Circle 3

Failing to Vectorize

We arrive at the third Circle, filled with cold, unending rain. Here stands Cerberus barking out of his three throats. Within the Circle were the blasphemous wearing golden, dazzling cloaks that inside were all of lead—weighing them down for all of eternity. This is where Virgil said to me, "Remember your science—the more perfect a thing, the more its pain or pleasure."

Here is some sample code:

```
lsum <- 0
for(i in 1:length(x)) {
   lsum <- lsum + log(x[i])
}
```

No. No. No.

This is speaking R with a C accent—a strong accent. We can do the same thing much simpler:

```
lsum <- sum(log(x))
```

This is not only nicer for your carpal tunnel, it is computationally much faster. (As an added bonus it avoids the bug in the loop when `x` has length zero.)

The command above works because of vectorization. The `log` function is vectorized in the traditional sense—it does the same operation on a vector of values as it would do on each single value. That is, the command:

```
log(c(23, 67.1))
```

has the same result as the command:

```
c(log(23), log(67.1))
```

The `sum` function is vectorized in a quite different sense—it takes a vector and produces something based on the whole vector. The command `sum(x)` is equivalent to:

```
x[1] + x[2] + ... + x[length(x)]
```

The `prod` function is similar to `sum`, but does products rather than sums. Products can often overflow or underflow (a suburb of Circle 1)—taking logs and doing sums is generally a more stable computation.

You often get vectorization for free. Take the example of `quadratic.formula` in Circle 1 (page 9). Since the arithmetic operators are vectorized, the result of this function is a vector if any or all of the inputs are. The only slight problem is that there are two answers per input, so the call to `cbind` is used to keep track of the pairs of answers.

In binary operations such as:

```
c(1,4) + 1:10
```

recycling automatically happens along with the vectorization.

Here is some code that combines both this Circle and Circle 2 (page 12):

```
ans <- NULL
for(i in 1:507980) {
   if(x[i] < 0) ans <- c(ans, y[i])
}
```

This can be done simply with:

```
ans <- y[x < 0]
```

A double `for` loop is often the result of a function that has been directly translated from another language. Translations that are essentially verbatim are unlikely to be the best thing to do. Better is to rethink what is happening with R in mind. Using direct translations from another language may well leave you longing for that other language. Making good translations may well leave you marvelling at R's strengths. (The catch is that you need to know the strengths in order to make the good translations.)

If you are translating code into R that has a double `for` loop, think.

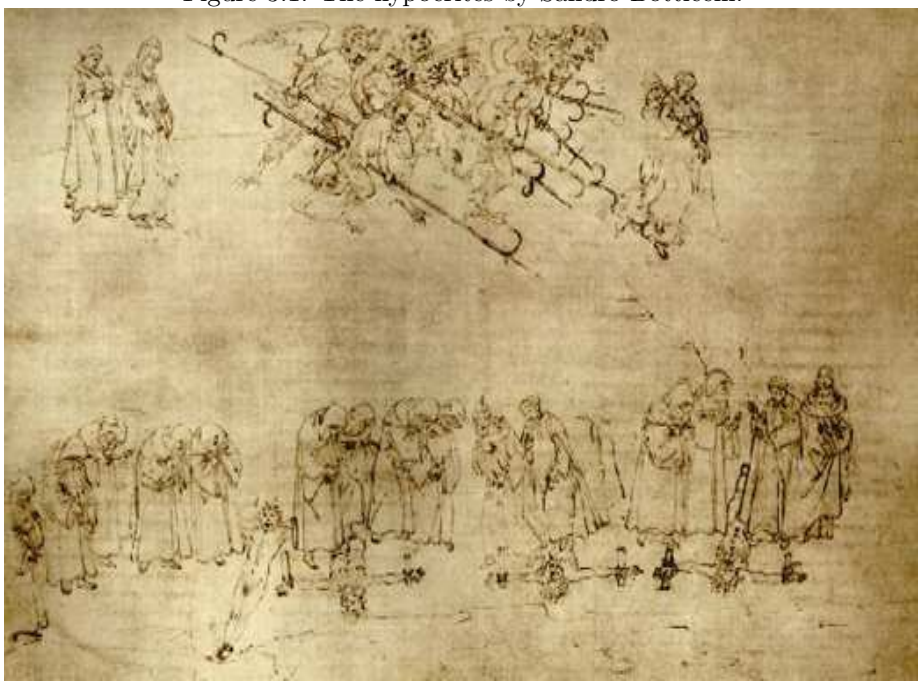
If your function is not vectorized, then you can possibly use the `Vectorize` function to make a vectorized version. But this is vectorization from an external point of view—it is not the same as writing inherently vectorized code. The `Vectorize` function performs a loop using the original function.

