MODULE 4

GRAPHICS

To begin, we'll look at the foundational function for creating graphs: plot(). Then we'll explore how to build a graph, from adding lines and points to attaching a legend.

The Workhorse of R Base Graphics: The plot() Function

The plot() function forms the foundation for much of R's base graphing operations, serving as the vehicle for producing many different kinds of graphs.The plot() is a generic function, or a placeholder for a family of functions. The function that is actually called depends on the class of the object on which it is called. Let's see what happens when we call plot() with an X vector and a Y vector, which are interpreted as a set of pairs in the (x,y) plane.

```
> plot(c(1,2,3), c(1,2,4))
```

This will cause a window to pop up, plotting the points (1,1), (2,2), and (3,4), as shown in Figure 12-1. As you can see, this is a very plain-Jane graph. We'll discuss adding some of the fancy bells and whistles later in the chapter.

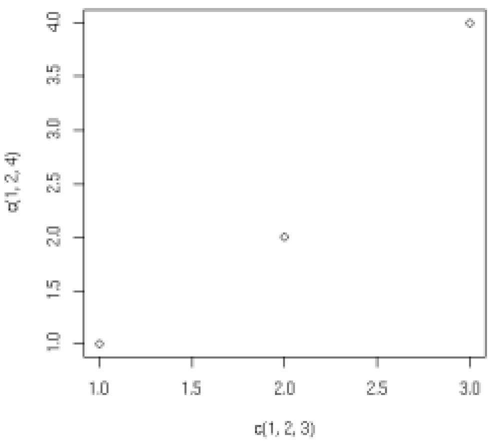

Figure 12-1: Simple point plot

The plot() function works in stages, which means you can build up a graph in stages by issuing a series of commands. For example, as a base, we might first draw an empty graph, with only axes, like this:

```
> plot(c(-3,3), c(-1,5), type = "n", xlab="x", ylab="y")
```

This draws axes labeled x and y. The horizontal (x) axis ranges from −3 to 3. The vertical (y) axis ranges from −1 to 5. The argument type="n" means that there is nothing in the graph itself.

Customizing Graphs

You've seen how easy it is to build simple graphs in stages, starting with plot(). Now you can begin to enhance those graphs, using the many options R provides.

Changing Character Sizes: The cex Option

The cex (for character expand) function allows you to expand or shrink characters within a graph, which can be very useful. You can use it as a named parameter in various graphing functions. For instance, you may wish to draw the text “abc” at some point, say (2.5,4), in your graph but with a larger font, in order to call attention to this particular text. You could do this by typing the following:

```
text(2.5,4,"abc",cex = 1.5)
```

This prints the same text as in our earlier example but with characters 1.5 times the normal size.

Changing the Range of Axes: The xlim and ylim Options

You may wish to have the ranges on the x- and y-axes of your plot be broader or narrower than the default. This is especially useful if you will be displaying several curves in the same graph. You can adjust the axes by specifying the xlim and/or ylim parameters in your call to plot() or points(). For example, ylim=c(0,90000) specifies a range on the y-axis of 0 to 90,000. If you have several curves and do not specify xlim and/or ylim, you should draw the tallest curve first so there is room for all of them. Otherwise, R will fit the plot to the first one your draw and then cut off taller ones at the top! We took this approach earlier, when we plotted two density estimates on the same graph (Figures 12-3 and 12-4). Instead, we could have first found the highest values of the two density estimates. For d1, we find the following:

```
> d1

Call:

density.default(x = testscores$Exam1, from = 0, to = 100)

Data: testscores$Exam1 (39 obs.); Bandwidth 'bw' = 6.967

x y

Min. : 0 Min. :1.423e-07

1st Qu.: 25 1st Qu.:1.629e-03

Median : 50 Median :9.442e-03

Mean : 50 Mean :9.844e-03

3rd Qu.: 75 3rd Qu.:1.756e-02

Max. :100 Max. :2.156e-02
```

So, the largest y-value is 0.022. For d2, it was only 0.017. That means we should have plenty of room if we set ylim at 0.03. Here is how we could draw the two plots on the same picture:

```
> plot(c(0, 100), c(0, 0.03), type = "n", xlab="score", ylab="density")

> lines(d2)

> lines(d1)
```

First we drew the bare-bones plot—just axes without innards, as shown in Figure 12-7. The first two arguments to plot() give xlim and ylim, so that the lower and upper limits on the Y axis will be 0 and 0.03. Calling lines() twice then fills in the graph, yielding Figures 12-8 and 12-9. (Either of the two lines() calls could come first, as we’ve left enough room.)

