Chapter 9

Programming

The ability to read, understand, modify and write simple pieces of code is an essential skill for modern data analysis. Lots of high-quality software already exists for specific purposes, which you can and should use, but statisticians need to grasp how such software works, tweak it to suit their needs, recombine existing pieces of code, and when needed create their own tools. Someone who just knows how to run canned routines is not a data analyst but a technician who tends a machine they do not understand.

Fortunately, writing code is not actually very hard, especially not in R. All it demands is the discipline to think logically, and the patience to practice. This chapter tries to illustrate what's involved, starting from the very beginning. It is redundant for many students, but included through popular demand.

9.1 Functions

Programming in R is organized around **functions**. You all know what a mathematical function is, like $\log x$ or $\phi(z)$ or $\sin\theta$: it is a rule which takes some **inputs** and delivers a definite **output**. A function in R, like a mathematical function, takes zero or more inputs, also called **arguments**, and **returns** an output. The output is arrived at by going through a series of calculations, based on the input, which we specify in the body of the function. As the computer follows our instructions, it may do other things to the system; these are called **side-effects**. (The most common sort of side-effect, in R, is probably making or updating a plot on the screen.) The basic **declaration** or **definition** of a function looks like so:

```
my.function <- function(argument.1, argument.2, ...) {
  # clever manipulations of arguments
  return(the.return.value)
}
```

Strictly speaking, we often don't need the `return()` command; without it, the function will return the last thing it evaluated. But it's usually clearer, and never hurts, to be explicit.

We write functions because we often find ourselves going through the same sequence of steps at the command line, perhaps with small variations. It saves mental effort on our part to take that sequence and bind it together into an integrated procedure, the function, so that then we can think about the function as a whole, rather than the individual steps. It also reduces error, because, by invoking the same function every time, we don't have to worry about missing a step, or wondering whether we forgot to change the third step to be consistent with the second, and so on.

9.2 First Example: Pareto Quantiles

Let me give a really concrete example. In Chapter 5, I mentioned the **Pareto distribution**, which has the probability density function

$$f(x;\alpha,x_0) = \begin{cases} \frac{\alpha-1}{x_0}\left(\frac{x}{x_0}\right)^{-\alpha} & x \geq x_0 \\ 0 & x < x_0 \end{cases} \tag{9.1}$$

Consequently, the CDF is

$$F(x;\alpha,x_0) = 1 - \left(\frac{x}{x_0}\right)^{-\alpha+1} \tag{9.2}$$

and the quantile function is

$$Q(p;\alpha,x_0) = x_0(1-p)^{-\frac{1}{\alpha-1}} \tag{9.3}$$

Say I want to find the median of a Pareto distribution with $\alpha = 2.34$ and $x_0 = 6 \times 10^8$. I can do that:

```
> 6e8 * (1-0.5)^(-1/(2.33-1))
[1] 1010391288
```

If I decide I want the 40th percentile of the same distribution, I can do that:

```
> 6e8 * (1-0.4)^(-1/(2.33-1))
[1] 880957225
```

If I decide to raise the exponent to 2.5, lower the threshold to 1×10^6, and ask about the 92nd percentile, I can do that, too:

```
> 1e6 * (1-0.92)^(-1/(2.5-1))
[1] 5386087
```

But doing this all by hand gets quite tiresome, and at some point I'm going to mess up and write when I meant ^. I'll write a function to do this for me, and so that there is only *one* place for me to make a mistake:

```
qpareto.1 <- function(p, exponent, threshold) {
  q <- threshold*((1-p)^(-1/(exponent-1)))
  return(q)
}
```

The name of the function is what goes on the left of the assignment `<-`, with the declaration (beginning `function`) on the right. (I called this `qpareto.1` to distinguish it from later modifications.) The three terms in the parenthesis after `function` are the arguments to `qpareto` — the inputs it has to work with. The body of the function is just like some R code we would type into the command line, after assigning values to the arguments. The very last line tells the function, explicitly, what its output or return value should be. Here, of course, the body of the function calculates the pth quantile of the Pareto distribution with the exponent and threshold we ask for.

When I enter the code above, defining `qpareto.1`, into the command line, R just accepts it without outputting anything. It thinks of this as assigning certain value to the name `qpareto.1`, and it doesn't produce outputs for assignments when they succeed, just as if I'd said `alpha <- 2.5`.

All that successfully creating a function means, however, is that we didn't make a huge error in the syntax. We should still check that it works, by invoking the function with values of the arguments where we know, by other means, what the output should be. I just calculated three quantiles of Pareto distributions above, so let's see if we can reproduce them.

```
> qpareto.1(p=0.5,exponent=2.33,threshold=6e8)
[1] 1010391288
> qpareto.1(p=0.4,exponent=2.33,threshold=6e8)
[1] 880957225
> qpareto.1(p=0.92,exponent=2.5,threshold=1e6)
[1] 5386087
```

So, our first function seems to work successfully.

9.3 Functions Which Call Functions

If we examine other quantile functions (e.g., `qnorm`), we see that most of them take an argument called `lower.tail`, which controls whether p is a probability from the lower tail or the upper tail. `qpareto.1` implicitly assumes that it's the lower tail, but let's add the ability to change this.

```
qpareto.2 <- function(p, exponent, threshold, lower.tail=TRUE) {
  if(lower.tail==FALSE) {
    p <- 1-p
  }
  q <- threshold*((1-p)^(-1/(exponent-1)))
  return(q)
}
```

When, in a function declaration, an argument is followed by = and an expression, the expression sets the **default value** of the argument, the one which will be used unless explicitly over-ridden. The default value of `lower.tail` is TRUE, so, unless it is explicitly set to false, we will assume `p` is a probability counted from $-\infty$ on up.

The `if` command is a **control structure** — if the condition in parenthesis is true, then the commands in the following braces will be executed; if not, not. Since lower tail probabilities plus upper tail probabilities must add to one, if we are given an upper tail probability, we just find the lower tail probability and proceed as before.

Let's try it:

```
> qpareto.2(p=0.5,exponent=2.33,threshold=6e8,lower.tail=TRUE)
[1] 1010391288
> qpareto.2(p=0.5,exponent=2.33,threshold=6e8)
[1] 1010391288
> qpareto.2(p=0.92,exponent=2.5,threshold=1e6)
[1] 5386087
> qpareto.2(p=0.5,exponent=2.33,threshold=6e8,lower.tail=FALSE)
[1] 1010391288
> qpareto.2(p=0.92,exponent=2.5,threshold=1e6,lower.tail=FALSE)
[1] 1057162
```

First: the answer `qpareto.2` gives with `lower.tail` explicitly set to true matches what we already got from `qpareto.1`. Second and third: the default value for `lower.tail` works, and it works for two different values of the other arguments. Fourth and fifth: setting `lower.tail` to `FALSE` works properly (since the 50th percentile is the same from above or from below, but the 92nd percentile is different, and smaller from above than from below).

The function `qpareto.2` is equivalent to this:

```
qpareto.3 <- function(p, exponent, threshold, lower.tail=TRUE) {
  if(lower.tail==FALSE) {
    p <- 1-p
  }
  q <- qpareto.1(p, exponent, threshold)
  return(q)
}
```

When R tries to execute this, it will look for a function named `qpareto.1` in the workspace. If we have already defined such a function, then R will execute it, with the arguments we have provided, and `q` will become whatever is returned by `qpareto.1`. When we give R the above function definition for `qpareto.3`, it does not check whether `qpareto.1` exists — it only has to be there at run time. If `qpareto.1` changes, then the behavior of `qpareto.3` will change with it, *without our having to redefine* `qpareto.3`.

This is *extremely useful*. It means that we can take our programming problem and sub-divide it into smaller tasks efficiently. If I made a mistake in writing `qpareto.1`, when I fix it, `qpareto.3` automatically gets fixed as well — along with any other function which calls `qpareto.1`, or `qpareto.3` for that matter. If I discover a more efficient way to calculate the quantiles and modify `qpareto.1`, the improvements are likewise passed along to everything else. But when I *write* `qpareto.3`, I don't have to worry about how `qpareto.1` works, I can just assume it does what I need somehow.

9.3.1 Sanity-Checking Arguments

It is good practice, though not *strictly* necessary, to write functions which check that their arguments make sense before going through possibly long and complicated calculations. For the Pareto quantile function, for instance, p must be in $[0,1]$, the exponent α must be at least 1, and the threshold x_0 must be positive, or else the mathematical function just doesn't make sense.

