Vectorization and Monte Carlo Estimation
Statistics 506

Vectorization

Vectorization is a programming technique used to avoid explicit loops in order to improve the performance and readability of code.

In its simplest form, vectorization means using built in vectorized functions. That is, rather than looping over a vector to compute its sum, we instead use the vectorized function `sum()`:

```
## Not vectorized
x <- 0:500

s0 <- 0
for (i in seq_along(x)) {
  s0 <- s0 + x[i]
}

## Vectorized
s1 <- sum(x)

## Do we get the same sum?
s0 == s1
```

```
[1] TRUE
```

The vectorized code is not only easier to type and read but is also faster.

```
# Library for timing comparison
library(microbenchmark)

x <- 1:10000
```

```r
microbenchmark(a = sum(x),
               b = {
                 s0 <- 0
                 for (i in seq_along(x)) {
                   s0 <- s0 + x[i]
                 }
               })
```

```
Unit: nanoseconds
 expr    min     lq      mean median     uq     max neval
    a     41     41    558.83     82    123   10127   100
    b 779697 795687 850271.12 810324 887978 1721385   100
```

The reason vectorization is faster has to do with the fact that *R* is an *interpreted* language and not a *compiled* one. The difference in time seen here is primarily due to two factors:

1. *R* needs to repeatedly *interpret* what your code means for each iteration of the loop;
2. each iteration of the `for` loop requires indexing into `x` using the subset function `[`.

The vectorized `sum()` function also loops over and repeatedly indexes `x` but it does so in the compiled language C and has been optimized to take advantage of the fact that the elements of a vector are contiguous in memory.

Writing efficient R code requires an understanding of these factors and how vectorization helps to overcome them.

The `apply` family of functions

Before we get any deeper into vectorization, there's a family of functions - `apply`, `lapply`, `sapply`, `vapply`, `tapply`, `mapply` - which commonly get referred to as vectorized. Let's explore what they do, then we'll discuss how they aren't actually vectorized.

`apply`

`apply` operates on matrices. It performs a function on each row or column of the matrix. The second argument, `MARGIN`, defines what dimension it operates on.

```r
mat <- matrix(sample(1:100, 24), ncol = 6)
apply(mat, 1, max)
```

```
[1]  89 100  93  98
```

```r
apply(mat, 2, min)
```

```
[1]  9 44 14  1  3 10
```

You can also define your own anonymous function.

```r
# Obtain the second lowest value in each column
apply(mat, 2, function(x) {
  z <- sort(x)
  return(z[2])
})
```

```
[1] 48 57 29 19 28 27
```

It also works for higher dimensional arrays.

```r
dim(mat) <- c(2, 4, 3)
mat
```

```
, , 1

     [,1] [,2] [,3] [,4]
[1,]    9   52   57   82
[2,]   48   98   44   92

, , 2

     [,1] [,2] [,3] [,4]
[1,]   89   29    1   19
[2,]   34   14  100   31

, , 3
```

```
     [,1] [,2] [,3] [,4]
[1,]   62    3   27   93
[2,]   74   28   10   70
```

```
apply(mat, 1, max) # row
```

```
[1]  93 100
```

```
apply(mat, 2, max) # column
```

```
[1]  89  98 100  93
```

```
apply(mat, 3, max) # 3rd dimension
```

```
[1]  98 100  93
```

```
apply(mat, c(1, 2), max) # row and column
```

```
     [,1] [,2] [,3] [,4]
[1,]   89   52   57   93
[2,]   74   98  100   92
```

Finally, you can pass additional arguments into the function.

```
mat <- matrix(c(5, 2, 3, NA), nrow = 2)
apply(mat, 1, sum)
```

```
[1]  8 NA
```

```
apply(mat, 1, sum, na.rm = TRUE)
```

```
[1] 8 2
```

Technically these are placed inside the `...` argument of `apply` (see `help(apply)`).

lapply, sapply, and vapply

These three functions operate the same, they differ only on how they return their result. Each operates on a list, applying a function to each element.

```r
li <- list(5, letters[1:4], head(mtcars))
length(li)
```

```
[1] 3
```

```r
lapply(li, length)
```

```
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 11
```

Again, we can use anonymous functions.

```r
lapply(li, function(x) {
  if (is.numeric(x)) {
    return(mean(x))
  } else {
    return("Not numeric")
  }
})
```

```
[[1]]
[1] 5

[[2]]
[1] "Not numeric"

[[3]]
[1] "Not numeric"
```