Some functions take a function as an argument and demand that the function be vectorized—these include `outer` and `integrate`.

There is another form of vectorization:

```
> max(2, 100, -4, 3, 230, 5)
[1] 230
> range(2, 100, -4, 3, 230, 5, c(4, -456, 9))
[1] -456 230
```

Figure 3.1: The hypocrites by Sandro Botticelli.



This form of vectorization is to treat the collection of arguments as the vector. This is NOT a form of vectorization you should expect, it is essentially foreign to R—`min`, `max`, `range`, `sum` and `prod` are rare exceptions. In particular, `mean` does not adhere to this form of vectorization, and unfortunately does not generate an error from trying it:

```
> mean(2, -100, -4, 3, -230, 5)
[1] 2
```

But you get the correct answer if you add three (particular) keystrokes:

```
> mean(c(2, -100, -4, 3, -230, 5))
[1] -54
```

One reason for vectorization is for computational speed. In a vector operation there is always a loop. If the loop is done in C code, then it will be much faster than if it is done in R code. In some cases, this can be very important. In other cases, it isn't—a loop in R code now is as fast as the same loop in C on a computer from a few years ago.

Another reason to vectorize is for clarity. The command:

```
volume <- width * depth * height
```

Table 3.1: Summary of subscripting with `[`.

subscript	effect
positive numeric vector	selects items with those indices
negative numeric vector	selects all but those indices
character vector	selects items with those names (or dimnames)
logical vector	selects the `TRUE` (and `NA`) items
missing	selects all

clearly expresses the relation between the variables. This same clarity is present whether there is one item or a million. Transparent code is an important form of efficiency. Computer time is cheap, human time (and frustration) is expensive. This fact is enshrined in the maxim of Uwe Ligges.

Uwe's Maxim Computers are cheap, and thinking hurts.

A fairly common question from new users is: "How do I assign names to a group of similar objects?" Yes, you can do that, but you probably don't want to—better is to vectorize your thinking. Put all of the similar objects into one list. Subsequent analysis and manipulation is then going to be much smoother.

3.1 Subscripting

Subscripting in R is extremely powerful, and is often a key part of effective vectorization. Table 3.1 summarizes subscripting.

The dimensions of arrays and data frames are subscripted independently.

Arrays (including matrices) can be subscripted with a matrix of positive numbers. The subscripting matrix has as many columns as there are dimensions in the array—so two columns for a matrix. The result is a vector (not an array) containing the selected items.

Lists are subscripted just like (other) vectors. However, there are two forms of subscripting that are particular to lists: `$` and `[[`. These are almost the same, the difference is that `$` expects a name rather than a character string.

```
> mylist <- list(aaa=1:5, bbb=letters)
> mylist$aaa
[1] 1 2 3 4 5
> mylist[['aaa']]
[1] 1 2 3 4 5
> subv <- 'aaa'; mylist[[subv]]
[1] 1 2 3 4 5
```

You shouldn't be too surprised that I just lied to you. Subscripting with `[[` can be done on atomic vectors as well as lists. It can be the safer option when

a single item is demanded. If you are using `[[` and you want more than one item, you are going to be disappointed.