Here is how to check all these requirements:

```
qpareto.4 <- function(p, exponent, threshold, lower.tail=TRUE) {
  stopifnot(p >= 0, p <= 1, exponent > 1, threshold > 0)
  q <- qpareto.3(p,exponent,threshold,lower.tail)
  return(q)
}
```

The function `stopifnot` halts the execution of the function, with an error message, if all of its arguments do not evaluate to `TRUE`. If all those conditions are met, however, R just goes on to the next command, which here happens to be running `qpareto.3`. Of course, I could have written the checks on the arguments directly into the latter.

Let's see this in action:

```
>  qpareto.4(p=0.5,exponent=2.33,threshold=6e8,lower.tail=TRUE)
[1] 1010391288
> qpareto.4(p=0.92,exponent=2.5,threshold=1e6,lower.tail=FALSE)
[1] 1057162
> qpareto.4(p=1.92,exponent=2.5,threshold=1e6,lower.tail=FALSE)
Error: p <= 1 is not TRUE
> qpareto.4(p=-0.02,exponent=2.5,threshold=1e6,lower.tail=FALSE)
Error: p >= 0 is not TRUE
> qpareto.4(p=0.92,exponent=0.5,threshold=1e6,lower.tail=FALSE)
Error: exponent > 1 is not TRUE
> qpareto.4(p=0.92,exponent=2.5,threshold=-1,lower.tail=FALSE)
Error: threshold > 0 is not TRUE
> qpareto.4(p=-0.92,exponent=2.5,threshold=-1,lower.tail=FALSE)
Error: p >= 0 is not TRUE
```

The first two lines give the same results as our earlier functions — as they should, because all the arguments are in the valid range. The third, fourth, fifth and sixth lines all show that `qpareto.4` stops with an error message when one of the conditions in the `stopifnot` is violated. Notice that the error message says *which* condition was violated. The seventh line shows one limitation of this: the arguments violate *two* conditions, but `stopifnot`'s error message will only mention the *first* one. (What is the other violation?)

9.4 Layering Functions and Debugging

Functions can call functions which call functions, and so on indefinitely. To illustrate, I'll write a function which generates Pareto-distributed random numbers, using the "quantile transform" method from Lecture 7. This, remember, is to generate a uniform random number U on $[0,1]$, and produce $Q(U)$, with Q being the quantile function of the desired distribution.

The first version contains a deliberate bug, which I will show how to track down and fix.

```
rpareto <- function(n,exponent,threshold) {
  x <- vector(length=n)
  for (i in 1:n) {
    x[i] <- qpareto.4(p=rnorm(1),exponent=exponent,threshold=threshold)
  }
  return(x)
}
```

Notice that this calls `qpareto.4`, which calls `qpareto.3`, which calls `qpareto.1`.

Let's this out:

```
> rpareto(10)
Error in exponent > 1 : 'exponent' is missing
```

This is a puzzling error message — the expression `exponent` >1 never appears in `rpareto`! The error is coming from further down the chain of execution. We can see where it happens by using the `traceback()` function, which gives the chain of function calls leading to the latest error:

```
> rpareto(10)
Error in exponent > 1 : 'exponent' is missing
> traceback()
3: stopifnot(p >= 0, p <= 1, exponent > 1, threshold > 0)
2: qpareto.4(p = rnorm(1), exponent = exponent, threshold = threshold)
1: rpareto(10)
```

`traceback()` outputs the sequence of function calls leading up to the error in reverse order, so that the last line, numbered `1`, is what we actually entered on the command line. This tells us that the error is happening when `qpareto.4` tries to check the arguments to the quantile function. And the reason it is happening is that we are not providing `qpareto.4` with any value of `exponent`. And the reason *that* is happening is that we didn't give `rpareto` any value of `exponent` as an explicit argument when we called it, and our definition didn't set a default.

Let's try this again.

```
> rpareto(n=10,exponent=2.5,threshold=1)
Error: p <= 1 is not TRUE
> traceback()
```

```
4: stop(paste(ch, " is not ", if (length(r) > 1L) "all ", "TRUE",
       sep = ""), call. = FALSE)
3: stopifnot(p >= 0, p <= 1, exponent > 1, threshold > 0)
2: qpareto.4(p = rnorm(1), exponent = exponent, threshold = threshold)
1: rpareto(n = 10, exponent = 2.5, threshold = 1)
```

This is progress! The `stopifnot` in `qpareto.4` is at least able to evaluate all the conditions — it just happens that one of them is false. (The line `4` here comes from the internal workings of `stopifnot`.) The problem, then, is that `qpareto.4` is being passed a negative value of `p`. This tells us that the problem is coming from the part of `rpareto.1` which sets `p`. Looking at that,

```
p = rnorm(1)
```

the culprit is obvious: I stupidly wrote `rnorm`, which generates a *Gaussian* random number, when I meant to write `runif`, which generates a *uniform* random number.[1]

The obvious fix is just to replace `rnorm` with `runif`

```
rpareto <- function(n,exponent,threshold) {
  x <- vector(length=n)
  for (i in 1:n) {
    x[i] <- qpareto.4(p=runif(1),exponent=exponent,threshold=threshold)
  }
  return(x)
}
```

Let's see if this is enough to fix things, or if I have any other errors:

```
> rpareto(n=10,exponent=2.5,threshold=1)
 [1] 1.000736 2.764087 2.775880 1.058910 1.061712 2.142950 4.220731
 [8] 1.496793 3.004766 1.194545
```

This function at least produces numerical return values rather than errors! Are they the right values?

We can't expect a random number generator to always give the same results, so I can't cross-check this function against direct calculation, the way I could check `qpareto.1`. (Actually, one way to check a random number generator is to make sure it *doesn't* give identical results when run twice!) It's at least encouraging that all the numbers are above `threshold`, but that's not much of a test. However, since this *is* a random number generator, if I use it to produce a lot of random numbers, the quantiles of the output should be close to the theoretical quantiles, which I *do* know how to calculate.

```
> r <- rpareto(n=1e4,exponent=2.5,threshold=1)
> qpareto.4(p=0.5,exponent=2.5,threshold=1)
[1] 1.587401
> quantile(r,0.5)
```

[1] I actually made this exact mistake the first time I wrote the function, back in 2004.

```
     50%
1.598253
> qpareto.4(p=0.1,exponent=2.5,threshold=1)
[1] 1.072766
> quantile(r,0.1)
     10%
1.072972
> qpareto.4(p=0.9,exponent=2.5,threshold=1)
[1] 4.641589
> quantile(r,0.9)
     90%
4.526464
```

This looks pretty good. Figure 9.1 shows a plot comparing all the theoretical percentiles to the simulated ones, confirming that we didn't just get lucky with choosing particular percentiles above.

9.4.1 More on Debugging

Everyone who writes their own code spends a lot of time debugging[2]. There are some guidelines for making it easier and less painful.

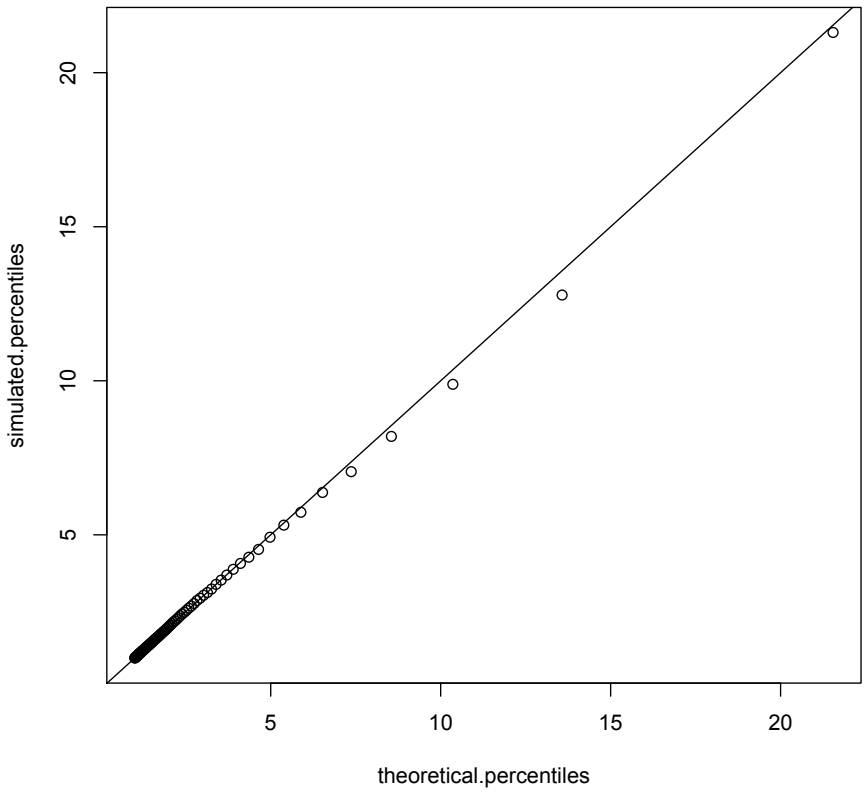
Characterize the Bug We've got a bug when the code we've written won't do what we want. To fix this, it helps a lot to know exactly what error we're seeing. The first step to this is to make the error reproducible. Can we always get the error when re-running the same code and values? If we start the same code in a clean copy of R, does the same thing happen? Once we can reproduce the error, we map its boundaries. How much can we change the inputs and get the same error? A different error? For what inputs (if any) does the bug go away? How big is the error?