`sapply` attempts to simplify the result.

```r
lapply(li, length)
```

```
[[1]]
[1] 1

[[2]]
[1] 4

[[3]]
[1] 11
```

```r
sapply(li, length)
```

```
[1]  1  4 11
```

Since the list returned from `lapply` are all scalars, it placed them in a single vector.

```r
lapply(list(mtcars, quakes), dim)
```

```
[[1]]
[1] 32 11

[[2]]
[1] 1000    5
```

```r
sapply(list(mtcars, quakes), dim)
```

```
     [,1] [,2]
[1,]   32 1000
[2,]   11    5
```

`vapply` again operates the same, but requires you to define what the output will be. It does so by taking in an "example" of the output.

```r
vapply(li, is.numeric, logical(1))
```

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

```
vapply(li, is.numeric, TRUE)
```

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

```
vapply(list(mtcars, quakes), nrow, integer(1))
```

```
[1]   32 1000
```

```
vapply(list(mtcars, quakes), nrow, 458392)
```

```
[1]   32 1000
```

Each of these has their own benefits:

- `vapply` is the fastest.
- `sapply` is the easiest to write and handle.
- `lapply` is the most flexible in terms of the results of the function.

`mapply`

A limitation of `apply` and `lapply`/`sapply`/`vapply` is that the funtions passed into them can only take in a single argument - a single row/column, or a single element of a list. `mapply` allows the function passed to take any number of entries.

```
x <- c(6, 2, 1, 4)
y <- c(9, 6, 0, -2)
z <- c(7, 4, 9, 1)
mapply(max, x, y, z)
```

```
[1] 9 6 9 4
```

Note that the `FUN` argument of `mapply` is the first argument, unlike previous functions.

`tapply`

`tapply` is used to execute a function to the value belonging to the similar group.

```r
head(iris)
```

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
1          5.1         3.5          1.4         0.2  setosa
2          4.9         3.0          1.4         0.2  setosa
3          4.7         3.2          1.3         0.2  setosa
4          4.6         3.1          1.5         0.2  setosa
5          5.0         3.6          1.4         0.2  setosa
6          5.4         3.9          1.7         0.4  setosa
```

```r
tapply(iris$Sepal.Length, iris$Species, mean)
```

```
    setosa versicolor  virginica 
     5.006      5.936      6.588 
```

Are the `apply` family of functions vectorized?

The R Inferno refers to the `apply` family of functions as "loop-hiding" - they contain implicit loops. `apply` actually has a loop in its R code. The others do drop down to C for their loops, however, at each step in the C loop, they evaluate the R function passed in. This is what makes them not vectorized, as a true vectorized function performs its loop in C *and* uses C compiled functions inside that loop.

Instead their main benefit is making code easier to write. They are shorter, more read-able, and less error-prone than loops. But they are **not vectorized**.

Writing vectorized functions

Consider the following small function: Given a number, it returns the negative version of it (e.g. the negative absolute value):

```r
#' Generate the negative absolute value of an input
#' @param x a numeric value
#' @return the negative absolute value
negabs <- function(x) {
  if (x > 0) {
    return(-x)
  } else {
    return(x)
```

```
  }
}
```

Putting aside the obvious inefficiency of such code, it also is not vectorized:

```
negabs(3)
```

```
[1] -3
```

```
negabs(-2)
```

```
[1] -2
```

```
negabs(c(5, 2, -1, 7))
```

```
Error in if (x > 0) {: the condition has length > 1
```

A naive way to "vectorize" would be

```
#' Generate the negative absolute value of an input
#'
#' This version is naively vectorized
#' @param x a numeric value
#' @return the negative absolute value
negabs2 <- function(x) {
  for (i in seq_along(x)) {
    if (x[i] > 0) {
      x[i] <- -x[i]
    }
  }
  return(x)
}

negabs2(3)
```

```
[1] -3
```

```r
negabs2(-2)
```

```
[1] -2
```

```r
negabs2(c(5, 2, -1, 7))
```

```
[1] -5 -2 -1 -7
```

While this is now "vectorized" in the sense that it takes in a vector and operates on each element, it is in no way efficient. Typically the best way to vectorize your custom functions is to in turn use existing vectorized functions.

