Common R Functions

2023-10-13

Vectorisation vs. `apply`

In this section, we compared the performance of for loops, vectorisation, and the `apply` family of functions. I wanted to take a look at how these different approaches affected performance when used for one of the exercises from the Statistical Methods computing lab. In this lab, we were asked to generate a random variable from a multivariate normal distribution using univariate random variables (rather than using an existing function for multivariate normal). Below, I have written different functions to do this using (1) 2 for loops, (2) 1 for loop, (3) vectorisation, and (4) `apply` to compare the performance.

All the functions below take the following parameters:

- `dimension`: the dimension of the random variable $\mathbf{x}$ to be generated, i.e. $\mathbf{x} \in \mathbb{R}^{\text{dimension}}$
- `mu`: the mean of $\mathbf{x}$, such that $\mu \in \mathbb{R}^{\text{dimension}}$ as well
- `Sigma`: the covariance matrix of $\mathbf{x}$
- `n_samples`: the number of samples to draw from the univariate normal distribution

I show the results of each function using the following objects:

```r
plot_mu <- matrix(c(2,1), ncol = 1)
plot_Sigma <- matrix(c(1,0.5,0.5,1), ncol = 2)
```

2 for loops

In this case, I have written 2 for loops. The inner one loops over `dimension` to calculate each element of my vector $\mathbf{y}$, where $\mathbf{y} = U^\top(\mathbf{x} - \mu)$ is the change of variables used to express the multivariate normal in terms of a univariate normal random variable. The second loops over the number of samples I am trying to calculate to generate $\mathbf{x}$ for each sample.

```r
mvn_generator_2_for_loops <- function(dimension, mu, Sigma, n_samples) {
  if(!matrixcalc::is.positive.definite(Sigma)) {
    stop("Sigma is not positive definite!")
  }

  # Find the eigendecomposition of Sigma inverse
  ev_Sigma_inv <- eigen(solve(Sigma))
  U <- ev_Sigma_inv$vectors
  eigenvalues <- ev_Sigma_inv$values
  D <- diag(eigenvalues)
  if(!all.equal(U %*% D %*% solve(U), solve(Sigma))) {
    stop("Something has gone wrong in your eigendecomposition!")
  }

  x_samples <- matrix(0, nrow = dimension, ncol = n_samples)
  for (i in 1:n_samples) {
    y_i <- matrix(0, nrow = dimension, ncol = 1)
    for (j in 1:dimension) {
      y_i[j] <- rnorm(1, mean = 0, sd = sqrt(1/eigenvalues[j]))
    }
    x_i <- U %*% y_i + mu
    x_samples[,i] <- x_i
  }

  return(x_samples)

}
```