Localize the Bug The problem *may* be a diffuse all-pervading wrongness, but often it's a lot more localized, to a few lines or even just one line of code; it helps to know where! We have seen some tools for localizing the bug above: `traceback()` and `stopifnot()`. Another very helpful one is to add `print` statements, so that our function gives us messages about the progress of its calculations, selected variables, etc., as it goes; the `warning` command can be used to much the same effect[3].

Fix the Bug Once you know what's going wrong and where it's going wrong, it's often not too hard to spot the error, either one of syntax (say = vs. ==) or logic. Try a fix and see if it makes it better. Do the inputs which gave you the bugs before now work properly? Are you getting different errors?

[2] Those who don't write their own code but use computers anyway spend a lot of time putting up with other people's bugs.

[3] Real software engineers look down on this, in favor of more sophisticated tools, like interactive debuggers. They have something of a point, but that's usually over-kill for the purposes of this class.




```
simulated.percentiles <- quantile(r,(0:99)/100)
theoretical.percentiles <- qpareto.4((0:99)/100,exponent=2.5,threshold=1)
plot(theoretical.percentiles,simulated.percentiles)
abline(0,1)
```

Figure 9.1: Theoretical percentiles of the Pareto distribution with $\alpha = 2.5$, $x_0 = 1$, and empirical percentiles from a sample of 10^4 values simulated from it with the `rpareto` function. (The solid line is the $x = y$ diagonal, for visual reference.)

9.5 Automating Repetition and Passing Arguments

The match between the theoretical quantiles and the simulated ones in Figure 9.1 is close, but it's not perfect. On the one hand, this might indicate some subtle mistake. On the other hand, it might just be random sampling noise — `rpareto` *is* supposed to be a random number generator, after all. We could check this by seeing whether we get *different* deviations around the line with different runs of `rpareto`, or if on the contrary they all pull in the same direction. We could just make many plots by hand, the way we made that plot by hand, but since we're doing almost exactly the same thing many times, let's write a function.

```
pareto.sim.vs.theory <- function() {
  r <- rpareto(n=1e4,exponent=2.5,threshold=1)
  simulated.percentiles <- quantile(r,(0:99)/100)
  points(theoretical.percentiles,simulated.percentiles)
}
```

This doesn't return anything. All it does is draw a new sample from the same Pareto distribution as before, re-calculate the simulated percentiles, and add them to an existing plot — this is an example of a side-effect. Notice also that the function presumes that `theoretical.percentiles` already exists. (The theoretical percentiles won't need to change from one simulation draw to the next, so it makes sense to only calculate them once.)

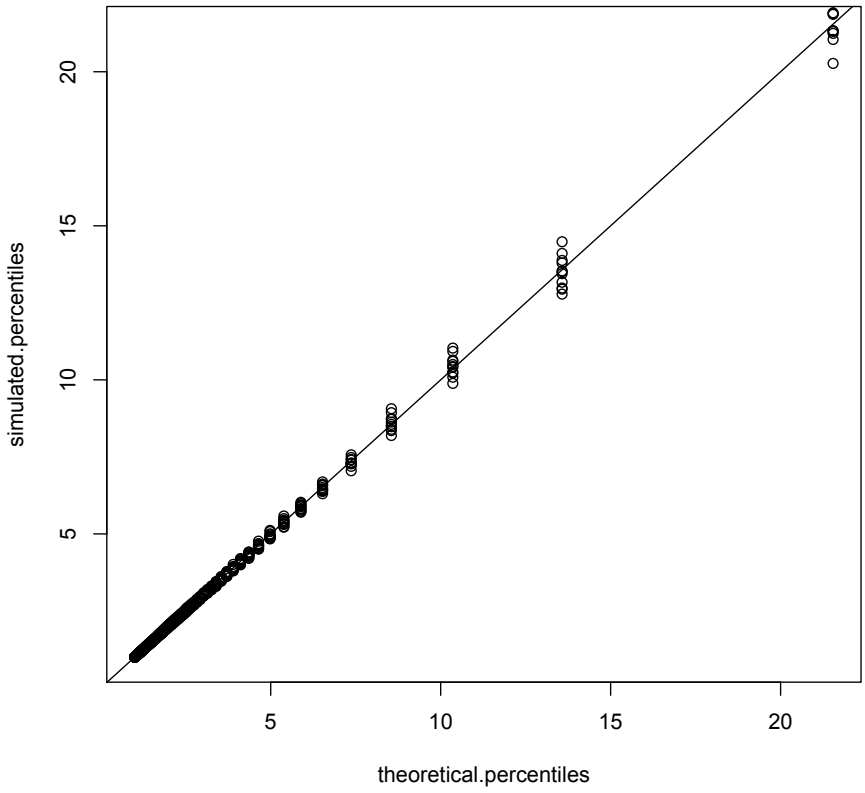
Figure 9.2 shows how we can use it to produce multiple simulation runs. We can see that, looking over many simulation runs, the quantiles seem to be too large about as often, and as much, as they are too low, which is reassuring.

One thing which that figure doesn't do is let us trace the connections between points from the same simulation. More generally, we can't modify the plotting properties, which is kind of annoying. This is easily fixed modifying the function to **pass along arguments**:

```
pareto.sim.vs.theory <- function(...) {
  r <- rpareto(n=1e4,exponent=2.5,threshold=1)
  simulated.percentiles <- quantile(r,(0:99)/100)
  points(theoretical.percentiles,simulated.percentiles,...)
}
```

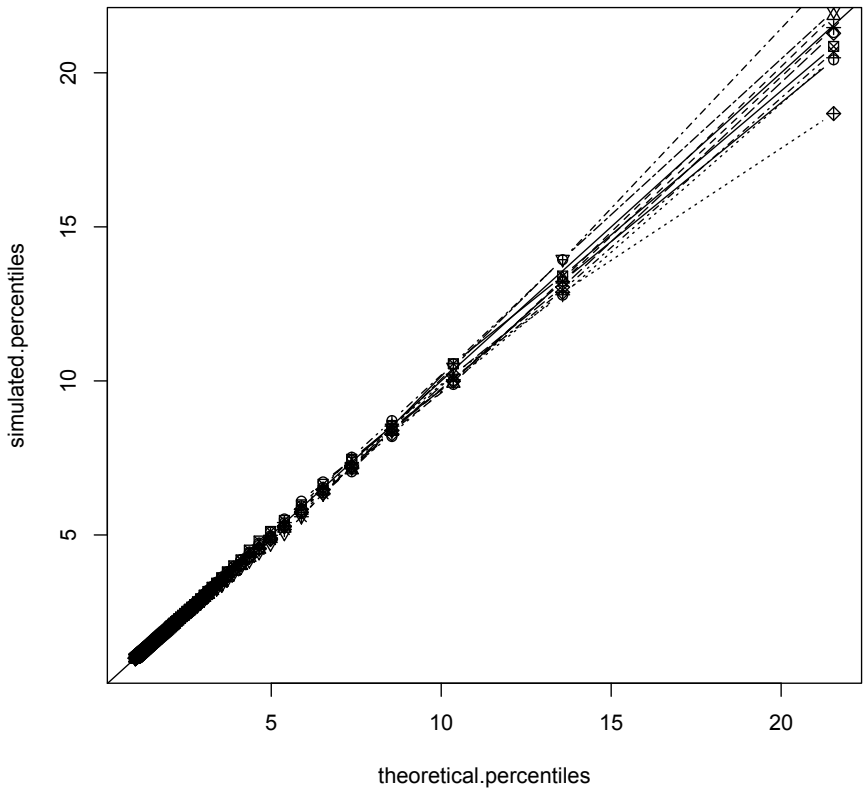
Putting the ellipses (`...`) in the argument list means that we can give `pareto.sim.vs.theory.2` an arbitrary collection of arguments, but with the expectation that it will pass them along unchanged to some other function that it will call with `...` — here, that's the `points` function. Figure 9.3 shows how we can use this, by passing along graphical arguments to `points` — in particular, telling it to connect the points by lines (`type="b"`), varying the shape of the points (`pch=i`) and the line style (`lty=i`).

