Homework 2

Costa Stavrianidis

2022-10-16

Problem 1

Explain why this returns FALSE,

```
identical(1.5e-09,1.6e-09)
```

```
## [1] FALSE
```

This returns FALSE because the function is testing for two objects being exactly equal, where these two values are not.

while this returns TRUE,

```
all.equal(1.5e-09,1.6e-09)
```

```
## [1] TRUE
```

This returns TRUE because it tests two objects for near equality. Since we are not modifying any of the default parameters for this function, it uses a default tolerance of close to 1.5e-8. Since the difference in our values is smaller than this tolerance, it considers them equal.

and this returns a mean difference

```
all.equal(1.5e-08,1.6e-08)
```

```
## [1] "Mean relative difference: 0.06666667"
```

This returns a mean difference because the difference between the two values is within the tolerance, so it does not consider them equal. This function then reports the mean relative difference between the two values if it does not consider them different.

Problem 2

Part 1

Consider solving the equation Ax=b in x where

```
set.seed(121)
m <- 4
matA <- matrix(rnorm(m*m), m, m)
vecx <- rnorm(m)
vecb <- as.numeric(matA%*%vecx)
```

The solution, by construction, is

```
vecx
```

```
## [1] -1.4690313  1.2157342 -1.1587392  0.5022799
```

Applying the Choleski decomposition

```
#include <Rcpp.h>
#include <RcppEigen.h>

using Eigen::MatrixXd;
using Eigen::VectorXd;

//[[Rcpp::depends(RcppEigen)]]
//[[Rcpp::export]]

VectorXd cholsol(MatrixXd matA, VectorXd vecb) {
  // Solve Ax = b in x using the Choleski decomposition
  // Note that you matA will need to be symmetric and pd
  const VectorXd vecx(matA.llt().solve(vecb));
  return vecx;
}
```

we get

```
xchol <- cholsol(matA, vecb)
xchol
```

```
## [1]  8.76287774  0.30745635 -0.04788003  0.76454195
```

```
all.equal(vecx, xchol)
```

```
## [1] "Mean relative difference: 2.879413"
```

What accounts for this discrepancy?

You are getting the incorrect vector using the Choleski decomposition because the Choleski decomposition requires the matrix A to be symmetric and positive definite, which it is not in this case.

```
isSymmetric(matA)
```

```
## [1] FALSE
```

If we were to apply the Choleski decomposition to a symmetric matrix A, we would get equal vectors.

```
matA <- matA %*% t(matA)
isSymmetric(matA)
```

```
## [1] TRUE
```

```
vecb <- as.numeric(matA%*%vecx)

xchol <- cholsol(matA, vecb)
xchol
```

```
## [1] -1.4690313  1.2157342 -1.1587392  0.5022799
```

```
all.equal(vecx, xchol)
```

```
## [1] TRUE
```

Part 2

Consider simulating symmetric m times m symmetric matrices of the form.

```
set.seed(7561)
m <- 100
```

```r
X <- matrix(rnorm(m*m), m, m)
A <- t(X)%*%X
```

- **Conduct a study comparing the speed and accuracy of matrix inversion using Choleski decomposition and Eigen's inverse (see tutorial)**
- **You may limit your study to matrices of the form shown above for m$\in$ {10,100,500,1000,2000}. Target using B=100 simulation replicates for each case (you may use lower (e.g., B=10) or higher (e.g., B=1000) number of replicates depending on availability of computing resources**
- **Quantify numerical accuracy using the L∞ norm. The simulation replication should be conducted with Eigen (for each case, the replication should be conducted using a loop within an Eigen function). You may iterate over the cases outside of Eigen (e.g., using foreach::foreach). See tutorial for a worked out example.**
- **Use the microbenchmark package to conduct the benchmarking (use microbenchmark::microbenchmark times argument to set number of replicates). For this part, let the package conduct the replication. See tutorial for a worked out example.**
- **Explain why the use of the Choleski decomposition may not be justified to the type of matrices simulated above**
- **Propose a numerically more efficient approach for simulating symmetric matrices**
- **Draft a report to summarize your observations. Use graphical summaries to summarize your results**

```cpp
#include <Rcpp.h>
#include <RcppEigen.h>

using Eigen::MatrixXd;
using Eigen::VectorXd;
using Eigen::Map;
using Eigen::Infinity;

//[[Rcpp::depends(RcppEigen)]]
//[[Rcpp::export]]

MatrixXd eigeninv(MatrixXd matA) {
  // Function for Eigen inverse
  const MatrixXd matinv(matA.inverse());
  return matinv;
}

//[[Rcpp::depends(RcppEigen)]]
//[[Rcpp::export]]

MatrixXd cholinv(MatrixXd matA) {
  // Function for Choleski inversion
  int nr = matA.rows();
  const MatrixXd matinv(matA.llt().solve(MatrixXd::Identity(nr, nr)));
  return matinv;
}

//[[Rcpp::depends(RcppEigen)]]
//[[Rcpp::export]]
```