```r
#' A fully vectorized function to generate the negative absolute
#' value of an input.
#'
#' @param x a numeric value
#' @return the negative absolute value
negabs3 <- function(x) {
  return(-abs(x))
}
negabs2(3)
```

```
[1] -3
```

```r
negabs2(-2)
```

```
[1] -2
```

```r
negabs2(c(5, 2, -1, 7))
```

```
[1] -5 -2 -1 -7
```

```r
x <- -5000:5000
microbenchmark(negabs2(x), negabs3(x))
```

```
Unit: microseconds
       expr     min        lq      mean   median        uq     max neval
 negabs2(x) 286.631 292.1045 301.9293 300.653 309.3040 340.874   100
 negabs3(x)   8.405  10.7625  18.0318  11.644  12.5665 582.692   100
```

If there are no existing vectorized functions to do what you need to do and a loop is the only thing feasible, you could consider writing your loop in C. The Rcpp package enables you to do this fairly easily, and there are other similar packages out there.

Monte Carlo Estimation

Vectorization can be particularly useful in Monte Carlo studies where we might otherwise be inclined to use explicit loops. We will look at some examples after an introduction to Monte Carlo estimates.

In statistics and data science we are often interested in computing expectations (means) of random outcomes of various types. When analytic expectations are unavailable or cumbersome to compute, it can be useful to obtain Monte Carlo approximations by simulating a random process and then directly averaging the values of interest.

This works because the sample average is generally a good estimate of the corresponding expectation:

$$\bar{\theta}_n := \sum_{i=1}^{n} \frac{X_i}{n} \to_p \theta := E[X].$$

$$\bar{\theta}_n \to_p \theta$$

In fact, assuming our data are independent and identically distributed (iid) from a distribution with finite variance, we can characterize the rate of convergence of a sample average to its population counterpart using the central limit theorem (CLT),

$$\sqrt{n}(\bar{\theta}_n - \theta) \to_d N(0, \sigma^2)$$

where $\sigma^2 = E[X^2] - E[X]^2$ is the variance of the underlying distribution from which X is drawn. This can be useful for constructing approximate confidence intervals for the Monte Carlo error.

Distribution functions

There are vectorized functions in **R** for simulating from many common distributions. Here are a few:

- `rnorm()` - Normal
- `runif()` - Uniform
- `rt()` - the t-distribution
- `rexp()` - Exponential
- `rpois()` - Poisson

A few list of built-in distributions are https://stat.ethz.ch/R-manual/R-devel/library/stats/html/Distributions.html. There are likely packages for any other distributions you may need.

Another useful function in R is `sample()` for sampling from a finite set of values, i.e. the discrete uniform distribution or any finite probability mass function.

As an aside, you should be aware that each of the distribution families above have corresponding `d*`, `p*`, and `q*` functions for computing densities, percentiles (CDF), or quantiles (inverse CDF) for each distribution.

Random Seeds

When we call one of the `r*` functions to generate random draws from a distribution, **R** relies on a pseudo-random number generate to generate from $U(0, 1)$ and produce the results. Thus the outcome of these calls depends on the current state of the generator. It is sometimes desirable to reproduce exactly the same pseudo-random sequence. You can do this by fixing the random seed using `set.seed()` which takes an integer argument.

```
a = runif(1)
b = runif(1)
a == b
```

```
[1] FALSE
```

```
set.seed(42)
a = runif(1)

set.seed(42)
b = runif(1)

a == b
```

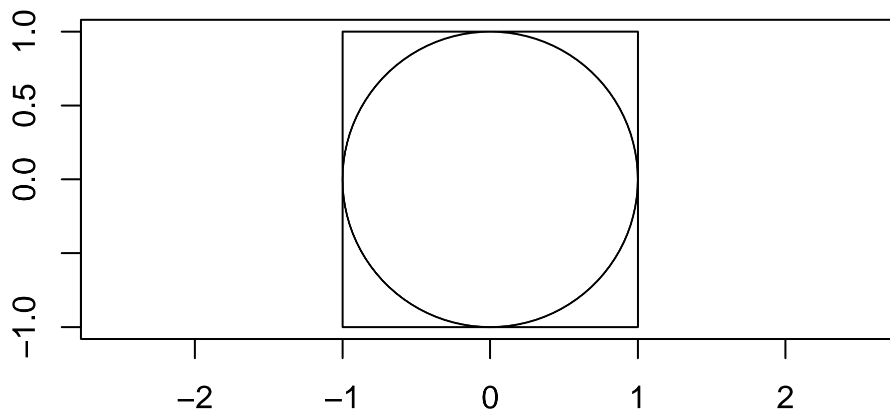
```
[1] TRUE
```

While `set.seed()` is useful in developing code, and when releasing final analyses, *do not* use it during your analysis - if your analysis relies on a stochastic process, removing the randomness removes the benefit of that tool.

Basic Example 1 - Estimating π

Let's estimate the value of π. Consider inscribing the largest possible circle inside a square:

```
plot(NULL, xlim = c(-1,1), ylim = c(-1,1), asp = 1, xlab = "",
     ylab = "")
rect(-1, -1, 1, 1)
plotrix::draw.circle(0, 0, 1)
```



The area of the square is $2 * 2 = 4$. The area of the circle is $\pi 1^2 = \pi$. This means that the ratio of the area of the circle to the area of the circle is $\frac{\pi}{4}$. We can therefore draw points from within the square, and use the ratio found within the circle to estimate π.

```r
#' Estimtes the value of pi by the proportion of points inside a square falling
#' inside a circle inscribed in that square.
#' @param n number of iterations
#' @return estimate of pi
estimate_pi <- function(n) {
  xcoord <- runif(n, -1, 1)
  ycoord <- runif(n, -1, 1)
  in_circle <- sqrt(xcoord^2 + ycoord^2) <= 1
  return(4 * sum(in_circle)/n) # estimate of pi
}
```

```r
estimate_pi(100) # estimate of pi
```

```
[1] 3.16
```

Let's re-run it with a larger `n` to see the convergence.

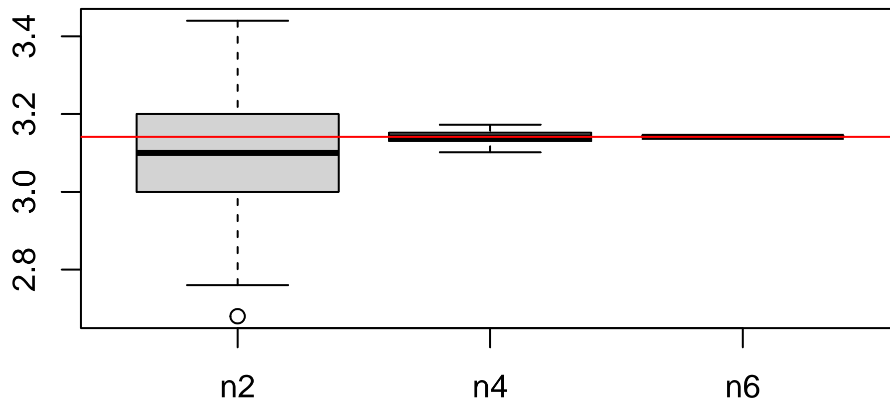
```r
estimate_pi(10000)
```

```
[1] 3.1312
```

We can re-run it several times to look at the distribution of estimates to see the uncertainty in the Monte Carlo methods:

```r
reps <- 100
n2 <- vector(length = reps)
for (i in seq_len(reps)) {
  n2[i] <- estimate_pi(100)
}
n4 <- vector(length = reps)
for (i in seq_len(reps)) {
  n4[i] <- estimate_pi(10000)
}
n6 <- vector(length = reps)
for (i in seq_len(reps)) {
  n6[i] <- estimate_pi(1000000)
}
```

```
boxplot(data.frame(n2, n4, n6))
abline(h = pi, col = "red")
```



Basic Example 2 - Estimating percentiles

Let's estimate percentiles for t-distributions with various degrees of freedom. Mathematically,

$$\theta_q := F(q) = \mathbb{P}(X \le q) = \int_{-\infty}^{q} f(x)dx = \int_{-\infty}^{\infty} \mathbb{I}_{[x \le q]} f(x)dx$$

where $F(\cdot)$ is the CDF and $f(\cdot)$ the PDF of a given t-distribution.

First, let's draw a random sample from the t_3 distribution

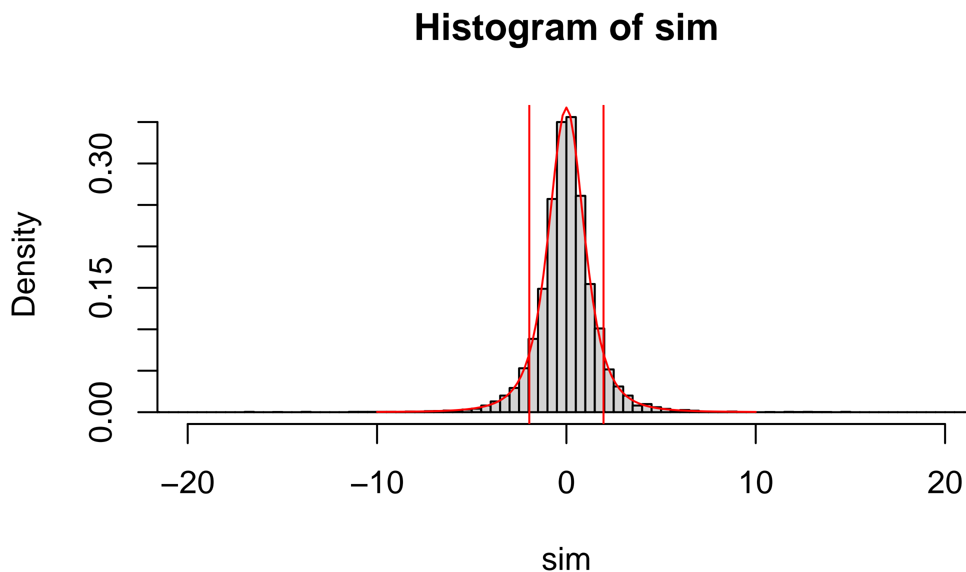
```
set.seed(100)
n <- 10000 # Number of samples
df <- 3 # Degrees of freedom
sim <- rt(n, df) # Draw distribution
```

We'll look specifically at the critical values for z, to see what happens if we use a z-approximation for t_3:

```
quantiles <- c(-1.96, 1.96)
```

Let's plot the results, including the empirical distribution and true curve, to make sure things look OK.

```
hist(sim, breaks = 200, probability = TRUE, xlim = c(-20, 20))
## distribution we drew from
curve(dt(x, df = df), -10, 10, add = TRUE, col = "red")
abline(v = quantiles, col = "red")
```



Finally, we can estimate $(\theta_{-1.96}, \theta_{1.96})$:

```
theta_hat = c(sum(sim <= quantiles[1]), sum(sim <= quantiles[2]))/n
```

In this case, our Monte Carlo estimate of $(\theta_{-1.96}, \theta_{1.96})$ is $(\hat{\theta}_{-1.96}, \hat{\theta}_{1.96}) = (0.0738,\ 0.9263)$. The actual values are $(\theta_{-1.96}, \theta_{1.96}) = (0.0724261,\ 0.9275739)$.

Advanced Example - Simulation Study for Nominal Confidence Intervals

Let's investigate the coverage probability of nominal 95% confidence intervals when the data does not come from a Normal (Gaussian) distribution.

We will assume the data come from an exponential distribution with mean one. The strategy here is to generate many (`mcrep`) data sets of size `n`.

For each data vector, we then calculate a nominal 95% confidence interval for the mean and check whether this interval contains the true value of one.

```
mcrep <- 10000                     # Simulation replications
n <- 30                            # Sample size we are studying
sim <- rexp(n * mcrep,             # Simulate standard exponential data
            rate = 1)              # mean = 1/rate
xmat <- matrix(sim,                # Reshape to matrix
               ncol = mcrep)
mn <- apply(xmat, 2, mean)         # Sample mean of each column (replicate)
std <- apply(xmat, 2, sd)          # Sample SD of each column (replicate)
se <- std / sqrt(n)                # Standard errors of the mean
m <- qt(1 - (1 - .95) / 2,
        df = n - 1)                # Multiplier for confidence level
lcb <- mn - m * se                 # lower confidence bounds
ucb <- mn + m * se                 # upper confidence bounds
```

Coverage probability is then the proportion of confidence intervals which cover the mean of 1.

```
cvrg <- sum(lcb < 1 & ucb > 1)/mcrep
cvrg
```

```
[1] 0.9271
```

What should be 95% coverage yields only 92.7%.

Broadcasting

We have previously discussed the need to be careful about **R** recycling values when performing arithmetic on arrays with differing dimensions. The formal name for this is *broadcasting* and it can be quite useful for vectorization.

Generally, when you perform a simple arithmetic operation (e.g. `*` or `+`) on arrays with the same dimensions the operation is applied point-wise so that `(A * B)[i, j] = A[i, j] * B[i, j]`. However, if the objects `A` and `B` have different dimensions, then the values can be *broadcast* like this:

```
c(1, 10) + c(2, 2, 6, 6)
```

```
[1]  3 12  7 16
```

Note how the value of 1 is added to the first and third entries of the larger vector, and the value of 10 is added to the second and fourth entry.

If the length of the arrays ("flattened" length, e.g. for a matrix, the total number of entries, or rows*columns) are not mulitples of each other, broadcasting still works, but the results can be unexpected, so a warning is given:

```
c(1, 10) + c(2, 2, 2)
```

```
Warning in c(1, 10) + c(2, 2, 2): longer object length is not a multiple of
shorter object length
```

```
[1]  3 12  3
```

The 1 is used for entries 1 and 3, the 10 only for entry 2.

We can take this to higher dimensions for some useful shortcuts. For example:

```
x <- matrix(c(1, 2, 3, 4, 5, 6), nrow = 2)
x
```

```
     [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6
```

```
x - c(1, 2)
```

```
     [,1] [,2] [,3]
[1,]    0    2    4
[2,]    0    2    4
```

```
x - c(1, 2, 3)
```

```
     [,1] [,2] [,3]
[1,]    0    0    3
[2,]    0    3    3
```

Matrices in R (and more generally, arrays) are **column-dominant**. That is, as the matrix gets "filled in", entries go down each column, as opposed to across each row. The broadcasting operates in the same fashion, down each column. In the second example, broadcasting the length 2 vector essentially subtracted 1 from the first row, and 2 from the second.

Broadcasting the length 3 vector, while not an error, did not necessarily produce anything useful.

This is useful for centering each row of a matrix or computing row-wise variances.

```
y <- matrix(c(4, 1, 6, 2, 9, 4), ncol = 3)
y
```

```
     [,1] [,2] [,3]
[1,]    4    6    9
[2,]    1    2    4
```

```
y - rowMeans(y)
```

```
          [,1]       [,2]     [,3]
[1,] -2.333333 -0.3333333 2.666667
[2,] -1.333333 -0.3333333 1.666667
```

```
rowSums((y - rowMeans(y))^2) / (dim(y)[2] - 1)
```

```
[1] 6.333333 2.333333
```

We can compare this to direct computation using an implicit loop via the `apply` function:

```
apply(y, 1, var)
```

```
[1] 6.333333 2.333333
```

Let's compare the timing of these two approaches:

```
reps <- 1000
n <- 30
xmat <- matrix(rnorm(reps * n), nrow = reps)
```

```r
microbenchmark(
  apply = apply(xmat, 1, var),
  vectorized = rowSums( (xmat - rowMeans(xmat))^2 ) / (dim(xmat)[2] - 1))
```

```
Unit: microseconds
       expr      min       lq       mean    median       uq      max neval
      apply 3742.152 3796.149 4058.73391 3851.3555 3916.812 7415.096   100
 vectorized   69.413   70.602   74.21246   71.5245   76.670   90.938   100
```

```r
# Ensure methods are equivalent:
all.equal(apply(xmat, 1, var),
          rowSums( (xmat - rowMeans(xmat))^2 ) / (dim(xmat)[2] - 1))
```

```
[1] TRUE
```

Example: Correlation Coefficients

In the next example, we consider the problem of computing the correlation coefficient between many vectors x and a single outcome y. This can be useful when screening a large set of potential predictors for a relationship with an outcome of interest y.

First, we generate data that has `n` observations, `m` predictors, and expected correlation `r` between each predictor and y.

```r
n <- 30
m <- 1000
r <- 0.4
y <- rnorm(n)
xmat <- matrix(rep(y, times = m), ncol = m)
# Reduce the variance in X by the amount added from correlation with y
xmat <- r * xmat + rnorm(n * m, sd = sqrt(1 - r^2))
```

Now, we can compute the correlations between each row of `xmat` and `y` and compare approaches.

First, a simple loop, as a baseline speed

```r
# First approach, calculate as a loop
mb1 <- microbenchmark(loop = {
```

```
  r1 <- vector(length = m)
  for (i in seq_len(m)) {
    r1[i] <- cor(xmat[, i], y)
  }
})
```

Next, using `apply`

```
mb2 <- microbenchmark(apply = {
  r2 <- apply(xmat, 2, function(v) cor(v, y))
})

all.equal(r1, r2)
```

```
[1] TRUE
```

Third, use linear algebra.