These figures are reasonably convincing that nothing is going seriously wrong with the simulation for *these* parameter values. To check other parameter settings, again, I could repeat all these steps by hand, or I could write another function:




```
simulated.percentiles <- quantile(r,(0:99)/100)
theoretical.percentiles <- qpareto.4((0:99)/100,exponent=2.5,threshold=1)
plot(theoretical.percentiles,simulated.percentiles)
abline(0,1)
for (i in 1:10) {
  pareto.sim.vs.theory()
}
```

Figure 9.2: Comparing multiple simulated quantile values to the theoretical quantiles.




```
simulated.percentiles <- quantile(r,(0:99)/100)
theoretical.percentiles <- qpareto.4((0:99)/100,exponent=2.5,threshold=1)
plot(theoretical.percentiles,simulated.percentiles)
abline(0,1)
for (i in 1:10) {
  pareto.sim.vs.theory(pch=i,type="b",lty=i)
}
```

Figure 9.3: As Figure 9.2, but using the ability to pass along arguments to a subsidiary function to distinguish separate simulation runs.

```
check.rpareto <- function(n=1e4,exponent=2.5,threshold=1,B=10) {
  # One set of percentiles for everything
  theoretical.percentiles <- qpareto.4((0:99)/100,exponent=exponent,
                                       threshold=threshold)
  # Set up plotting window, but don't put anything in it:
  plot(0,type="n", xlim=c(0,max(theoretical.percentiles)),
       # No more horizontal room than we need
       ylim=c(0,1.1*max(theoretical.percentiles)),
       # Allow some extra vertical room for noise
       xlab="theoretical percentiles", ylab="simulated percentiles",
       main = paste("exponent = ", exponent, ", threshold = ", threshold))
  # Diagonal, for visual reference
  abline(0,1)
  for (i in 1:B) {
    pareto.sim.vs.theory(n=n,exponent=exponent,threshold=threshold,
                         pch=i,type="b",lty=i)
  }
}
```

Defining this will work just fine, but it won't work properly until we re-defined `pareto.sim.vs.theory` to take the arguments `n`, `exponent` and `threshold`.[4]

It seems like a simple modification of the old definition should do the trick:

```
pareto.sim.vs.theory <- function(n,exponent,threshold,...) {
  r <- rpareto(n=n,exponent=exponent,threshold=threshold)
  simulated.percentiles <- quantile(r,(0:99)/100)
  points(theoretical.percentiles,simulated.percentiles,...)
}
```

After defining this, the checker function seems to work fine. The following commands produce the plot in Figure 9.4, which looks very like the manually-created one. (Random noise means it won't be exactly the same.) Putting in the default arguments explicitly gives the same results (not shown).

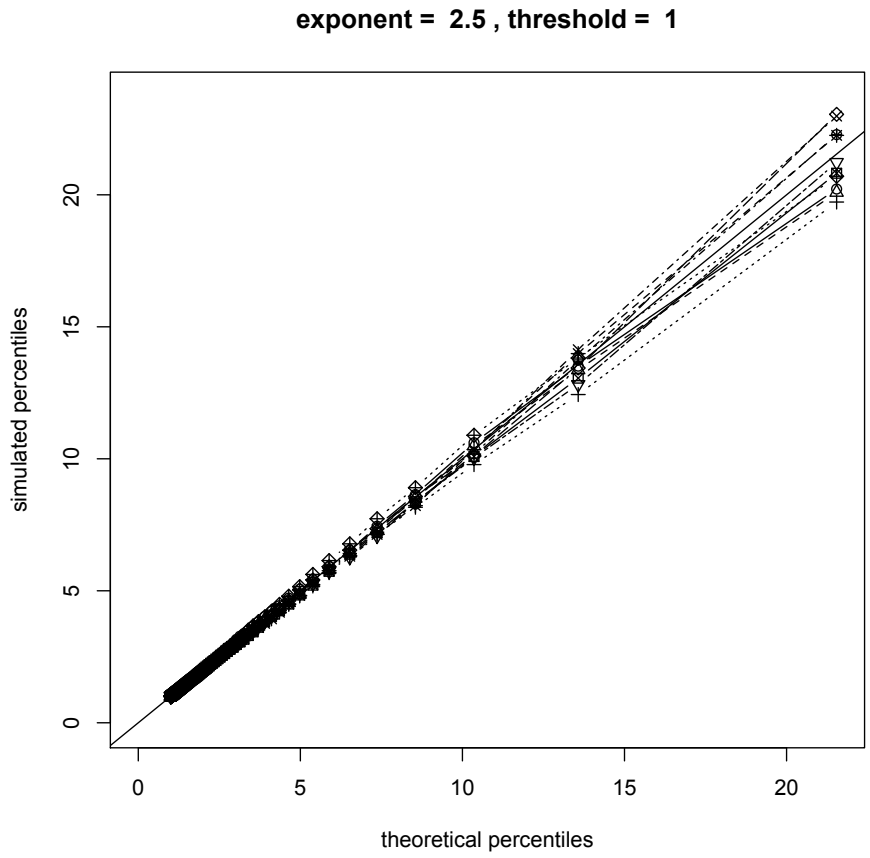
```
> check.rpareto()
> check.rpareto(n=1e4,exponent=2.5,threshold=1)
```

Unfortunately, changing the arguments reveals a bug (Figure 9.5). Notice that the vertical coordinates of the points, coming from the simulation, look like they have about the same range as the theoretical quantiles, used to lay out the plotting window. But the horizontal coordinates are all pretty much the same (on a scale of tens of billions, anyway). What's going on?

The horizontal coordinates for the points being plotted are set in `pareto.sim.vs.theory.3`:

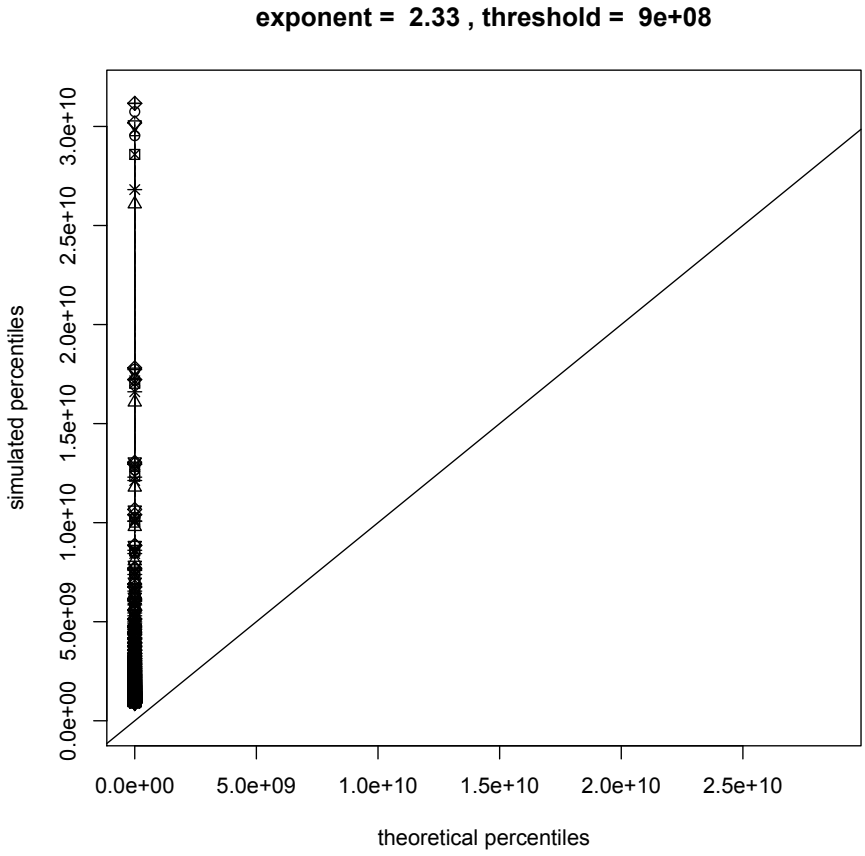
```
  points(theoretical.percentiles,simulated.percentiles,...)
```

[4] Try running `check.rpareto()`, follows by `warnings()`.




```
check.rpareto()
```

Figure 9.4: Automating the checking of `rpareto`.




```
check.rpareto(n=1e4,exponent=2.33,threshold=9e8)
```

Figure 9.5: A bug in `check.rpareto`.

