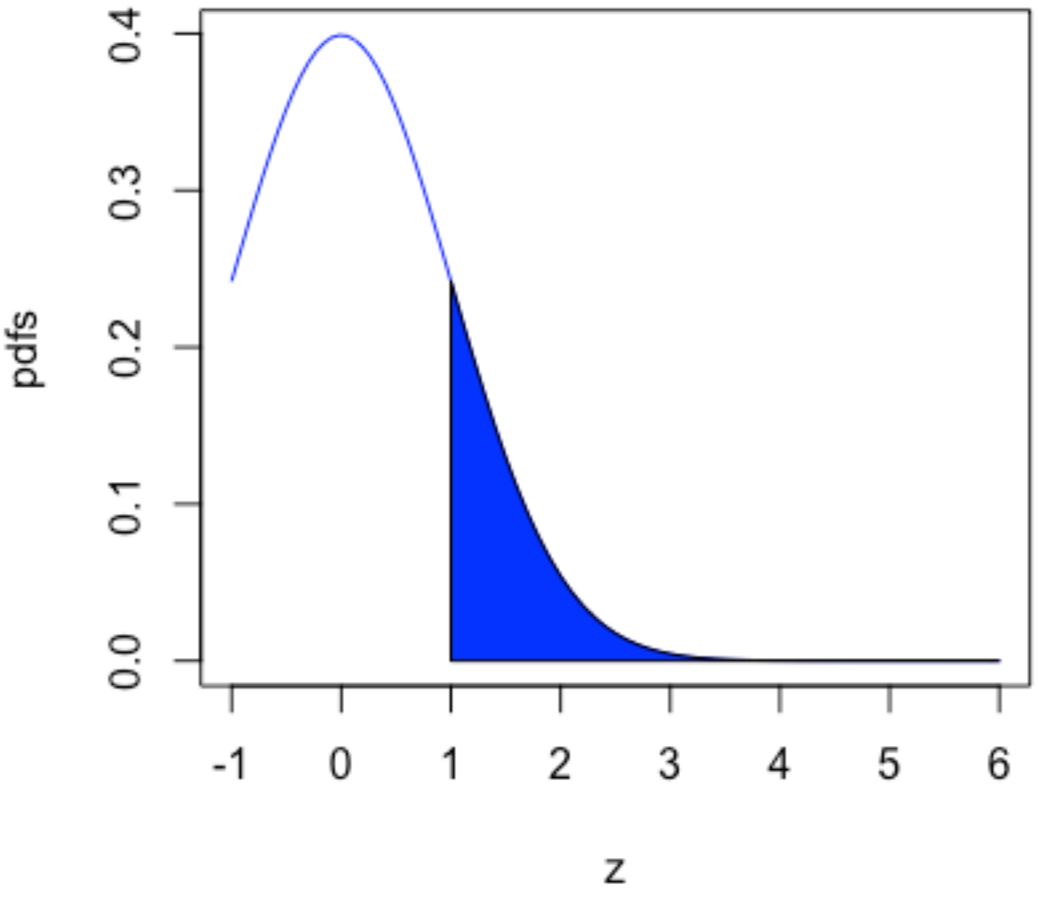
Line 9 uses the **`seq()`** function (for sequence) to generate a sequence of values starting with **`zeta`**, ending with **`upperLimit`** and with spacing **`step`**. In the current example, because **`zeta`** is 1, the sequence is 1, 1.1, 1.2, ….., 4.9, 5. This array is assigned to the variable **`seqNor`** (for sequence for non-diseased cases). Line 10 pads, using the **`c()`** function, **`seqNor`** with an extra **`zeta`** (with value 1) at the beginning of the array and an extra **`upperLimit`** (with value 5) at the end of the array and assigns the result to the variable **`cord.x`**. The padding is for the following reason: at x = 1 one needs two values of y: 0 and $\phi(1)$, respectively, which define the left edge of the region to be shaded blue in Fig. 3.3. Likewise there is a tiny downward jump at the right edge of the region to be shaded blue, with y coordinates equal to $\phi(5)$ and 0, respectively (one uses the convention of always going counter-clockwise in enumerating the vertices of the polygon). The **`dnorm()`** function, which implements the ϕ function, fills in the required values of *y*, corresponding to the upper curved portion of the blue-shaded region shown in Fig. 3.3. The result is assigned to the variable **`cord.y`**. The values of **`cord.y`** are (to see this highlight lines 1-13 and click on **Run**; then highlight **`cord.y`** and click on **Run**):

```
> cord.y
 [1] 0.00e+00 2.42e-01 2.18e-01 1.94e-01 1.71e-01 1.50e-01 1.30e-01 1.11e-01 9.40e-02 7.90e-02
6.56e-02 5.40e-02 4.40e-02 3.55e-02
[15] 2.83e-02 2.24e-02 1.75e-02 1.36e-02 1.04e-02 7.92e-03 5.95e-03 4.43e-03 3.27e-03 2.38e-03
1.72e-03 1.23e-03 8.73e-04 6.12e-04
[29] 4.25e-04 2.92e-04 1.99e-04 1.34e-04 8.93e-05 5.89e-05 3.85e-05 2.49e-05 1.60e-05 1.01e-05
6.37e-06 3.96e-06 2.44e-06 1.49e-06
[43] 8.97e-07 5.36e-07 3.17e-07 1.86e-07 1.08e-07 6.18e-08 3.51e-08 1.98e-08 1.10e-08 6.08e-09
0.00e+00
```