```
mb3 <- microbenchmark(linalg = {
  rmn <- colMeans(xmat)
  xmat_c <- xmat - matrix(rep(rmn, each = n), ncol = m)
  rsd <- apply(xmat, 2, sd)
  xmat_s <- xmat_c / matrix(rep(rsd, each = n), ncol = m)
  y_s <- (y - mean(y)) / sd(y)
  r3 <- y_s %*% xmat_s / (n - 1)
  r3 <- as.vector(r3)
})

all.equal(r1, r3)
```

```
[1] TRUE
```

Finally, repeat the linear algebra approach, while utilizing broadcasting

```
mb4 <- microbenchmark(linalg_b = {
  rmn <- colMeans(xmat)
  xmat_c <- t(t(xmat) - rmn)
  rvar <- colSums(xmat_c^2) / (dim(xmat)[1] - 1)
  rsd <- sqrt(rvar)
  xmat_s <- t(t(xmat_c) / rsd)
```

```r
  y_s <- (y - mean(y)) / sd(y)
  r4 <- y_s %*% xmat_s / (n - 1)
  r4 <- as.vector(r4)
})

all.equal(r1, r4)
```

```
[1] TRUE
```

Comparing the results:

```r
rbind(mb1, mb2, mb3, mb4)
```

```
Unit: microseconds
     expr      min        lq      mean    median        uq       max neval
     loop 7134.410 7287.7500 7823.2420 7385.4120 8074.8065 28051.708   100
    apply 6789.190 6907.5160 7262.6871 6981.1520 7773.8870  9000.074   100
   linalg 4306.189 4394.7695 4649.3619 4447.1675 4577.6705  5543.815   100
 linalg_b  332.018  374.9655  415.2837  390.4635  409.5285  2625.845   100
```

The fourth approach using linear algebra and broadcasting is by far the most efficient here. All approaches are much more efficient than computing $\binom{10{,}001}{2}$ correlations (by simply running `cor` against the entire matrix) when we only need 10,000.

While we should keep in mind that this was a single trial and not a formal comparison with replicates, a difference of this size is still meaningful. We should also be aware that one of the reasons the other approaches are slower is the time needed to allocate memory for (larger) intermediate objects.

Functional programming in Monte Carlo Studies

Recall our original example of computing coverage probabilities of nominal confidence intervals for non-Gaussian data.

If we wanted to carry out similar studies for many distributions, we may wish to write a function whose body carries out the simulation study using configurable parameters.

Here is the original example with a few changes.

```r
mcrep <- 10000                  # Simulation replications
n <- 30                         # Sample size we are studying
sim <- rexp(n * mcrep,          # Simulate standard exponential data
            rate = 1)           # mean = 1/rate
xmat <- matrix(sim,             # Reshape to matrix
               ncol = mcrep)
mn <- apply(xmat, 2, mean)      # Sample mean of each column (replicate)
std <- apply(xmat, 2, sd)       # Sample SD of each column (replicate)
se <- std / sqrt(n)             # Standard errors of the mean
m <- qt(1 - (1 - .95) / 2,
        df = n - 1)             # Multiplier for confidence level
lcb <- mn - m * se              # lower confidence bounds
ucb <- mn + m * se              # upper confidence bounds

cvrg <- sum(lcb < 1 & ucb > 1)/mcrep
```

Below we incorporate the simulation into a function with parameters for simulation settings. Here we use the special argument `...` to pass additional parameters to `rgen` by name. Note that `rgen` is expected to be a function - recall that in R, everything is an object.

```r
#' Estimate nominal CI coverage for data generated by `rgen`
#' @param rgen a function generating a vector of simulated data, i.e `rexp`,
#'   with length equal to its first argument.
#' @param target the actual expectation of `rgen`
#' @param mcrep the number of Monte Carlo replications and sample size.
#' @param n Sample size
#' @param conf_level the nominal coverage probability
#' @param ... additional parameters to pass to rgen
#' @return A length 1 numeric vector with the estimated coverage probability.
estimate_nominal_coverage <- function(rgen,
                                      target,
                                      mcrep = 10000,
                                      n = 30,
                                      conf_level = .95,
                                      ...) {
  sim <- rgen(n * mcrep, ...)  # Simulate data
  xmat <- matrix(sim,
                 ncol = mcrep)  # Each column is a replicate
  mn <- apply(xmat, 2, mean)    # Sample mean of each column
  std <- apply(xmat, 2, sd)     # Sample SD of each column
  se <- std / sqrt(n)           # Standard errors of the mean
```

```
  m <- qt( 1 - (1 - conf_level) / 2,
          df = n - 1) # Multiplier for confidence level
  lcb <- mn - m * se              # lower confidence bounds
  ucb <- mn + m * se              # upper confidence bounds

  # coverage probability
  return(sum((lcb < target) & (ucb > target))/mcrep)
}
```

Now we can use `estimate_nominal_coverage` for multiple simulations.