Where does this function get `theoretical.percentiles` from? Since the variable isn't assigned inside the function, R tries to figure it out from context. Since `pareto.sim.vs.theory` was defined on the command line, the context R uses to interpret it is the global workspace — where there is, in fact, a variable called `theoretical.percentiles`, which I set by hand for the previous plots. So the *plotted* theoretical quantiles are all too small in Figure 9.5, because they're for a distribution with a much lower threshold.

Didn't `check.rpareto` assign is own value to `theoretical.percentiles`, which it used to set the plot boundaries? Yes, but that assignment only applied *in the context of the function*. Assignments inside a function have limited **scope**, they leave values in the broader context alone. Try this:

```
> x <- 7
> x
[1] 7
> square <- function(y) { x <- y^2; return(x) }
> square(7)
[1] 49
> x
[1] 7
```

The function `square` assigns `x` to be the square of its argument. This assignment holds within the scope of the function, as we can see from the fact that the returned value is always the square of the argument, and not what we assigned `x` to be in the global, command-line context. However, this does not over-write that global value, as the last line shows.[5]

There are two ways to fix this problem. One is to re-define `pareto.sim.vs.theory` to calculate the theoretical quantiles:

```
pareto.sim.vs.theory <- function(n,exponent,threshold,...) {
  r <- rpareto(n=n,exponent=exponent,threshold=threshold)
  theoretical.percentiles <- qpareto.4((0:99)/100,exponent=exponent,
                                       threshold=threshold)
  simulated.percentiles <- quantile(r,(0:99)/100)
  points(theoretical.percentiles,simulated.percentiles,...)
}
```

This will work (try running `check.rpareto(1e4,2.33,9e8)` now), but it's very redundant — every time we call this, we're recalculating the same percentiles, which we already calculated in `check.rpareto`. A cleaner solution is to make the vector of theoretical percentiles an argument to `pareto.sim.vs.theory`, and change `check.rpareto` to provide it.

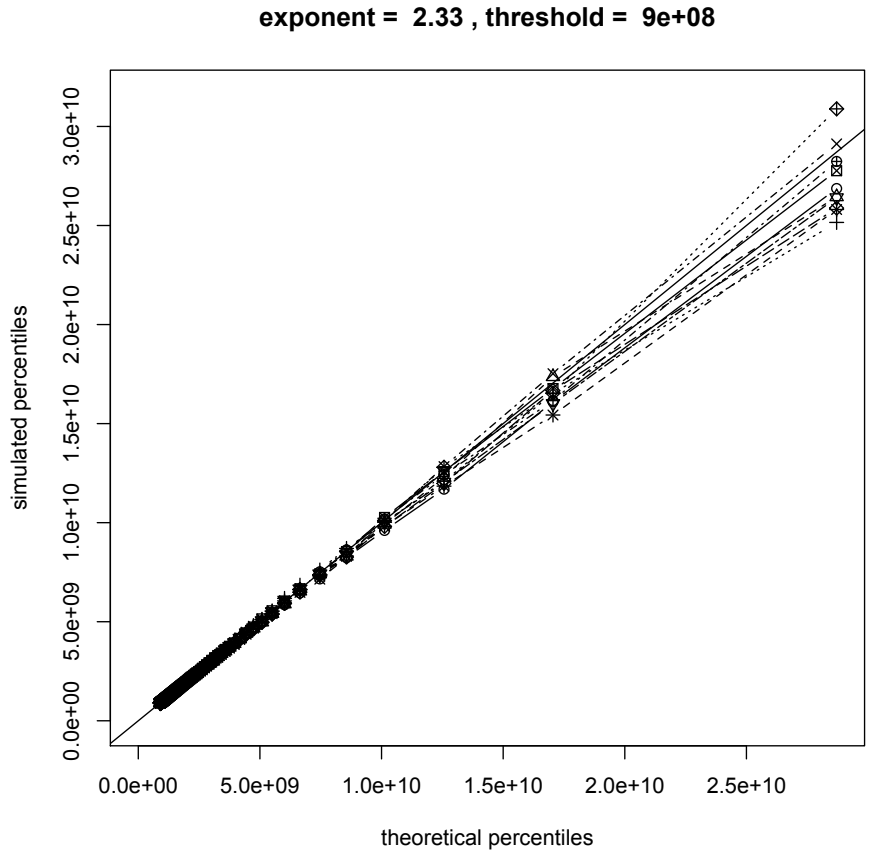
```
check.rpareto <- function(n=1e4,exponent=2.5,threshold=1,B=10) {
```

[5] There are techniques by which functions can change assignments outside of their scope. They are tricky, rare, and best avoided except by those who really know what they are doing. (If you think you do, you are probably wrong.)

```
  # One set of percentiles for everything
  theoretical.percentiles <- qpareto.4((0:99)/100,exponent=exponent,
    threshold=threshold)
  # Set up plotting window, but don't put anything in it:
  plot(0,type="n", xlim=c(0,max(theoretical.percentiles)),
    # No more horizontal room than we need
    ylim=c(0,1.1*max(theoretical.percentiles)),
    # Allow some extra vertical room for noise
    xlab="theoretical percentiles", ylab="simulated percentiles",
    main = paste("exponent = ", exponent, ", threshold = ", threshold))
  # Diagonal, for visual reference
  abline(0,1)
  for (i in 1:B) {
    pareto.sim.vs.theory.4(n=n,exponent=exponent,threshold=threshold,
      theoretical.percentiles=theoretical.percentiles,
      pch=i,type="b",lty=i)
  }
}

pareto.sim.vs.theory <- function(n,exponent,threshold,
  theoretical.percentiles,...) {
  r <- rpareto(n=n,exponent=exponent,threshold=threshold)
  simulated.percentiles <- quantile(r,(0:99)/100)
  points(theoretical.percentiles,simulated.percentiles,...)
}
```

Figure 9.6 shows that this succeeds.




```
check.rpareto(1e4,2.33,9e8)
```

Figure 9.6: Using the corrected simulation checker.

9.6 Avoiding Iteration: Manipulating Objects

Let's go back to the declaration of `rpareto`, which I repeat here, unchanged, for convenience:

```
rpareto <- function(n,exponent,threshold) {
  x <- vector(length=n)
  for (i in 1:n) {
    x[i] <- qpareto.4(p=runif(1),exponent=exponent,threshold=threshold)
  }
  return(x)
}
```

We've confirmed that this works, but it involves explicit iteration in the form of the `for` loop. Because of the way R carries out iteration[6], it is slow, and better avoided when possible. Many of the utility functions in R, like `replicate`, are designed to avoid explicit iteration. We could re-write `rpareto` using `replicate`, for example:

```
rpareto <- function(n,exponent,threshold) {
  x <- replicate(n,qpareto.4(p=runif(1),exponent=exponent,threshold=thresho
  return(x)
}
```

(The outstanding use of `replicate` is when we want to repeat the same random experiment many times — there are examples in the notes for Chapters 5.)