This is a 53-dimensional array (glance at **Environment** panel) whose first and last elements are zeroes (these were deliberately set by us using the zero-padding at the two ends of the array shown on the right hand side of line 13) and the intervening elements were obtained by applying the ϕ function, i.e., **`dnorm`**, to **`seqNor`**.

Lines 14-15: these plot the *pdf* function over the x-range **`lowerLimit`** to **`upperLimit`**, in color blue, and labels the axes appropriately. Now here is an example of how **R** does away with the continuation character for long lines of code, which is necessary in some other languages. Basically **R** interprets a complex statement extending over several lines one line at a time. If the first line makes sense, **R** evaluates it; otherwise it reads the second line, and if still no sense, the third line, etc. [**R** tries really hard to make it easy for the user but one has to be careful how one splits the long sentence; best to leave it ending with a comma, so R is forced to read the nect line.] In the current example, the first line does not make sense because the opening bracket of the **`curve()`** function is not closed at this point, so **R** reads the second line, and since that yields a complete statement (the bracket is closed) **R** evaluates it. Line 16 uses the **`polygon()`** function to superpose the defined polygon over an existing plot and fills it in with the color blue. Finally, line 18 draws the *pdf* of the diseased distribution in red and superposes it on the existing plot (**`add = TRUE`** argument is needed here).

To see what is happening, exit debug mode, clear any existing break points and insert a break point at line 18 (in case one has forgotten, click on the gray area to the left of the line number; a red dot should appear to the left of the line number) and click on **`Source`**. One should see Fig. 3.3 in the book. **R** has completed the defined open polygon by automatically connecting the last point (6,0) to the first point (1,0) with a vertical line.



Online Fig. 3.7: Result of running lines 1 – 17 of code in **`mainShadedPlotsSimple.R`**.

It has been demonstrated how to shade a portion of the non-diseased *pdf* from ζ to infinity. One's cursor should be at line 18. Click on **`Next`** in the debug window (under **`Console`**). You should see the red colored line for the *pdf* of the diseased distribution superposed on the preceding figure (not shown).

Your cursor should be at line 20. Shading in the diseased distribution (i.e., under the red curve) is a little more complicated as one needs to shade the enclosed portion above the non-diseased distribution (blue curve) differently from that below it. The first order of business is to find the point at which the two curves cross each other, roughly $z = 1.5$ according to Fig. 3.3 in the book. Lines 20-21, reproduced below, find the crossing point:

```
crossing <- uniroot(function(x) dnorm(x) - dnorm(x,mu,sigma),
                    lower = 0, upper = 3)$root
```

It uses the one-dimensional root finding function **`uniroot()`** (a root is a zero-crossing of a function). The first argument of **`uniroot()`** is the function whose root is desired. In our example, the function is **`function (x) dnorm(x) - dnorm(x,mu,sigma)`**, which defines **`function(x)`**. In other words, one wishes to find that value of *x* at which the densities of the non-diseased and diseased pdf's are identical. The other arguments are intended to help the function in finding the root. For example, **`lower = 0, upper = 3`** says to restrict the search for the root to the range 0 to 3. The result is assigned to the variable **`crossing`**. Your cursor should be at line 20. Click on **Next** twice. Highlight **`crossing`** and click on **Run**:

```
Browse[2]> crossing
[1] 1.5
Browse[2]>
```

This value should come as no surprise. Since the separation of the two distributions is 3, they must, by symmetry, cross at 1.5.