```
# Geometric(p) with mean (1 - p) / p
estimate_nominal_coverage(rgeom, target = 3, p = .25)
```

```
[1] 0.9278
```

```
# Poisson(lambda) with mean lambda
estimate_nominal_coverage(rpois, target = 4, lambda = 4)
```

```
[1] 0.9446
```

```
# t(df) with mean 0
estimate_nominal_coverage(rt, target=0, df=2)
```

```
[1] 0.9589
```

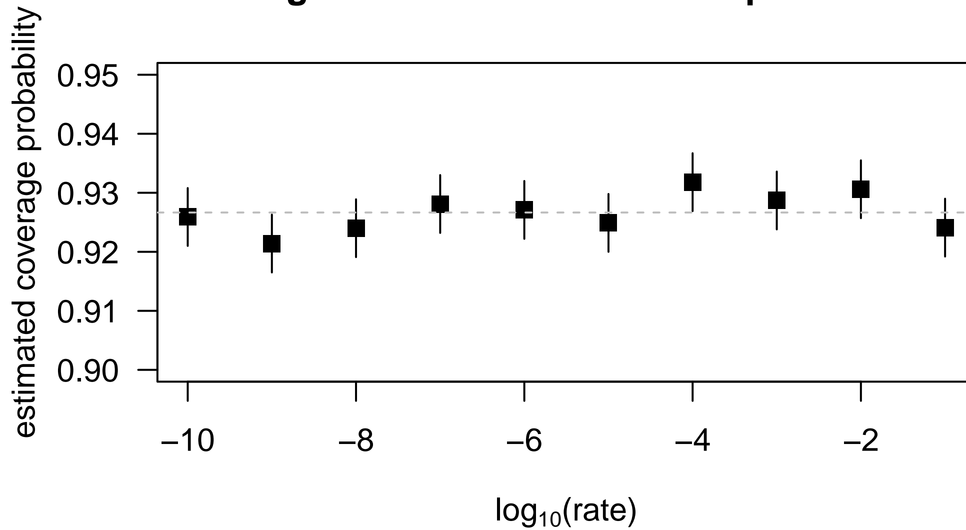
This could be useful, say, for exploring how the mean or another parameter impacts the coverage probability for a particular distribution.

Exponential Example

Here we use our function from above to explore how the nominal coverage for the exponential distribution changes over a range of mean parameters.

```
# rate parameters to explore
lambdas <- exp(-seq(1, 10, 1))

# store the results
```

```r
coverage_probs = vector(length = length(lambdas))
for (i in seq_along(lambdas)) {
  coverage_probs[i] <- estimate_nominal_coverage(rexp,
                                                 target = 1 / lambdas[i],
                                                 rate = lambdas[i])
}

plot(-seq(1, 10, 1), coverage_probs,
     ylab = "estimated coverage probability",
     xlab = expression("log"[10]*"(rate)"),
     las = 1, pch = 15, cex = 1.2, ylim = c(.9, .95),
     main = "Nominal Coverage Probabilities for the Exponential Distribution"
)

# Add Monte Carlo confidence bounds
for (i in seq_along(lambdas)) {
  x <- rep(-seq(1, 10, 1)[i], 2)
  y <- coverage_probs[i] + qnorm(.975) * c(-1, 1) * .25 / sqrt(10000)
  lines(x, y)
}

# add a reference line for the global mean
abline(h = mean(coverage_probs), lty = "dashed", col = "grey")
```



t distribution example

Here we examine how the coverage probability depends on the degrees of freedom if the data come from a t_{df} distribution.

```
df <- c(1, seq(2, 30, 2)) # df parameters to explore
coverage_probs <- vector(length = length(df)) # store the results
for (i in seq_along(df)) {
  coverage_probs[i] <- estimate_nominal_coverage(rt,
                                                 target = 0,
                                                 df = df[i])
}

plot(df, coverage_probs,
     ylab = "estimated coverage probability",
     xlab = "degrees of freedom for t",
     las = 1, pch = 15, cex = 1.2,
     ylim = c( 0.93, 1),
     main = "Nominal Coverage Probabilities for the t Distribution")

# Add Monte Carlo confidence bounds
```

```
for (i in seq_along(df)) {
  x <- rep(df[i], 2 )
  y <- coverage_probs[i] + qnorm(.975) * c(-1, 1) * .25 / sqrt(10000)
  lines(x, y)
}

# add a reference line for the nominal coverage value
abline(h = 0.95, lty = "dashed", col = "grey")
```