An every clearer alternative makes use of the way R automatically **vectorizes** arithmetic:

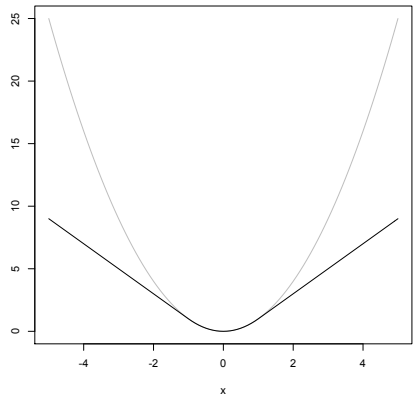
```
rpareto <- function(n,exponent,threshold) {
  x <- qpareto.4(p=runif(n),exponent=exponent,threshold=threshold)
  return(x)
}
```

This feeds `qpareto.4` a *vector* of quantiles `p`, of length `n`, which in turn gets passed along to `qpareto.1`, which finally tries to evaluate

```
threshold*((1-p)^(-1/(exponent-1)))
```

With `p` being a vector, R hopes that `threshold` and `exponent` are also vectors, and of the same length, so that it evaluate this arithmetic expression component-wise. If `exponent` and `threshold` are shorter, it will "recycle" their values, in order, until it has vectors equal in length to `p`. In particular, if `exponent` and `threshold` have length 1, it will repeat both of them `length(p)` times, and then evaluate everything component by component. (See the "Introduction to R" manual for more on this "recycling rule".) The quantile functions we have defined inherit this ability to recycle, without any special work on our part. The final version of `rpareto` we have written is not only faster, it is clearer and easier to read. It focuses our attention on what is being done, and not on the mechanics of doing it.

[6] Roughly speaking, it ends up having to create and destroy a whole copy of everything which gets changed in the course of one pass around the iteration loop, which can involve lots of memory and time.




```
curve(x^2,col="grey",from=-5,to=5,ylab="")
curve(huber,add=TRUE)
```

Figure 9.7: The Huber loss function ψ (black) versus the squared error loss (grey).

`ifelse` and `which`

Sometimes we want to do different things to different parts of a vector (or larger structure) depending on its values. For instance, in robust regression one often replaces the squared error loss with what's called the Huber loss[7],

$$\psi(x) = \begin{cases} x^2 & \text{if } |x| \leq 1 \\ 2|x| - 1 & \text{if } |x| > 1 \end{cases} \tag{9.4}$$

which isn't so vulnerable to outliers, as in Figure 9.7.

We might code this up like so:

```
huber <- function(x) {
  n <- length(x)
  y <- vector(n)
  for (i in 1:n) {
    if (abs(x) =< 1) {
      y[i] <- x[i]^2
    } else {
      y[i] <- 2*abs(x[i])-1
    }
  }
  return(y)
}
```

[7] One applies this not to the residuals directly, but to residuals divided by some robust measure of dispersion.

This is not very easy follow. R provides a very useful function, `ifelse`, which lets us apply a binary test, and then draw from either of two calculations. Using it, we re-write `huber` like so:

```
huber <- function(x) {
  return(ifelse(abs(x) =< 1, x^2, 2*abs(x)-1))
}
```

The first argument needs to produce a vector of TRUE/FALSE values; the second argument provides the outputs for the TRUE positions, the third the outputs for the FALSE positions. Here all three are expressions involving the same variable, but that's not essential.

Another useful device is the `which` function, whose argument is a vector of TRUE/FALSE values, returning a vector of the indices where the argument is TRUE, e.g.,

```
incomplete.cases <- which(is.na(cholesterol))
```

would give us the positions at which the vector `cholesterol` had NA values. This is equivalent to

```
incomplete.cases <- c()
for (i in 1:length(cholesterol)) {
  if (is.na(cholesterol[i])) {
    incomplete.cases <- c(incomplete.cases,i)
  }
}
```

9.6.1 `apply` and Its Variants

Particularly useful ways of avoiding iteration come from the function `apply`, and the closely related `sapply` and `lapply` functions. We saw `apply` in Chapter 5:

```
x <- replicate(10,rpareto(100,2.5,1))
apply(x,2,quantile,probs=0.9)
```

Each call to `rpareto` inside the `replicate` creates a vector of length 100. Replicate then stacks these, as columns, into an array. The `apply` function applies the same function to each row or column of the array, depending on whether its second argument is 1 (rows) or 2 (columns). So this will find the 90th percentile of each of the 10 random-number draws, and give that back to us as a vector.

`array` only works for arrays, matrices and data frames (and works on them by treating them as arrays). If we want to apply the same function to every element of a vector or list, we use `lapply`. This gives us back a list, which can be inconvenient:

```
> y <- c(0.9,0.99,0.999,0.99999)
> lapply(y,qpareto.4,exponent=2.5,threshold=1)
[[1]]
[1] 4.641589
```

```
[[2]]
[1] 21.54435

[[3]]
[1] 100

[[4]]
[1] 2154.435
```

The function `sapply` works like `lapply`, but tries to simplify its output down to a vector or array:

```
> sapply(y,qpareto.4,exponent=2.5,threshold=1)
[1]    4.641589   21.544347  100.000000 2154.434690
```

With this function, this is equivalent to `qpareto.4(y,exponent=2.5,threshold=1)`, but `sapply` can take considerably more complicated functions:

```
# Suppose we have models lm.1 and lm.2 hanging around
some.models <- list(model.1=lm.1, model.2=lm.2)
# Extract all the coefficients from all the models
sapply(some.models,coefficients)
```

`sapply` has a `simplify` argument, which defaults to TRUE; setting it to `FALSE` turns off the simplification. `replicate` actually has the same argument. Usually, simplifying the output of `replicate` is a good thing, but it can weirdness when what's being replicated is a complicated value itself.

For instance, here's a little bit of bootstrapping regression models, using the fossil-animal data set from homework 3.

```
resample <- function(x) { sample(x,size=length(x),replace=TRUE) }
nampd.lm.subset <- function(s) {
  lm(delta_ln_mass ~ ln_old_mass,data=nampd,subset=s)
}
boot.models.1 <- replicate(10,nampd.lm.subset(resample(1:nrow(nampd))))
```

Working with `boot.models.1` is going to be very hard, because it wants to be an array, but isn't quite, and is generally very confused. (Try it!) Instead do it this way:

```
boot.models.2 <- replicate(10,nampd.lm.subset(resample(1:nrow(nampd))),
  simplify=FALSE)
```

`boot.models.2` is simply a list with 10 elements, each one of which is an `lm`-style model. Now it's easy extract information about any particular one, or use `sapply`:

```
> sapply(boot.models.2,coefficients)
                   [,1]         [,2]         [,3]        [,4]
(Intercept)  0.21613522  0.092359537  0.184610989  0.15530334
```

```
ln_old_mass -0.01379554 -0.002729451 -0.007396701 -0.01078759
                    [,5]          [,6]         [,7]         [,8]
(Intercept)  0.124932040  0.115330144  0.192097575 0.0880172496
ln_old_mass -0.003754933 -0.007362125 -0.008486858 0.0008434435
                    [,9]         [,10]
(Intercept)  0.17065043  0.207331222
ln_old_mass -0.01430204 -0.009881709
```

9.7 More Complicated Return Values

So far, all the functions we have written have returned either a single value, or a simple vector, or nothing at all. The built-in functions return much more complicated things, like matrices, data frames, or lists, and we can too.

To illustrate, let's switch gears away from the Pareto distribution, and think about the Gaussian for a change. As you know, if we have data $x_1, x_2, \ldots x_n$ and we want to fit a Gaussian distribution to them by maximizing the likelihood, the best-fitting Gaussian has mean

$$\hat{\mu} = \frac{1}{n}\sum_{i=1}^{n} x_i \tag{9.5}$$

which is just the sample mean, and variance

$$\hat{\sigma}^2 = \frac{1}{n}\sum_{i=1}^{n} (x_i - \hat{\mu})^2 \tag{9.6}$$

which differs from the usual way of defining the sample variance by having a factor of n in the denominator, instead of $n-1$. Let's write a function which takes in a vector of data points and returns the maximum-likelihood parameter estimates for a Gaussian.

```
gaussian.mle <- function(x) {
  n <- length(x)
  mean.est <- mean(x)
  var.est <- var(x)*(n-1)/n
  est <- list(mean=mean.est, sd=sqrt(var.est))
  return(est)
}
```

There is one argument, which is the vector of data. To be cautious, I should probably check that it *is* a vector of numbers, but skip that to be clear here. The first line figures out how many data points we have. The second takes the mean. The third finds the estimated variance — the definition of the built-in `var` function uses $n-1$ in its denominator, so I scale it down by the appropriate factor[8]. The fourth line creates a list, called `est`, with two components, named `mean` and `sd`, since those

[8] Clearly, if n is large, $\frac{n-1}{n} = 1 - 1/n$ will be very close to one, but why not be precise?

are the names R likes to use for the parameters of Gaussians. The first component is our estimated mean, and the second is the standard deviation corresponding to our estimated variance[9]. Finally, the function returns the list.