Next comes the shading part. It will convenient to define the polygon explicitly and not leave it to **R** to make any assumptions about how to close the polygon. Lines 23 – 28 follow:

```
crossing <- max(c(zeta, crossing))
seqAbn <- seq(crossing,upperLimit,step)
cord.x <- c(seqAbn, rev(seqAbn))
# reason for reverse
# we want to explicitly define the polygon
# we dont want R to close it
```

Click on **Next** twice bringing the cursor to line 30. Line 24 defines **`seqAbn`**: it is the sequence 1.5, 1.6, 1.7, ….., 45.9, 6. Line 25 concatenates this array to the *reversed* array using the reverse function **`rev()`**. The result is assigned to **`cord.x`** which looks like: 1.5, 1.6, 1.7, ….., 5.9, 6, 6, 5.9, ……, 1.7, 1.6, 1.5, i.e., it starts at 1.5, extends to 6 and then returns to 1.5, all in steps of 0.1. You can confirm this by highlighting **`cord.x`** and clicking **Run**.

```
Browse[2]> cord.x
 [1] 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2 3.3 3.4 3.5 3.6 3.7
3.8 3.9 4.0 4.1 4.2 4.3 4.4 4.5 4.6 4.7
```

```
[34] 4.8 4.9 5.0 5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 6.0 6.0 5.9 5.8 5.7 5.6 5.5 5.4 5.3 5.2 5.1
5.0 4.9 4.8 4.7 4.6 4.5 4.4 4.3 4.2 4.1
[67] 4.0 3.9 3.8 3.7 3.6 3.5 3.4 3.3 3.2 3.1 3.0 2.9 2.8 2.7 2.6 2.5 2.4 2.3 2.2 2.1 2.0 1.9 1.8
1.7 1.6 1.5
```

Clear any break points, exit debug mode (red square button), insert a break point at line 30 and click on **Source**. Lines 30 – 33 show how one fills in the portion of Fig. 3.3 in the book that is unconditionally red.

```
cord.y <- c()
for (i in seq(1,length(cord.x)/2)) {
  cord.y <- c(cord.y,dnorm(cord.x[i],mu, sigma))
}
```

In line 30 one starts by declaring a **NULL** array with nothing in it, **c()**. Line 31 begins a **for** loop: it says, effectively, **for (i in seq(1,length(cord.x)/2))** do the statements enclosed in the curly brackets. Now **seq(1,length(cord.x)/2)** evaluates to 1, 2, 3, ….., 44, 45, 46. Try it! Highlight **seq(1,length(cord.x)/2**, on line 34, being sure to get the brackets matched up, and click on **Run**. You will see:

```
Browse[2]> seq(1,length(cord.x)/2)
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
32 33 34 35 36 37 38 39 40 41 42 43 44
[45] 45 46
Browse[2]>
```