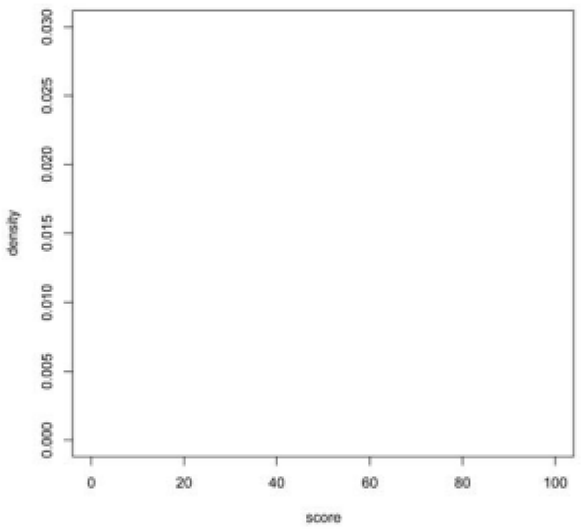

Figure 12-7: Axes only

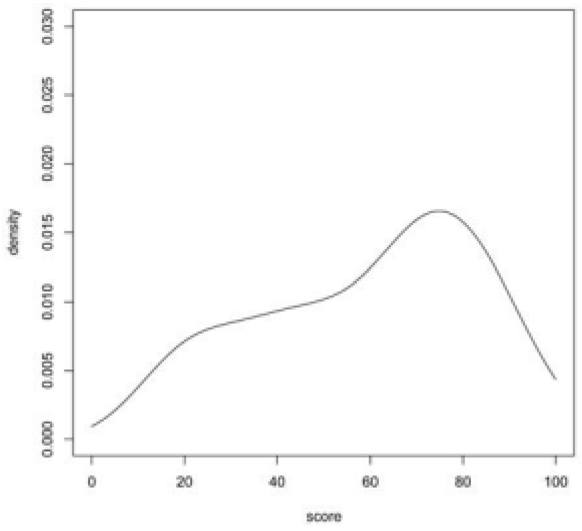

Figure 12-8: Addition of `d2`

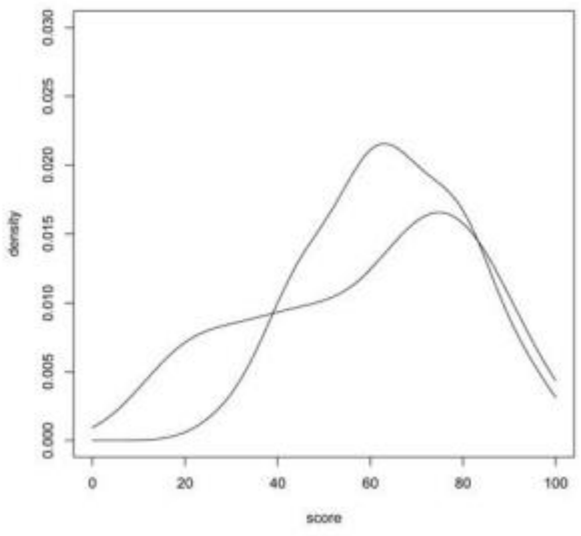

Figure 12-9: Addition of d1

Adding a Polygon: The polygon() Function

You can use polygon() to draw arbitrary polygonal objects. For example, the following code draws the graph of the function f(x)=1 − e−x and then adds a rectangle that approximates the area under the curve from x = 1.2 to x = 1.4.

```
> f <- function(x) return(1-exp(-x))

> curve(f,0,2)

> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),col="gray")
```