As always, it's a good idea to check the function on a case where we know the answer.

```
> x <- 1:10
> mean(x)
[1] 5.5
> var(x) * (9/10)
[1] 8.25
> sqrt(var(x) * (9/10))
[1] 2.872281
> gaussian.mle(x)
$mean
[1] 5.5

$sd
[1] 2.872281
```

9.8 Re-Writing Your Code: An Extended Example

Suppose we want to find a standard error for the median of a Gaussian distribution. We know, somehow, that the mean of the Gaussian is 3, the standard deviation is 2, and the sample size is one hundred. If we do

```
x <- rnorm(n=100,mean=3,sd=2)
```

we'll get a draw from that distribution in `x`. If we do

```
x <- rnorm(n=100,mean=3,sd=2)
median(x)
```

we'll calculate the median on one random draw. Following the general idea of bootstrapping we can approximate the standard error of the median by repeating this many times and taking the standard deviation. We'll do this by explicitly iterating, so we need to set up a vector to store our intermediate results first.

```
medians <- vector(length=100)
for (i in 1:100) {
  x <- rnorm(n=100,mean=3,sd=2)
  medians[i] <- median(x)
}
se.in.median <- sd(medians)
```

[9] If n is large, $\sqrt{\frac{n-1}{n}} = \sqrt{1-\frac{1}{n}} \approx 1 - \frac{1}{2n}$ (using the binomial theorem in the last step). For reasonable data sets, the error of just using `sd(x)` would have been small — but why have it at all?

Well, how do we know that 100 replicates is enough to get a good approximation? We'd need to run this a couple of times, typing it in or at least pasting it in many times. Instead, we can write a function which just gives everything we've done a single name. (I'll add comments as I go on.)

```
# Inputs: None; everything is hard-coded
# Output: the standard error in the median
find.se.in.median <- function() {
  # Set up a vector to store the simulated medians
  medians <- vector(length=100)
  # Do the simulation 100 times
  for (i in 1:100) {
    x <- rnorm(n=100,mean=3,sd=2) # Simulate
    medians[i] <- median(x) # Calculate the median of the simulation
  }
  se.in.median <- sd(medians) # Take standard deviation
  return(se.in.median)
}
```

If we decide that 100 replicates isn't enough and we want 1000, we need to change this function. We could just change the first two appearances of "100" to "1000", but we have to catch all of them; we have to remember that the 100 in `rnorm` is there for a different reason and leave it alone; and if we later decide that actually 500 replicates would be enough, we have to do everything all over again.

It is easier, safer, clearer and more flexible to abstract a little and add an argument to the function, which is the number of replicates. I'll add comments as I go.

```
# Inputs: Number of bootstrap replicates B
# Output: the standard error in the median
find.se.in.median <- function(B) {
  # Set up a vector to store the simulated medians
  medians <- vector(length=B)
  # Do the simulation B times
  for (i in 1:B) {
    x <- rnorm(n=100,mean=3,sd=2) # Simulate
    medians[i] <- median(x) # Calculate median of the simulation
  }
  se.in.median <- sd(medians) # Take standard deviation
  return(se.in.median)
}
```

Now suppose we want to find the standard error of the median for an exponential distribution with rate 2 and sample size 37. We could write another function,

```
find.se.in.median.exp <- function(B) {
  # Set up a vector to store the simulated medians
  medians <- vector(length=B)
  # Do the simulation B times
```

```
  for (i in 1:B) {
    x <- rexp(n=37,rate=2) # Simulate
    medians[i] <- median(x) # Calculate median of the simulation
  }
  se.in.median <- sd(medians) # Take standard deviation
  return(se.in.median)
}
```

but it is wasteful to define two functions which do almost the same job. It's not just inelegant; it invites mistakes, it's harder to read (imagine coming back to this in two weeks — was there a big reason why we had two separate functions here?), and it's harder to improve. We need to abstract a bit more.

We *could* put in some kind of switch which would simulate from either of these two distributions, maybe like this:

```
# Inputs: number of replicates (B)
  # flag for whether to use a normal or an exponential (use.norm)
# Output: The standard error in the median
find.se.in.median <- function(B,use.norm=TRUE) {
  medians <- vector(length=B)
  for (i in 1:B) {
    if (use.norm) {
      x <- rnorm(100,3,2)
    } else {
      x <- rexp(37,2)
    }
    medians[i] <- median(x)
  }
  se.in.median <- sd(medians)
  return(se.in.median)
}
```

but why just these two? If we wanted any other distribution whatsoever, plainly all we'd have to do is change how `x` is simulated. So we really want to be able to *give* a simulator to the function as an argument.

Fortunately, in R you can give one function as an argument to another, so we'd do something like this.

```
# Inputs: Number of replicates (B)
  # Simulator function (simulator)
# Presumes: simulator is a no-argument function which produce a vector of
  # numbers
# Output: The standard error in the media
find.se.in.median <- function(B,simulator) {
  median <- vector(length=B)
  for (i in 1:B) {
    x <- simulator()
```

```
    medians[i[ <- median(x)
  }
  se.in.median <- sd(medians)
  return(se.in.medians)
}
```

Now to repeat our original calculations, we define a simulator function:

```
# Inputs: None
# Output: ten draws from the mean 3, s.d. 2 Gaussian
simulator.1 <- function() {
  return(rnorm(10,3,2))
}
```

If we now call

```
find.se.in.median(B=100,simulator=simulator.1)
```

then every time `find.se.in.median` goes through the `for` loop, it will call `simulator.1`, which in turn will produce the right random numbers. If we also define

```
# Inputs: None
# Output: 37 draws from the rate 2 exponential
simulator.2 <- function() {
  return(rexp(37,2))
}
```

then to find the standard error in the median of *this*, we just call

```
find.se.in.median(B=100,simulator=simulator.2)
```

This same approach works if we want to sample from a much more complicated distribution. If we fit a locally-linear kernel regression to the Old Faithful data, and want a standard error in the median of the predicted waiting times, with noise coming from resampling cases, we would do something like this for the simulator

```
# Inputs: None
# Output: The fitted waiting times of a bootstrapped kernel smooth from the
  # geyser data
simulator.3 <- function() {
  if (!exists("geyser")) {
    require(MASS)
    data(geyser)
  }
  n <- nrow(geyser)
  resampled.rows <- sample(1:n,size=n,replace=TRUE)
  geyser.r <- geyser[resampled.rows,]
  fit <- npreg(waiting~duration,data=geyser.r,regtype="ll")
  waiting.times <- npreg$mean
  return(waiting.times)
}
```

and then this for to find the standard error in the median:

```
find.se.in.median(B=100,simulator=simulator.3)
```

By breaking up the task this way, if we encounter errors or just general trouble when we run that last command, it is easier to localize the problem. We can check whether `find.se.in.median` seems to work properly with other simulator functions. (For instance, we might write a "simulator" that either does `rep(10,1)` or `rep(10,-1)` with equal probability, since then we can work out what the standard error of the median ought to be.) We can also check whether `simulator.3` is working properly, and finally whether there is some issue with putting them together, say that the output from the simulator is not quite in a format that `find.se.in.median` can handle. If we just have one big ball of code, it is much harder to read, to understand, to debug, and to improe.

To turn to that last point, one of the things R does poorly is explicit iteration with `for` loops. As mentioned above, it's generally better to replace such loops with "vectorized" functions, which do the iteration using fast code outside of R. One of these, especially for this situation, is the function `replicate`. We can re-write `find.se.in.median` using it:

```
# Inputs: number of replicates (B)
  # Simulator function (simulator)
# Presumes: simulator is a no-argument function which produces a vector of
  # numbers
# Outputs: Standard error in the median of the output of simulator
find.se.in.median <- function(B,simulator) {
  medians <- replicate(B,median(simulator()))
  se.in.median <- sd(medians)
  return(se.in.median)
}
```

Again: shorter, faster, and easier to understand (if you know what `replicate` does). Also, because we are telling this what simulation function to use, and writing those functions separately, we do not have to change any of our simulators. They don't *care* how `find.se.in.median` works. In fact, they don't care that there is any such function — they could be used as components in many other functions which can also process their outputs. So long as these *interfaces* are maintained, the inner workings of the functions are irrelevant to each other.