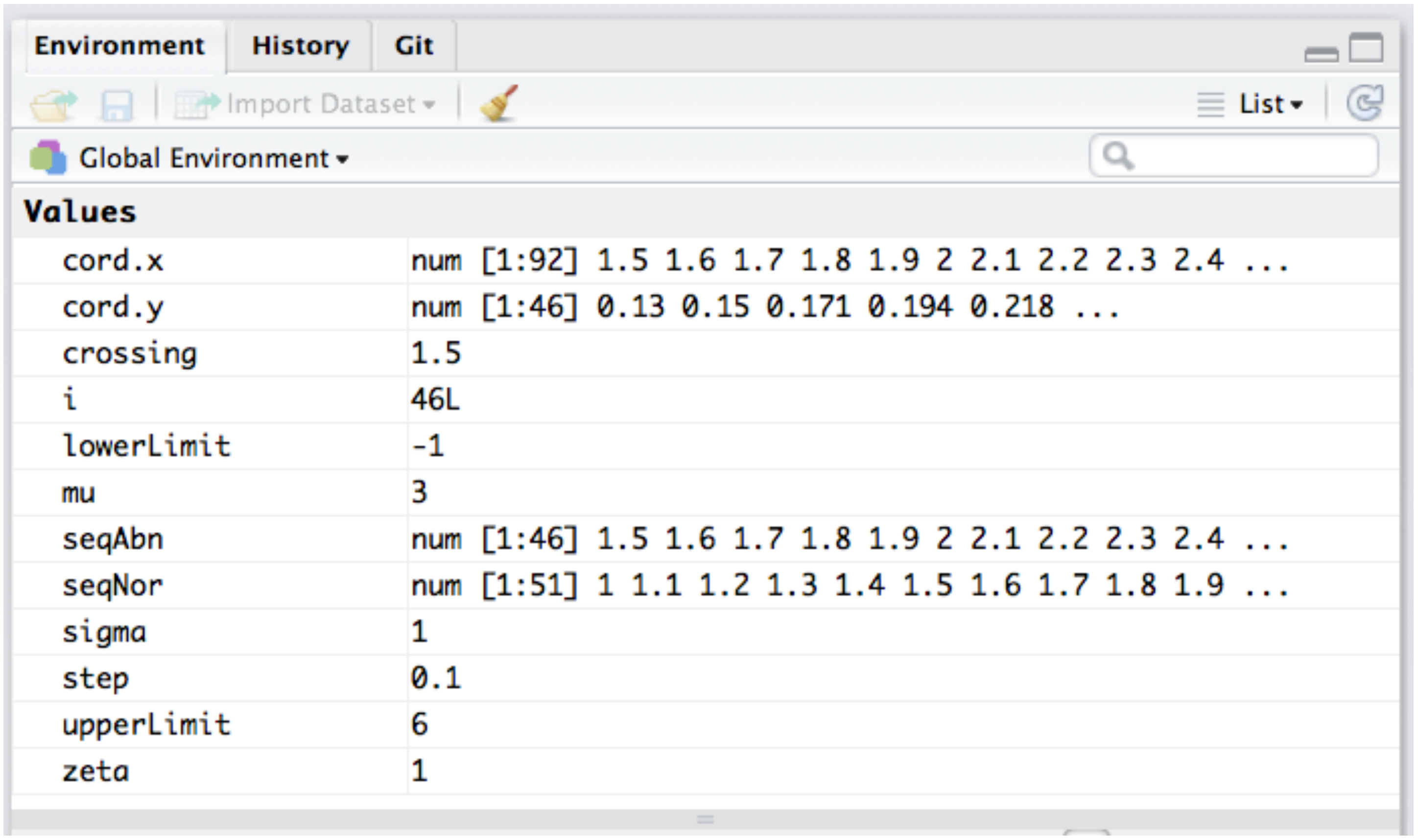
These define the values of **i**. For these values the following statement is executed:

```
cord.y <- c(cord.y,dnorm(cord.x[i],mu, sigma))
```

Click on **Next** a few times; the code pointer should be stuck at line 32. Print out the values of **cord.y** as one goes along. Again, one starts with the right hand side, working from the inside out. The starting value of **i** is unity. So what gets executed is **c(cord.y,dnorm(cord.x[1],mu, sigma))**. Since **cord.x[1]** is the value of the **crossing** variable, this statement evaluates the density function for the diseased *pdf*, centered at 3 and with standard deviation equal to unity; the evaluation is done at $z = 1.5$. This yields 0.13 which value is concatenated with NULL, yielding the one element array 0.13. The next time around (**i** = 2) the **dnorm** function for the diseased *pdf* is evaluated at **cord.x[2]**, i.e., 1.6. This yields 0.15, which is concatenated to the existing one element array 0.13 to give the two-element array 0.13, 0.15. You can see how the array will grow, with each point on the array corresponding to the y-coordinate of a point on the diseased distribution *pdf* whose corresponding x-coordinate is in the first half of the array **cord.x**. Click on the "*get me out of loop*" icon in the debug menu. Highlight **cord.y** and click on **Run**:

```
Browse[2]> cord.y
 [1] 0.12952 0.14973 0.17137 0.19419 0.21785 0.24197 0.26609 0.28969 0.31225 0.33322 0.35207
0.36827 0.38139 0.39104 0.39695 0.39894
[17] 0.39695 0.39104 0.38139 0.36827 0.35207 0.33322 0.31225 0.28969 0.26609 0.24197 0.21785
0.19419 0.17137 0.14973 0.12952 0.11092
[33] 0.09405 0.07895 0.06562 0.05399 0.04398 0.03547 0.02833 0.02239 0.01753 0.01358 0.01042
0.00792 0.00595 0.00443
Browse[2]>
```