The result is shown in Figure 12-10. In the call to polygon() here, the first argument is the set of x-coordinates for the rectangle, and the second argument specifies the y-coordinates. The third argument specifies that the rectangle in this case should be shaded in solid gray. As another example, we could use the density argument to fill the rectangle with striping. This call specifies 10 lines per inch:

```
> polygon(c(1.2,1.4,1.4,1.2),c(0,0,f(1.3),f(1.3)),density=10)
```

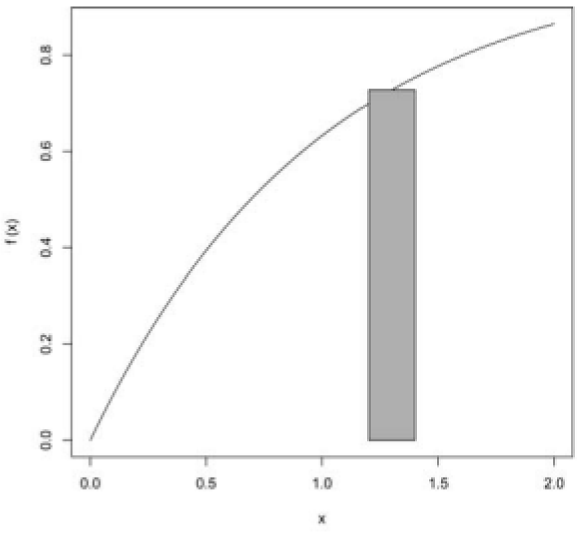

Figure 12-10: Rectangular area strip

Smoothing Points: The lowess() and loess() Functions

Just plotting a cloud of points, connected or not, may give you nothing but an uninformative mess. In many cases, it is better to smooth out the data by fitting a nonparametric regression estimator such as lowess(). Let's do that for our test score data. We'll plot the scores of exam 2 against those of exam 1:

```
> plot(testscores)

> lines(lowess(testscores))
```

The result is shown in Figure 12-11. A newer alternative to lowess() is loess(). The two functions are similar but have different defaults and other options. You need some advanced knowledge of statistics to appreciate the differences. Use whichever you find gives better smoothing.

Graphing Explicit Functions

Say you want to plot the function g(t)=(t 2 + 1)0.5 for t between 0 and 5. You could use the following R code:

```
g <- function(t) { return (t^2+1)^0.5 } # define g()

x <- seq(0,5,length=10000) # x = [0.0004, 0.0008, 0.0012,..., 5]

y <- g(x) # y = [g(0.0004), g(0.0008), g(0.0012), ..., g(5)]

plot(x,y,type="l")
```

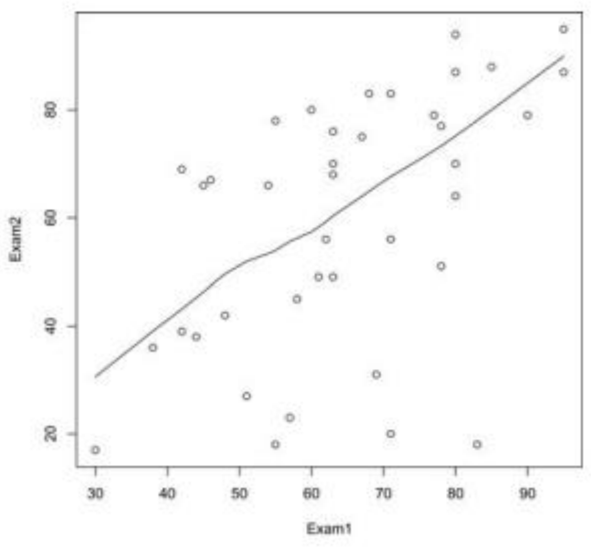

Figure 12-11: Smoothing the exam score relation

But you could avoid some work by using the curve() function, which basically uses the same method:

```
> curve((x^2+1)^0.5,0,5)
```

If you are adding this curve to an existing plot, use the add argument:

```
> curve((x^2+1)^0.5,0,5,add=T)
```

The optional argument n has the default value 101, meaning that the function will be evaluated at 101 equally spaced points in the specified range of x. Use just enough points for visual smoothness. If you find 101 is not enough, experiment with higher values of n. You can also use plot(), as follows:

```
> f <- function(x) return((x^2+1)^0.5)

> plot(f,0,5) # the argument must be a function name
```

Here, the call plot() leads to calling plot.function(), the implementation of the generic plot() function for the function class. Again, the approach is your choice; use whichever one you prefer.