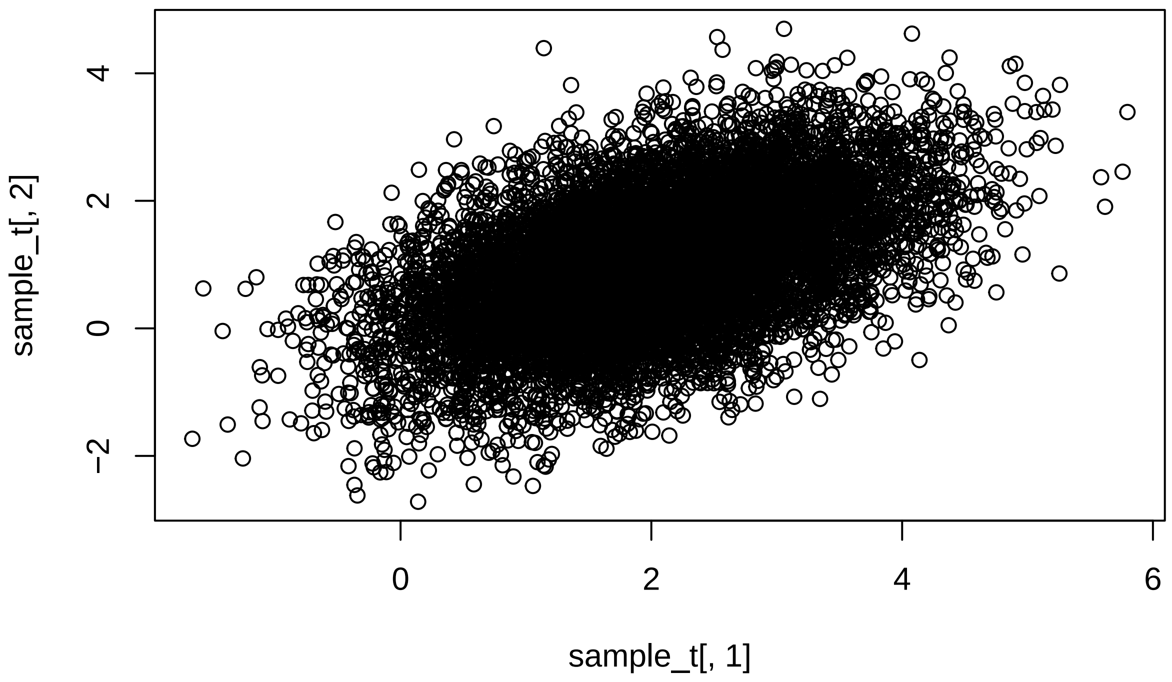
We can check the speed of the above function:

```
tictoc::tic()
sample <- mvn_generator_2_for_loops(dimension = 2,
                                    mu = plot_mu,
                                    Sigma = plot_Sigma,
                                    n_samples = 10000)
tictoc::toc()
```

```
## 0.111 sec elapsed
```

We can plot the result to verify that our function is appropriately sampling the way it should be:

```
sample_t <- t(sample)
plot(sample_t[,2] ~ sample_t[,1])
```



1 for loop

Alternatively, we can use a single for loop to decrease computation time:

```
mvn_generator_1_for_loop <- function(dimension, mu, Sigma, n_samples) {
  if(!is.positive.definite(Sigma)) {
    stop("Sigma is not positive definite!")
  }

  # Find the eigendecomposition of Sigma inverse
  ev_Sigma_inv <- eigen(solve(Sigma))
  U <- ev_Sigma_inv$vectors
  eigenvalues <- ev_Sigma_inv$values
  D <- diag(eigenvalues)
  if(!all.equal(U %*% D %*% solve(U), solve(Sigma))) {
    stop("Something has gone wrong in your eigendecomposition!")
  }

  x_samples <- matrix(0, nrow = dimension, ncol = n_samples)
  for (i in 1:n_samples) {
    y_i <- rnorm(n = dimension, mean = rep(0, dimension), sd = sqrt(1/eigenvalues))
    x_i <- U %*% y_i + mu
    x_samples[,i] <- x_i
  }
  return(x_samples)

}
```

