Chapter 32: Speeding up tough-to-vectorize code

Section 32.1: Speeding tough-to-vectorize for loops with Rcpp

Consider the following tough-to-vectorize for loop, which creates a vector of length `len` where the first element is specified (`first`) and each element `x_i` is equal to `cos(x_{i-1} + 1)`:

```
repeatedCosPlusOne <- function(first, len) {
  x <- numeric(len)
  x[1] <- first
  for (i in 2:len) {
    x[i] <- cos(x[i-1] + 1)
  }
  return(x)
}
```

This code involves a for loop with a fast operation (`cos(x[i-1]+1)`), which often benefit from vectorization. However, it is not trivial to vectorize this operation with base R, since R does not have a "cumulative cosine of x+1" function.

One possible approach to speeding this function would be to implement it in C++, using the Rcpp package:

```
library(Rcpp)
cppFunction("NumericVector repeatedCosPlusOneRcpp(double first, int len) {
  NumericVector x(len);
  x[0] = first;
  for (int i=1; i < len; ++i) {
    x[i] = cos(x[i-1]+1);
  }
  return x;
}")
```

This often provides significant speedups for large computations while yielding the exact same results:

```
all.equal(repeatedCosPlusOne(1, 1e6), repeatedCosPlusOneRcpp(1, 1e6))
# [1] TRUE
system.time(repeatedCosPlusOne(1, 1e6))
#    user  system elapsed
#   1.274   0.015   1.310
system.time(repeatedCosPlusOneRcpp(1, 1e6))
#    user  system elapsed
#   0.028   0.001   0.030
```

In this case, the Rcpp code generates a vector of length 1 million in 0.03 seconds instead of 1.31 seconds with the base R approach.

Section 32.2: Speeding tough-to-vectorize for loops by byte compiling

Following the Rcpp example in this documentation entry, consider the following tough-to-vectorize function, which creates a vector of length `len` where the first element is specified (`first`) and each element `x_i` is equal to `cos(x_{i-1} + 1)`:

```
repeatedCosPlusOne <- function(first, len) {
```

```
  x <- numeric(len)
  x[1] <- first
  for (i in 2:len) {
    x[i] <- cos(x[i-1] + 1)
  }
  return(x)
}
```

One simple approach to speeding up such a function without rewriting a single line of code is byte compiling the code using the R compile package:

```
library(compiler)
repeatedCosPlusOneCompiled <- cmpfun(repeatedCosPlusOne)
```

The resulting function will often be significantly faster while still returning the same results:

```
all.equal(repeatedCosPlusOne(1, 1e6), repeatedCosPlusOneCompiled(1, 1e6))
# [1] TRUE
system.time(repeatedCosPlusOne(1, 1e6))
#    user  system elapsed
#   1.175   0.014   1.201
system.time(repeatedCosPlusOneCompiled(1, 1e6))
#    user  system elapsed
#   0.339   0.002   0.341
```

In this case, byte compiling sped up the tough-to-vectorize operation on a vector of length 1 million from 1.20 seconds to 0.34 seconds.

Remark

The essence of `repeatedCosPlusOne`, as the cumulative application of a single function, can be expressed more transparently with `Reduce`:

```
iterFunc <- function(init, n, func) {
  funcs <- replicate(n, func)
  Reduce(function(., f) f(.), funcs, init = init, accumulate = TRUE)
}
repeatedCosPlusOne_vec <- function(first, len) {
  iterFunc(first, len - 1, function(.) cos(. + 1))
}
```

`repeatedCosPlusOne_vec` may be regarded as a "vectorization" of `repeatedCosPlusOne`. However, it can be expected to be *slower* by a factor of 2:

```
library(microbenchmark)
microbenchmark(
  repeatedCosPlusOne(1, 1e4),
  repeatedCosPlusOne_vec(1, 1e4)
)
#> Unit: milliseconds
#>                              expr       min        lq     mean   median       uq      max neval
cld
#>      repeatedCosPlusOne(1, 10000)  8.349261  9.216724 10.22715 10.23095 11.10817 14.33763   100
 a
#>  repeatedCosPlusOne_vec(1, 10000) 14.406291 16.236153 17.55571 17.22295 18.59085 24.37059   100
  b
```