# Computational Methods for Linear Model

Fred Wu

05 March 2022

This file [1] contains my own studies on the computations of linear models using **R/C++** (particularly `Rcpp`). I will implement QR decomposition for least square problems to estimate regression coefficients. I will use `sperm competition data I` from the `gamair` package for a simple illustration.

A linear model could be estimated in **R** as follows

```r
library(gamair)
data("sperm.comp1")
lmf <- lm(count ~ time.ipc + prop.partner, sperm.comp1)
summary(lmf)
```

```
##
## Call:
## lm(formula = count ~ time.ipc + prop.partner, data = sperm.comp1)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -239.740  -96.772    2.171   96.837  163.997
##
## Coefficients:
##              Estimate Std. Error t value Pr(>|t|)
## (Intercept)   357.4184    88.0822   4.058  0.00159 **
## time.ipc        1.9416     0.9067   2.141  0.05346 .
## prop.partner -339.5602   126.2535  -2.690  0.01969 *
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 136.6 on 12 degrees of freedom
## Multiple R-squared:  0.4573, Adjusted R-squared:  0.3669
## F-statistic: 5.056 on 2 and 12 DF,  p-value: 0.02554
```

For a linear regression, the coefficients could be estimated by the method of least square, which is to minimize

$$\hat{\beta} = \arg\min \|y - X\beta\|^2$$

[1] When compiling this document, instead of `knit` from `RStudio` interface, using `rmarkdown::render("lm.Rmd")` from the console could access Environment for pre-defined or pre-compiled objects or functions. This could be useful if you don't want `C++` codes to be compiled every time when you generate the document. In the `Rcpp` chunk below, the codes will not run but use the one that has already been compiled in the global environment.

## 0.1 Function lmQR

```r
X_mat <- model.matrix(~ time.ipc + prop.partner, sperm.comp1)
y <- sperm.comp1$count

lmQR <- function(X, y) {
  xQR <- qr(X)
  R <- qr.R(xQR)
  xTxInv <- chol2inv(R)
  lmCoef <- qr.coef(xQR, y)
  df <- nrow(X) - ncol(X)
  residVar <- crossprod(y - X %*% lmCoef) / df
  lmCoefStdErr <- sqrt(as.numeric(residVar) * diag(xTxInv))
  list(coefficients = as.numeric(lmCoef), stderr = lmCoefStdErr,
       df.residuals = df)
}
```

```r
lmQR(X_mat, y)
```

```
## $coefficients
## [1]  357.418434    1.941609 -339.560170
##
## $stderr
## [1]  88.0821592   0.9067154 126.2534581
##
## $df.residuals
## [1] 12
```

## 0.2 Function lmRcpp

`Armadillo` is a high performance `C++` library for linear algebra and scientific computing. It is extremely esay to use thanks to its design of syntax analogue to `Matlab`. The `R` package `RcppArmadillo` comes up with its own functions for linear model, which are `fastLm` and `fastLmPure`. The latter provides a reference use case of the `Armadillo` library, which will be used to conduct a speed test.

The following **C++** codes were written with `Rcpp` and `RcppArmadillo` by implementing the `QR` decomposition as well, which returned the same result as from `lmQR`. It is clear that the overall logic to implement the `QR` decomposition for linear regression is the same as in `lmQR` with very similar syntax.

```cpp
#include <RcppArmadillo.h>
using namespace arma;
using Rcpp::_;

//[[Rcpp::depends(RcppArmadillo)]]
// [[Rcpp::export]]
Rcpp::List lmRcpp(const mat &X, const vec &y) {

  mat Q;
  mat R;
  qr_econ(Q, R, X);

  mat xTxInv = square(inv(trimatu(R)));
```

```
  vec coef = solve(trimatu(R), Q.t() * y);
  vec res = y - X * coef;
  int df = X.n_rows - X.n_cols;
  double residVar = arma::dot(res, res) / (double) df;
  vec coefStdErr = sqrt(residVar * sum(xTxInv, 1));


  return Rcpp::List::create(_["coef"] = coef,
                            _["coefStdErr"] = coefStdErr,
                            _["df"] = df);

}
```

```
lapply(lmRcpp(X_mat, y), as.numeric)
```

```
## $coef
## [1]  357.418434    1.941609 -339.560170
##
## $coefStdErr
## [1]  88.0821592   0.9067154 126.2534581
##
## $df
## [1] 12
```

Here is a speed comparison between my functions, `fastLmPure` from the `RcppArmadillo` package with a simulated data. It is a surprise to see that `lmRcpp` works much faster than `fastLmPure`.