Suppose for instance that we want not the standard error of the median, but the interquartile range of the median — the median is after all a "robust", outlier-resistant measure of the central tendency, and the IQR is likewise a robust measure of dispersion. This is now easy:

```
# Inputs: number of replicates (B)
  # Simulator function (simulator)
# Presumes: simulator is a no-argument function which produces a vector of
  # numbers
# Outputs: Interquartile range of the median of the output of simulator
```

```
find.iqr.of.median <- function(B,simulator) {
  medians <- replicate(B,median(simulator()))
  iqr.of.median <- IQR(medians)
  return(iqr.of.median)
}
```

Or for that matter the good old standard error of the mean:

```
# Inputs: number of replicates (B)
  # Simulator function (simulator)
# Presumes: simulator is a no-argument function which produces a vector of
  # numbers
# Outputs: Standard error of the mean of the output of simulator
find.se.of.mean <- function(B,simulator) {
  means <- replicate(B,mean(simulator()))
  se.of.mean <- sd(means)
  return(se.of.mean)
}
```

These last few examples suggest that we could abstract even further, by swapping in and out different estimators (like `median` and `mean`) and different summarizing functions (like `se` or `IQR`).

```
# Inputs: number of replicates (B)
  # Simulator function (simulator)
  # Estimator function (estimator)
  # Sample summarizer function (summarizer)
# Presumes: simulator is a no-argument function which produces a vector of
  # numbers
  # estimator is a function that takes a vector of numbers and produces one
  # output
  # summarizer takes a vector of outputs from estimator
# Outputs: Summary of the simulated distribution of estimates
summarize.sampling.dist.of.estimates <- function(B,simulator,estimator,
                                                 summarizer) {
  estimates <- replicate(B,estimator(simulator()))
  return(summarizer(estimates))
}
```

The name is too long, of course, so we should replace it with something catchier:

```
bootstrap <- function(B,simulator,estimator,summarizer) {
  estimates <- replicate(B,estimator(simulator()))
  return(summarizer(estimates))
}
```

Our very first example is equivalent to

```
bootstrap(B=100,simulator=simulator.1,estimator=median,summarizer=sd)
```

`bootstrap` is just two lines: one simulates and re-estimates, the other summarizes the re-estimates. This is the essence of what we are trying to do, and is logically distinct from the details of particular simulators, estimators and summaries.

We started with a particular special case and generalized it. The alternative route is to start with a very general framework — here, writing `bootstrap` — and then figure out what lower-level functions we would need to make it work in a the case at hand, writing them if necessary. (We need to write a simulator, but someone's already written `median` for us.) Getting the first stage right involves a certain amount of reflection on how to solve the problem — it's rather like the strategy of doing a "show that" math problem by starting from the desired conclusion and working backwards.

It is still somewhat clunky to have to write a new function every time we want to change the settings in the simulation, but this has gone on long enough.

9.9 General Advice on Programming

Programming is an act of communication: with the computer, of course, but also with your co-workers, and with yourself in the future[10]. Clear and effective communication is a valuable skill in itself; it also tends to make it easier to do the job, and to make debugging easier.

9.9.1 Comment your code

Comments lengthen your file, but they make it immensely easier for other people to understand. ("Other people" includes your future self; there are few experiences more frustrating than coming back to a program after a break only to wonder what you were thinking.) Comments should say what each part of the code does, and how it does it. The "what" is more important; you can change the "how" more often and more easily.

Every function (or subroutine, etc.) should have comments at the beginning saying:

- what it does;
- what all its inputs are (in order);
- what it requires of the inputs and the state of the system ("presumes");
- what side-effects it may have (e.g., "plots histogram of residuals");
- what all its outputs are (in order)

Listing what other functions or routines the function calls ("dependencies") is optional; this can be useful, but it's easy to let it get out of date.

You should treat "Thou shalt comment thy code" as a commandment which Moses brought down from Mt. Sinai, written on stone by a fiery Hand.

[10] And, in this class, with your graders.

9.9.2 Use meaningful names

Unlike some older languages, R lets you give variables and functions names of essentially arbitrary length and form. So give them meaningful names. Writing `loglikelihood`, or even `loglike`, instead of `L` makes your code a little longer, but generally a lot clearer, and it runs just the same.

This rule is lower down in the list because there are exceptions and qualifications. If your code is tightly associated to a mathematical paper, or to a field where certain symbols are conventionally bound to certain variables, you may as well use those names (e.g., call the probability of success in a binomial `p`). You should, however, explain what those symbols are in your comments. In fact, since what you regard as a meaningful name may be obscure to others (e.g., those grading your work), you should use comments to explain variables in any case. Finally, it's OK to use single-letter variable names for counters in loops (but see the advice on iteration below).

9.9.3 Check whether your program works

It's not a enough — in fact it's very little — to have a program which runs and gives you some output. It needs to be the right output. You should therefore construct tests, which are things that the correct program should be able to do, but an incorrect program should not. This means that:

- you need to be able to check whether the output is right;
- your tests should be reasonably severe, so that it's hard for an incorrect program to pass them;
- your tests should help you figure out what isn't working;
- you should think hard about programming the test, so it checks whether the output is right, and you can easily repeat the test as many times as you need.

Try to write tests for the component functions, as well as the program as a whole. That way you can see where failures are. Also, it's easier to figure out what the right answers should be for small parts of the problem than the whole.

Try to write tests as very small function which call the component you're testing with controlled input values. For instance, we tested `qpareto` by looking at what it returned for selected arguments with manually carrying out the computation. With statistical procedures, tests can look at average or distributional results — we saw an example of this with checking `rpareto`.

Of course, unless you are very clever, or the problem is very simple, a program could pass all your tests and still be wrong, but a program which fails your tests is definitely not right.

(Some people would actually advise writing your tests before writing any actual functions. They have their reasons but I think that's overkill for this class.)

9.9.4 Avoid writing the same thing twice

Many data-analysis tasks involve doing the same thing multiple times, either as iteration, or to slightly different pieces of data, or with some parameters adjusted, etc. Try to avoid writing two pieces of code to do the same job. If you find yourself copying the same piece of code into two places in your program, look into writing *one* function, and *calling* it twice.

Doing this means that there is only one place to make a mistake, rather than many. It also means that when you fix your mistake, you only have one piece of code to correct, rather than many. (Even if you don't make a mistake, you can always make improvements, and then there's only one piece of code you have to work on.) It also leads to shorter, more comprehensible and more adaptable code.

9.9.5 Start from the beginning and break it down

When you have a big problem, start by thinking about what you want your program to do. Then figure out a set of slightly smaller steps which, put together, would accomplish that. Then take each of those steps and break them down into yet smaller ones. Keep going until the pieces you're left with are so small that you can see how to do each of them with only a few lines of code. Then write the code for the smallest bits, check it, once it works write the code for the next larger bits, and so on.

In slogan form:

- Think before you write.
- What first, then how.
- Design from the top down, code from the bottom up.

(Not everyone likes to design code this way, and it's not in the written-in-stone-atop-Sinai category, but there are many much worse ways to start.)

9.9.6 Break your code into many short, meaningful functions

Since you have broken your programming problem into many small pieces, try to make each piece a short function. (In other languages you might make them subroutines or methods, but in R they should be functions.)

Each function should achieve a single coherent task — its function, if you will. The division of code into functions should respect this division of the problem into sub-problems. More exactly, the way you break your code into functions is how you have divided your problem.

Each function should be short, generally less than a page of print-out. The function should do one single meaningful thing. (Do not just break the calculation into arbitrary thirty-line chunks and call each one a function.) These functions should generally be separate, not nested one inside the other.

Using functions has many advantages:

- you can re-use the same code many times, either at different places in this program or in other programs

- the rest of your code only has to care about the inputs and outputs to the function (its interfaces), not about the internal machinery that turns inputs into outputs. This makes it easier to design the rest of the program, and it means you can change that machinery without having to re-design the rest of the program.
- it makes your code easier to test (see below), to debug, and to understand.

Of course, every function should be commented, as described above.

9.10 Further Reading

Matloff (2011) is a good introduction to programming for total novices using R. Braun and Murdoch (2008) has more on statistical calculations and related topics, but can also work as an introduction for absolute beginners. Adler (2009) is an introduction to R for those with some prior knowledge of other programming languages. Chambers (2008) is excellent for anyone who wants to be serious about programming in R.