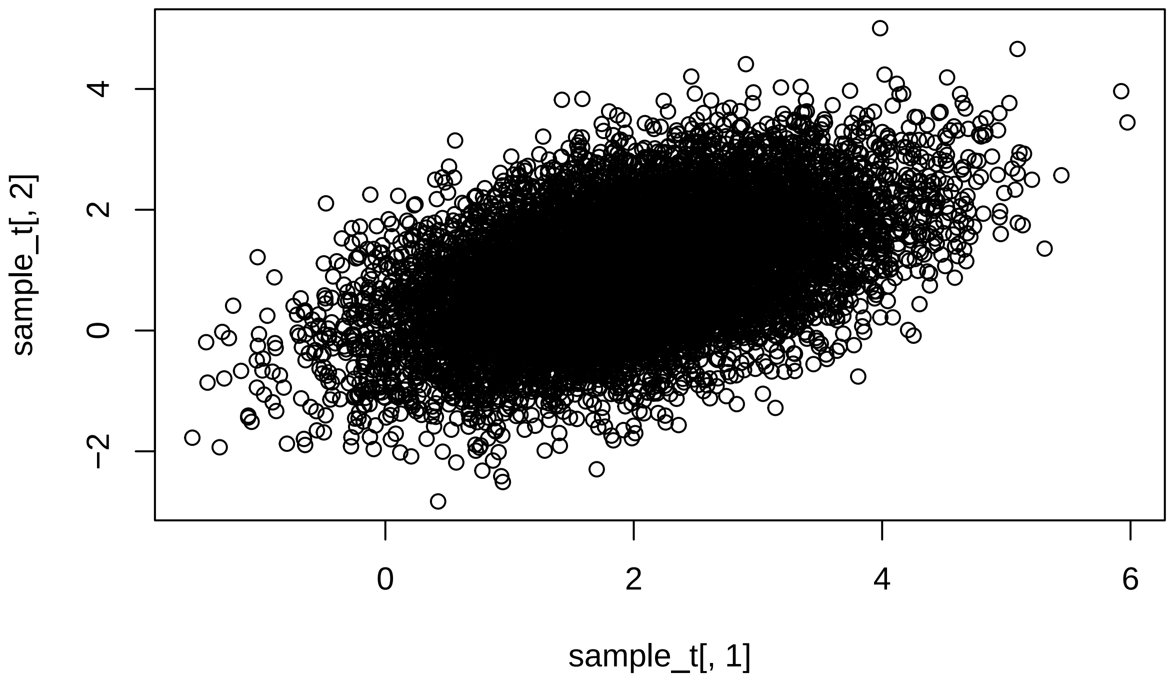
Now comparing the speed using only a single for loop:

```
tictoc::tic()
sample <- mvn_generator_1_for_loop(dimension = 2,
                                    mu = plot_mu,
                                   Sigma = plot_Sigma,
                                   n_samples = 10000)
tictoc::toc()
```

```
## 0.032 sec elapsed
```

Removing one of the for loops makes the function over three times faster (although this varied a bit depending on whether I was running the code in my console or in the knitted RMarkdown). Checking that our function is still sampling properly:

```
sample_t <- t(sample)
plot(sample_t[,2] ~ sample_t[,1])
```



Vectorisation

We can eliminate the for-loops altogether by making use of vectorisation in R.

```r
mvn_generator_vector <- function(dimension, mu, Sigma, n_samples) {
  if(!is.positive.definite(Sigma)) {
    stop("Sigma is not positive definite!")
  }

  # Find the eigendecomposition of Sigma inverse
  ev_Sigma_inv <- eigen(solve(Sigma))
  U <- ev_Sigma_inv$vectors
  eigenvalues <- ev_Sigma_inv$values
  D <- diag(eigenvalues)
  if(!all.equal(U %*% D %*% solve(U), solve(Sigma))) {
    stop("Something has gone wrong in your eigendecomposition!")
  }

  y_samples <- matrix(rnorm(n = dimension*n_samples,
                             mean = rep(0, dimension*n_samples),
                             sd = rep(sqrt(1/eigenvalues), n_samples)),
                       nrow = dimension,
                       ncol = n_samples)
  x_samples <- U %*% y_samples + matrix(rep(mu, n_samples),
                                         nrow = dimension,
                                         ncol = n_samples)

  return(x_samples)

}
```