If one glances at the **Environment** panel, Online Fig. 3.7, one sees that **cord.x** is an array with 92 elements, while **cord.y** has 46 elements. One's code pointer should be at line 34.



```
Values
cord.x       num [1:92] 1.5 1.6 1.7 1.8 1.9 2 2.1 2.2 2.3 2.4 ...
cord.y       num [1:46] 0.13 0.15 0.171 0.194 0.218 ...
crossing     1.5
i            46L
lowerLimit   -1
mu           3
seqAbn       num [1:46] 1.5 1.6 1.7 1.8 1.9 2 2.1 2.2 2.3 2.4 ...
seqNor       num [1:51] 1 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 ...
sigma        1
step         0.1
upperLimit   6
zeta         1
```

Online Fig. 3.7: Environment window contents, code pointer at line 34.

Now look at lines 34 – 37:

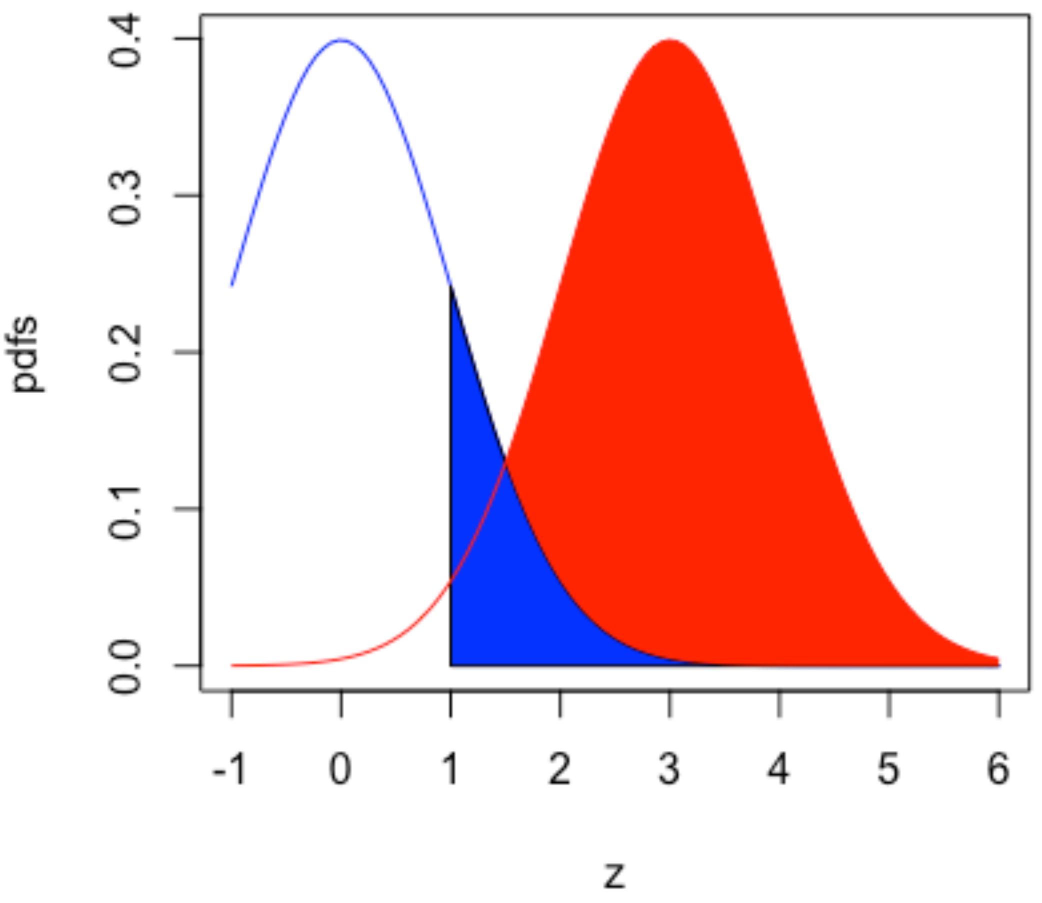
```
for (i in seq(1,length(cord.x)/2)) {
  cord.y <- c(cord.y,dnorm(cord.x[length(cord.x)/2+i]))
}
polygon(cord.x,cord.y,lty = 0, col='red')
```