We've already seen (in the `lsum` example) that subscripting can be a symptom of not vectorizing.

As an example of how subscripting can be a vectorization tool, consider the following problem: We have a matrix `amat` and we want to produce a new matrix with half as many rows where each row of the new matrix is the product of two consecutive rows of `amat`.

It is quite simple to create a loop to do this:

```
bmat <- matrix(NA, nrow(amat)/2, ncol(amat))
for(i in 1:nrow(bmat)) bmat[i,] <- amat[2*i-1,] * amat[2*i,]
```

Note that we have avoided Circle 2 (page 12) by preallocating `bmat`.

Later iterations do not depend on earlier ones, so there is hope that we can eliminate the loop. Subscripting is the key to the elimination:

```
> bmat2 <- amat[seq(1, nrow(amat), by=2),] *
+     amat[seq(2, nrow(amat), by=2),]
> all.equal(bmat, bmat2)
[1] TRUE
```

3.2 Vectorized if

Here is some code:

```
if(x < 1) y <- -1 else y <- 1
```

This looks perfectly logical. And if `x` has length one, then it does as expected. However, if `x` has length greater than one, then a warning is issued (often ignored by the user), and the result is not what is most likely intended. Code that fulfills the common expectation is:

```
y <- ifelse(x < 1, -1, 1)
```

Another approach—assuming `x` is never exactly 1—is:

```
y <- sign(x - 1)
```

This provides a couple of lessons:

1. The condition in `if` is one of the few places in R where a vector (of length greater than 1) is not welcome (`the` `:` operator is another).
2. `ifelse` is what you want in such a situation (though, as in this case, there are often more direct approaches).

Recall that in Circle 2 (page 12) we saw:

```
hit <- NA
for(i in 1:one.zillion) {
    if(runif(1) < 0.3) hit[i] <- TRUE
}
```

One alternative to make this operation efficient is:

```
ifelse(runif(one.zillion) < 0.3, TRUE, NA)
```

If there is a mistake between `if` and `ifelse`, it is almost always trying to use `if` when `ifelse` is appropriate. But ingenuity knows no bounds, so it is also possible to try to use `ifelse` when `if` is appropriate. For example:

```
ifelse(x, character(0), '')
```

The result of `ifelse` is ALWAYS the length of its first (formal) argument. Assuming that `x` is of length 1, the way to get the intended behavior is:

```
if(x) character(0) else ''
```

Some more caution is warranted with `ifelse`: the result gets not only its length from the first argument, but also its attributes. If you would like the answer to have attributes of the other two arguments, you need to do more work. In Circle 8.2.7 we'll see a particular instance of this with factors.

3.3 Vectorization impossible

Some things are not possible to vectorize. For instance, if the present iteration depends on results from the previous iteration, then vectorization is usually not possible. (But some cases are covered by `filter`, `cumsum`, etc.)

If you need to use a loop, then make it lean:

- Put as much outside of loops as possible. One example: if the same or a similar sequence is created in each iteration, then create the sequence first and reuse it. Creating a sequence is quite fast, but appreciable time can accumulate if it is done thousands or millions of times.
- Make the number of iterations as small as possible. If you have the choice of iterating over the elements of a factor or iterating over the levels of the factor, then iterating over the levels is going to be better (almost surely).

The following bit of code gets the sum of each column of a matrix (assuming the number of columns is positive):

```
sumxcol <- numeric(ncol(x))
for(i in 1:ncol(x)) sumxcol[i] <- sum(x[,i])
```

A more common approach to this would be:

```
sumxcol <- apply(x, 2, sum)
```

Since this is a quite common operation, there is a special function for doing this that does not involve a loop in R code:

```
sumxcol <- colSums(x)
```

There are also `rowSums`, `colMeans` and `rowMeans`.

Another approach is:

```
sumxcol <- rep(1, nrow(x)) %*% x
```

That is, using matrix multiplication. With a little ingenuity a lot of problems can be cast into a matrix multiplication form. This is generally quite efficient relative to alternatives.