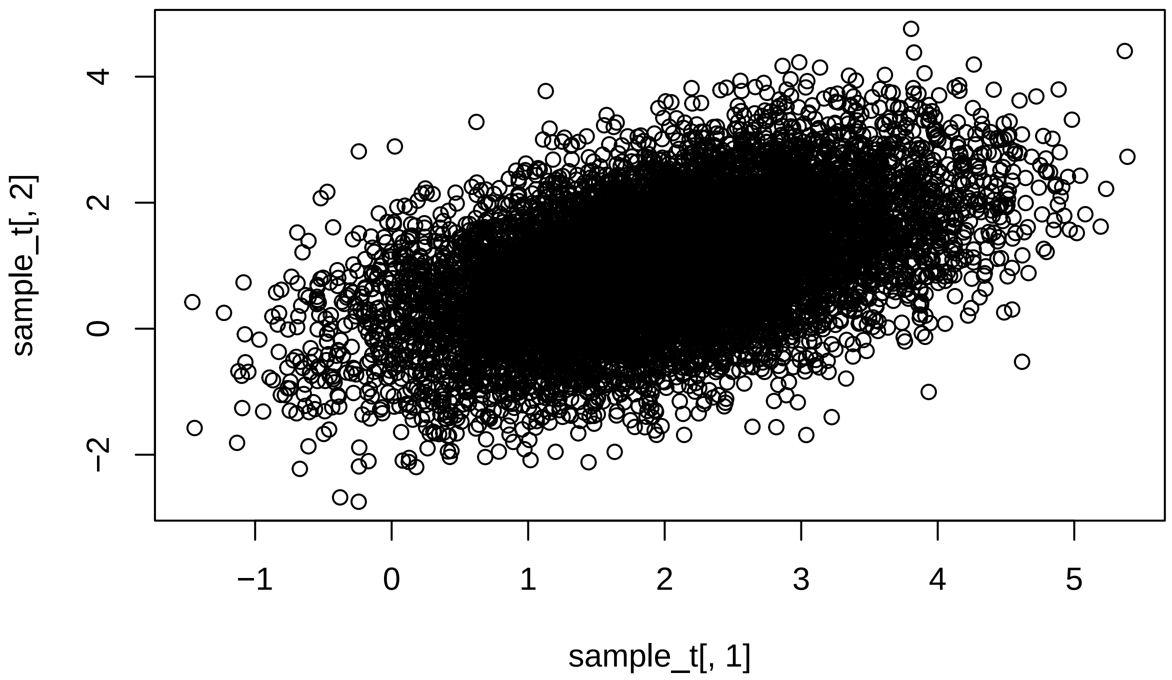
Speed with vectorisation:

```r
tictoc::tic()
sample <- mvn_generator_vector(dimension = 2,
                               mu = plot_mu,
                               Sigma = plot_Sigma,
                               n_samples = 10000)
tictoc::toc()
```

```
## 0.007 sec elapsed
```

The vectorised version of the function was approximately 4.5 times faster than the version with a single for loop, making this a huge improvement. We check the results again:

```r
sample_t <- t(sample)
plot(sample_t[,2] ~ sample_t[,1])
```



`apply` family

R also has a family of functions known as the `apply` family that allows us to apply the same function over and over across a given object. It contains the following functions:

- `apply()`: Applies a function over an array or a matrix (note, `apply()` must be applied to data with greater than one dimension)
- `lapply()`: Applies a function over a list or vector to return a list
- `sapply()`: Works like `lapply()` but returns a simplified output, rather than a list
- `mapply()`: Works like `sapply()` in multivariate situations

We can make use of the – function to rewrite our multivariate normal function once more:

```
mvn_generator_apply <- function(dimension, mu, Sigma, n_samples) {
  if(!is.positive.definite(Sigma)) {
    stop("Sigma is not positive definite!")
  }

  # Find the eigendecomposition of Sigma inverse
  ev_Sigma_inv <- eigen(solve(Sigma))
  U <- ev_Sigma_inv$vectors
  eigenvalues <- ev_Sigma_inv$values
  D <- diag(eigenvalues)
  if(!all.equal(U %*% D %*% solve(U), solve(Sigma))) {
    stop("Something has gone wrong in your eigendecomposition!")
  }

  # Create a matrix where the first column is the means
  # and the second is the standard deviations
  mean_sd <- matrix(c(rep(0, dimension), sqrt(1/eigenvalues)), nrow = dimension, ncol = 2)
  y_samples <- t(apply(mean_sd, 1, function(x) rnorm(n_samples, mean = x[1], sd = x[2])))
  x_samples <- U %*% y_samples + matrix(rep(mu, n_samples),
                                       nrow = dimension,
                                       ncol = n_samples)

  return(x_samples)

}
```