Extended Example: Magnifying a Portion of a Curve

After you use curve() to graph a function, you may want to "zoom in" on one portion of the curve. You could do this by simply calling curve() again on the same function but with a restricted x range. But suppose you wish to display the original plot and the close-up one in the same picture. Here, we will develop a function, which we'll name inset(), to do this. In order to avoid redoing the work that curve() did in plotting the original graph, we will modify its code slightly to save that work, via a return value. We can do this by taking advantage of the fact that you can easily inspect the code of R functions written in R (as opposed to the fundamental R functions written in C), as follows:

```
> curve

function (expr, from = NULL, to = NULL, n = 101, add = FALSE,

type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)

{
```

```
sexpr <- substitute(expr)
if (is.name(sexpr)) {
# ...lots of lines omitted here...
x <- if (lg != "" && "x" %in% strsplit(lg, NULL)[[1]]) {
if (any(c(from, to) <= 0))
stop("'from' and 'to' must be > 0 with log=\"x\"")
exp(seq.int(log(from), log(to), length.out = n))
}
else seq.int(from, to, length.out = n)
y <- eval(expr, envir = list(x = x), enclos = parent.frame())
if (add)
lines(x, y, type = type, ...)
else plot(x, y, type = type, ylab = ylab, xlim = xlim, log = lg, ...)
}
```

The code forms vectors x and y, consisting of the x- and y-coordinates of the curve to be plotted, at n equally spaced points in the range of x. Since we'll make use of those in inset(), let's modify this code to return x and y. Here's the modified version, which we've named crv():

```
> crv
function (expr, from = NULL, to = NULL, n = 101, add = FALSE,
type = "l", ylab = NULL, log = NULL, xlim = NULL, ...)
{
sexpr <- substitute(expr)
if (is.name(sexpr)) {
# ...lots of lines omitted here...
x <- if (lg != "" && "x" %in% strsplit(lg, NULL)[[1]]) {
if (any(c(from, to) <= 0))
stop("'from' and 'to' must be > 0 with log=\"x\"")
exp(seq.int(log(from), log(to), length.out = n))
}
else seq.int(from, to, length.out = n)
y <- eval(expr, envir = list(x = x), enclos = parent.frame())
if (add)
lines(x, y, type = type, ...)
else plot(x, y, type = type, ylab = ylab, xlim = xlim, log = lg, ...)
return(list(x=x,y=y)) # this is the only modification
}
```

Saving Graphs to Files

Let’s open a file:

```
> pdf("d12.pdf")
```

This opens the file d12.pdf. We now have two devices open, as we can confirm:

```
> dev.list()
X11 pdf
2 3
```

The screen is named X11 when R runs on Linux. (It’s named windows on Windows systems.) It is device number 2 here. Our PDF file is device number 3. Our active device is the PDF file:

```
> dev.cur()
pdf
3
```

All graphics output will now go to this file instead of to the screen. But what if we wish to save what’s already on the screen?

Saving the Displayed Graph

One way to save the graph currently displayed on the screen is to reestablish the screen as the current device and then copy it to the PDF device, which is 3 in our example, as follows:

```
> dev.set(2)
X11
2
> dev.copy(which=3)
pdf
3
```

But actually, it is best to set up a PDF device as shown earlier and then rerun whatever analyses led to the current screen. This is because the copy operation can result in distortions due to mismatches between screen devices and file devices.

Closing an R Graphics Device

Note that the PDF file we create is not usable until we close it, which we do as follows:

```
> dev.set(3)
```

```
pdf

3

> dev.off()

X11

2
```

You can also close the device by exiting R, if you’re finished working with it. But in future versions of R, this behavior may not exist, so it’s probably better to proactively close.