It too has a **`for`** loop but this time the index of **`cord.x`** is offset by **`length(cord.x)/2`** from the corresponding element in the previous **`for`** loop. In other words, one is using the other half of the **`cord.x`** array, starting at 5, and working back to 1.5:

```
> i <- 1; cord.x[length(cord.x)/2+i]
[1] 5
```

Moreover, this time one is using the *pdf* for the non-diseased cases (line 35; the default values for **`mean`** and **`sd`** are zero and unity, respectively). And every time one keeps augmenting the **`cord.y`** array, building up its length. So the end result is that the first half of the **`cord.y`** array corresponds to the red curve and the second half to the blue curve. The polygon is complete. All that remains is to call the **`polygon`** function line 37, with shading color red.

Clear any break points, exit debug mode, insert a beak point at line 39 and click on **`Source`**; the result is Online Fig. 3.8:



Online Fig. 3.8: Result of running lines 1 – 37 in **`mainShadedPlotsSimple.R`**. The figure is almost complete except for the vertical lines.

It remains to fill in the common portion under the red and blue curves with vertical lines. The code, lines 39 – 47, is:

```
seqAbn <- seq(zeta,upperLimit,step)
seqAbn <- rep(seqAbn,each = 2)
for (i in seq(1,length(seqAbn), 2)) {
  # define xs and ys of two points, separated only along y-axis
  x <- c(seqAbn[i], seqAbn[i+1])
  y <- c(0,dnorm(seqAbn[i+1], mu, sigma))
  # draw vertical line between them
  lines(x,y, col = 'red', lty = 1, lwd = 2)
}
```

Click on **Next**. Line 39 defines a new variable **seqAbn** extending from **zeta** to **upperLimit** in steps of 0.1.

```
Browse[2]> seqAbn
 [1] 1.0 1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9 2.0 2.1 2.2 2.3 2.4 2.5 2.6 2.7 2.8 2.9 3.0 3.1 3.2
3.3 3.4 3.5 3.6 3.7 3.8 3.9 4.0 4.1 4.2
[34] 4.3 4.4 4.5 4.6 4.7 4.8 4.9 5.0 5.1 5.2 5.3 5.4 5.5 5.6 5.7 5.8 5.9 6.0
Browse[2]>
```

Click on **Next**. The code pointer should be at line 41. Using the function **rep()**, line 40 repeats each element of the above array twice (**each = 2**) and replaces **seqAbn**, which now has the values:

```
Browse[2]> seqAbn
  [1] 1.0 1.0 1.1 1.1 1.2 1.2 1.3 1.3 1.4 1.4 1.5 1.5 1.6 1.6 1.7 1.7 1.8 1.8 1.9 1.9 2.0 2.0 2.1
2.1 2.2 2.2 2.3 2.3 2.4 2.4 2.5 2.5 2.6
 [34] 2.6 2.7 2.7 2.8 2.8 2.9 2.9 3.0 3.0 3.1 3.1 3.2 3.2 3.3 3.3 3.4 3.4 3.5 3.5 3.6 3.6 3.7 3.7
3.8 3.8 3.9 3.9 4.0 4.0 4.1 4.1 4.2 4.2
 [67] 4.3 4.3 4.4 4.4 4.5 4.5 4.6 4.6 4.7 4.7 4.8 4.8 4.9 4.9 5.0 5.0 5.1 5.1 5.2 5.2 5.3 5.3 5.4
5.4 5.5 5.5 5.6 5.6 5.7 5.7 5.8 5.8 5.9
[100] 5.9 6.0 6.0
Browse[2]>
```

Line 41 starts a **for**-loop with **i** in the sequence containing alternate elements of **seqAbn**; try it! Highlight **seq(1,length(seqAbn), 2)** and click on **Run**:

```
Browse[2]> seq(1,length(seqAbn), 2)
 [1]   1   3   5   7   9  11  13  15  17  19  21  23  25  27  29  31  33  35  37  39  41  43  45
47  49  51  53  55  57  59  61  63  65
[34]  67  69  71  73  75  77  79  81  83  85  87  89  91  93  95  97  99 101
Browse[2]>
```