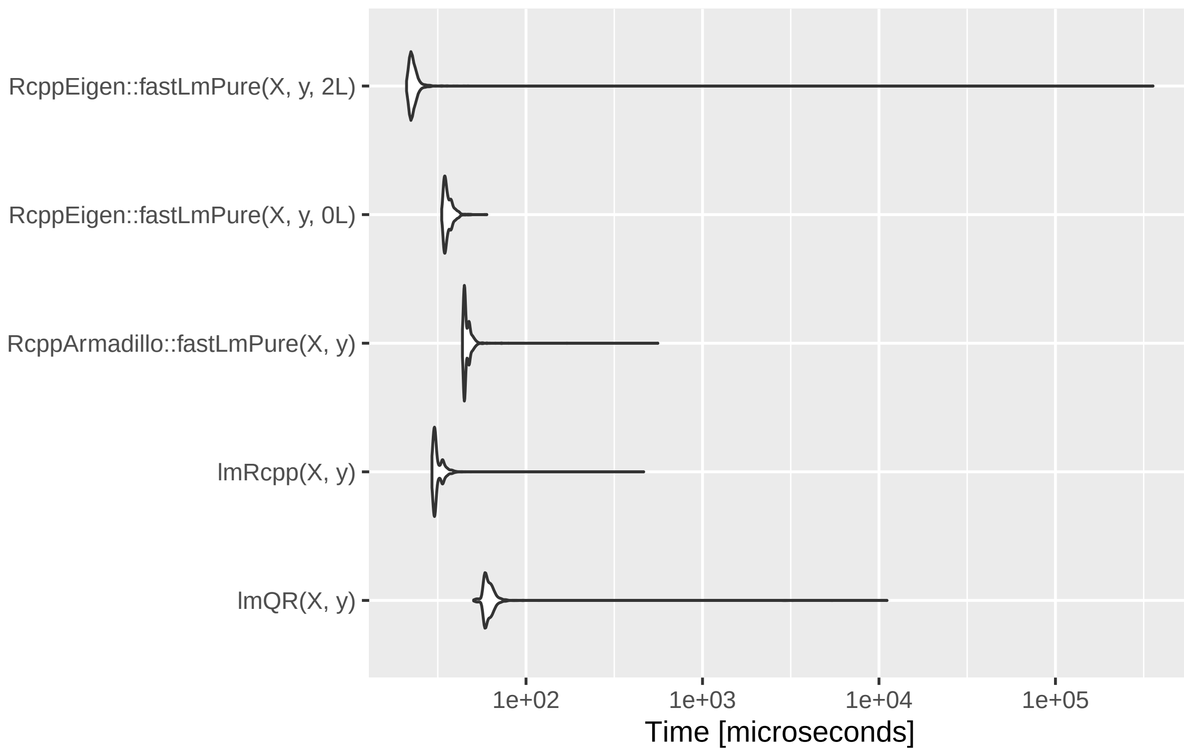
```
library(RcppArmadillo)
set.seed(1216)
n <- 500
p <- 10
X <- matrix(rnorm(n * p), n)
y <- rowSums(X) + rnorm(n)
(ans <- microbenchmark::microbenchmark(lmQR(X, y),
                                       lmRcpp(X, y),
                                       RcppArmadillo::fastLmPure(X, y),
                                       RcppEigen::fastLmPure(X, y, 0L),
                                       RcppEigen::fastLmPure(X, y, 2L),
                                       times = 1000))
```

```
## Warning in microbenchmark::microbenchmark(lmQR(X, y), lmRcpp(X, y),
## RcppArmadillo::fastLmPure(X, : less accurate nanosecond times to avoid potential
## integer overflows
```

```
## Unit: microseconds
##                             expr    min     lq      mean  median      uq
##                       lmQR(X, y) 50.471 58.466  89.86515 60.5160 63.9805
##                     lmRcpp(X, y) 29.315 30.135  32.03383 30.5860 33.0870
##  RcppArmadillo::fastLmPure(X, y) 43.624 44.649  47.00695 45.2230 47.5600
##  RcppEigen::fastLmPure(X, y, 0L) 33.292 34.563  36.22813 35.3420 37.3510
##  RcppEigen::fastLmPure(X, y, 2L) 21.033 22.058 380.45130 22.6115 23.4930
##         max neval
```

```
##   11115.510  1000
##     464.448  1000
##     558.666  1000
##      59.983  1000
##  357419.919  1000
```

```
ggplot2::autoplot(ans)
```




## 0.3 Summary

In `R`, to extract the diagonal of $(R^T R)^{-1}$, where $R$ is an upper triangular matrix from the `QR` decomposition, `chol2inv()` is much faster than `tcrossprod(solve(R))`, `apply(solve(R)^2, 1, sum)` or `backsolve(R, solve(t(R)))`, in which the last two ways are similar in speed test. However, in **C++** with `RcppArmadillo`, the equivalent way `sum(square(inv(R)), 1)` is much faster than the equivalent `backsolve::solve(R, inv(R.t()))`.