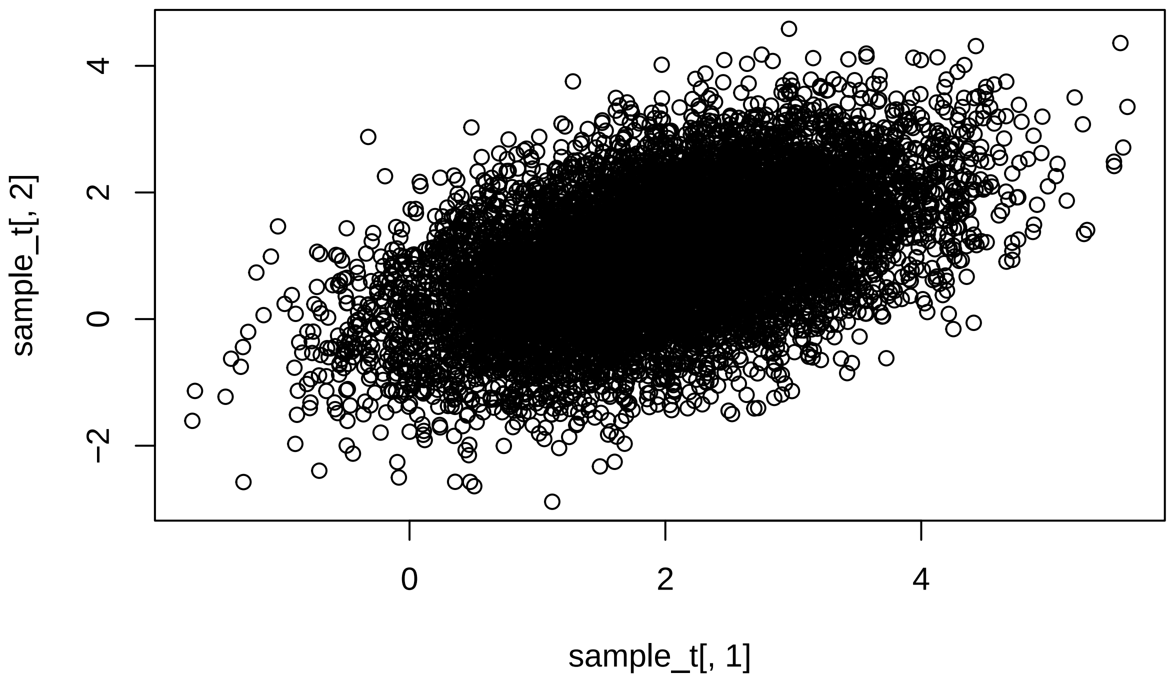
Speed with `apply()`:

```
tictoc::tic()
sample <- mvn_generator_apply(dimension = 2,
                              mu = plot_mu,
                              Sigma = plot_Sigma,
                              n_samples = 10000)
tictoc::toc()
```

```
## 0.009 sec elapsed
```

Speed with `apply()` was essentially identical to the speed with the vectorised function. These speed tests are a bit different than what was shown in the course notes, which showed the for loop as faster than `lapply`. Results check:

```
sample_t <- t(sample)
plot(sample_t[,2] ~ sample_t[,1])
```



Parallel programming

It is possible to parallelise R code to distribute code across different cores / processors. The `parallel` function contains `mclapply()` and `mcmapply()` functions, which are the parallelised versions of the `lapply` and `mapply` functions respectively. Note that these functions will not work on Windows. You can check the number of cores available with which to run these functions in the following way:

```
parallel::detectCores()
```

```
## [1] 20
```

Alternatively, the `foreach` and `doParallel` packages can also be used to evaluate R code in parallel using a set number of cores, which can be a more versatile choice that `mclapply` in complex situations.