On the first pass of the **for** loop, line 43 defines **x** as **c(seqAbn[i], seqAbn[i+1])**, which is the same as the array **c(1,1)**. Line 44 defines y as **c(0,dnorm(seqAbn[i+1], mu, sigma))**; the first element is zero and the second element is the *pdf* of the diseased distribution corresponding to the common value of **x**, namely unity. Line 46 uses the **lines()** function to draw a vertical line between the two points, each defined by two coordinates. The first point is (1,0) and the second point is (1, 0.054). This is repeated on successive passes of the **for** loop to indicate the common area under the non-diseased and the diseased distributions with vertical lines. Try experimenting with different values of the model parameters. The final result (not shown) is almost equivalent to Fig. 3.3 in the book, but lacks the panache of **ggplot2**.

Online Appendix 3.I: Numerical integration in R

Open the file **mainNumericalIntegration.R** in the source-code window. The listing follows:

Online Appendix 3.I.1: Code listing

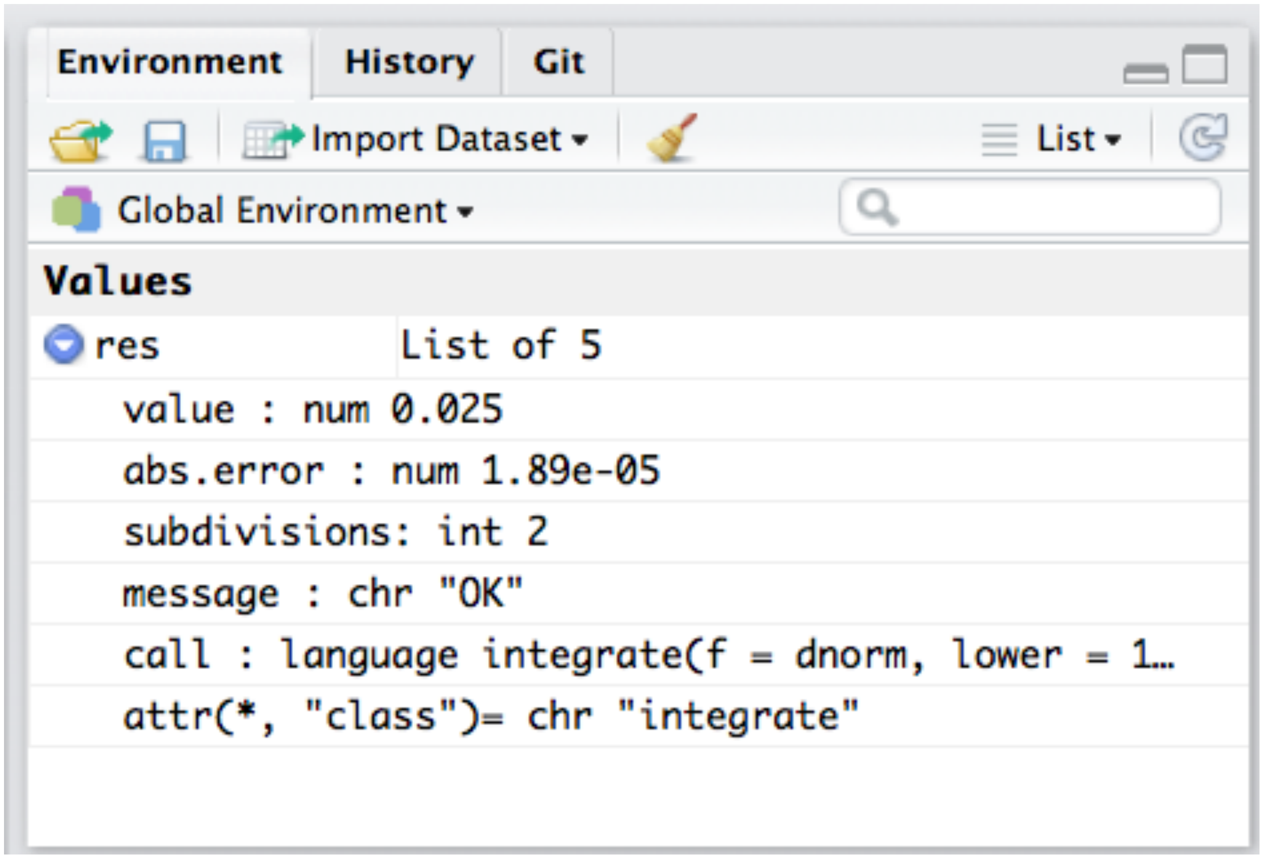
```
#mainNumericalIntegration.R
rm(list = ls()); set.seed(1); #options(digits=3)
# numerically integrate pdf of N(0,1) from -infinity to infinty; should give unity
res <- integrate(dnorm, -Inf, Inf)
cat("integral from -inf to inf = ",res$value, "\n")
# numerically integrate the pdf of N(0,1) from 1.96 to infinty; should give 0.025
res <- integrate(dnorm, 1.96, Inf)
cat("integral from 1.96 to inf = ",res$value, "\n")
```

Line 4 uses the **R** function **integrate()** to numerically integrate the function named in the first argument (no parentheses needed after the name), **dnorm**, in the current example, from -∞ to ∞. Yes, **R** has a symbol for infinity, namely **Inf** (case sensitive). Since **dnorm()** is the *pdf* function, its integral over the entire range of *x* should yield unity. Go ahead and **source** the file. You should see the following output in the **console** window.

Online Appendix 3.I.2: Code output

```
> source('~/book2/02 A ROC analysis/A3
ModelingBinaryParadigm/software/mainNumericalIntegration.R')
integral from -inf to inf =  1
integral from 1.96 to inf =  0.025
```

A brief digression into lists: Notice that the result of the integration, line 4, is assigned to the variable **res** and that in printing ("**cat**"-ing) the result one used the construct **res$value**. If one looks in the **Environment** panel (upper-right) of the **RStudio** interface, Online Fig. 3.9, one sees that **res** is a **List of 5**. Clicking on the arrow next to **res** yields the additional information that the list consists of several list variables: **value**, **abs.error**, and various other quantities that need to be signaled after a properly conducted numerical integration, see Online Fig. 3.9.



```
Values
res                List of 5
  value : num 0.025
  abs.error : num 1.89e-05
  subdivisions: int 2
  message : chr "OK"
  call : language integrate(f = dnorm, lower = 1…
  attr(*, "class")= chr "integrate"
```

Online Fig. 3.9: Environment window showing contents of variable returned by the **`integrate`** function.

For example, the **`list`** variable **`value`** has a member named **`value`** (sic) **`1`** (this is the expected 0.025 resulting from the integration). The quantity **`abs.error`** is 1.89e-05, the absolute error of the numerical integration. *To extract the value of a specific list variable, for example "value", use* **`res$value`**. This says, "starting with the **`list`** variable **`res`**, what is the value of the specific member of this list that is named **`value`**". Try typing the following into the Console window:

Online Appendix 3.I.3: Code snippets

```
> res <- integrate(dnorm, -Inf, Inf)
> names(res)
[1] "value"        "abs.error"    "subdivisions" "message"      "call"
> res$Value
NULL
> res$value
[1] 1
> res$abs.error
[1] 9.36e-05
> res$subdivisions
[1] 3
> res$message
[1] "OK"
```

```
> res$call
integrate(f = dnorm, lower = -Inf, upper = Inf)
```

The second line introduces the **R** function **`names()`** which gives the names of the members of the list supplied as argument to this function. To get further details of the **`integrate()`** function type **`integrate`** in the search window of the **`Help`** panel in the **`RStudio`** interface. Or highlight it and click on the tab button on one's keyboard. Or go to the **R** web site.

Getting back to numerical integration, line 7 numerically integrate the *pdf* of the unit normal distribution from 1.96 to infinity; this should give approximately 0.025. So far one has been limiting the number of decimal places, but now one needs more than 3 decimal places (look at line 1: the author have commented out the **`options`** call.). One may need to restart **R** by going to the **`Session`** menu (main menu of **`RStudio`**) as otherwise any pre-existing value of **`options`** will apply. Now **`source`** the file. The output is:

Online Appendix 3.I.4: Code snippets

```
> source('~/.active-rstudio-document')
integral from -inf to inf =  1
integral from 1.96 to inf =  0.0249979
```

References

1. Wilkinson L. *The grammar of graphics.* Springer Science & Business Media; 2006.
2. Van Dyke CW, White RD, Obuchowski NA, Geisinger MA, Lorig RJ, Meziane MA. Cine MRI in the diagnosis of thoracic aortic dissection. *79th RSNA Meetings.* 1993.