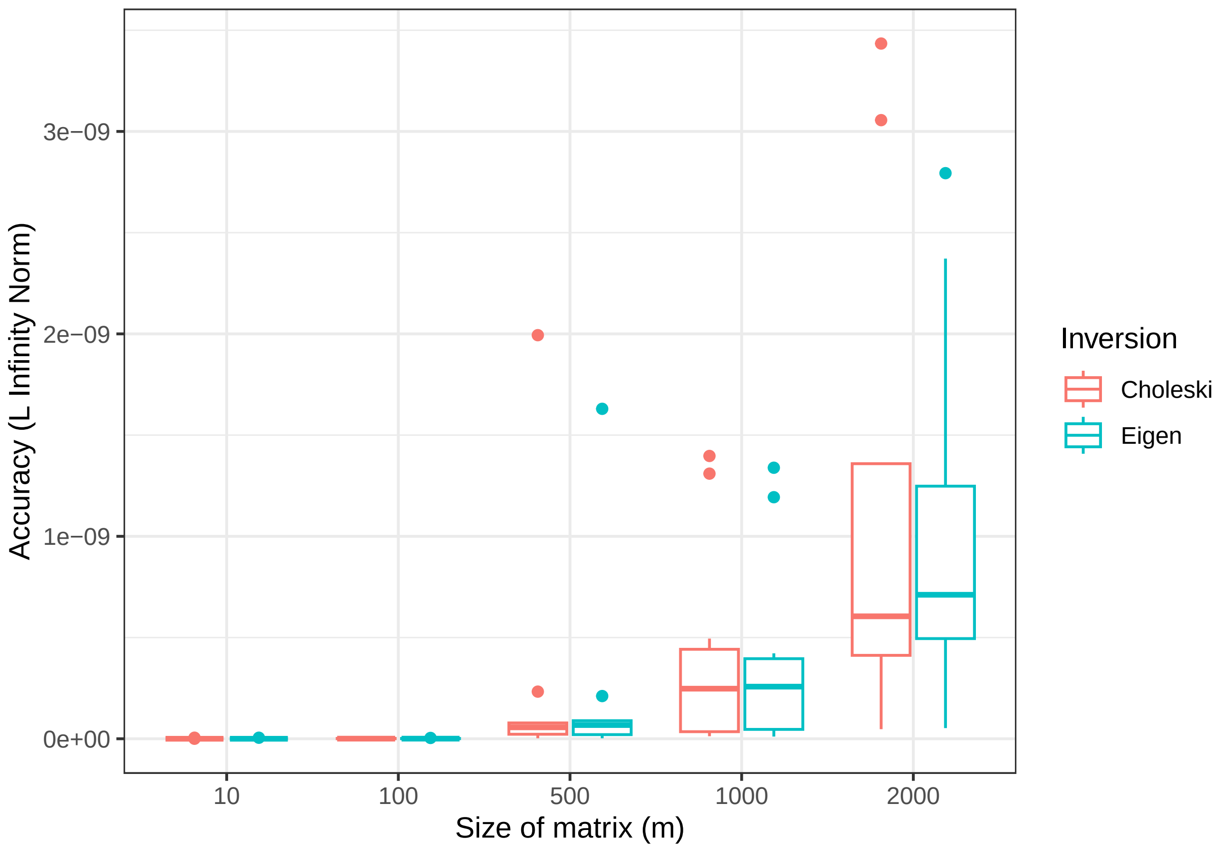
```
double getnorm(MatrixXd matA) {
  // L-infinity (max norm)
  const double maxnorm = matA.lpNorm<Infinity>();
  return maxnorm;
}

//[[Rcpp::depends(RcppEigen)]]
//[[Rcpp::export]]

MatrixXd simmat(int nrow, int ncol, double mu, double sigma) {
  // Simulate a random matrix of dimension nrow  x ncol whose
  // entries are drawn independently from a Normal(mu, sigma) law
  MatrixXd matA(nrow, ncol);
  VectorXd randvec(nrow*ncol);
  // Starting with Eigen 3.4 there is a more covenient approach to reshape
  randvec = Rcpp::as< Map<VectorXd> >(Rcpp::rnorm(nrow*ncol, mu,sigma));
  matA = Map<MatrixXd>(randvec.data(), nrow, ncol);
  return(matA);
}

//[[Rcpp::depends(RcppEigen)]]
//[[Rcpp::export]]

VectorXd accuracy(int size, int reps) {
  VectorXd finalvect(reps*2);
  for (int i=0; i<reps; i++) {
    MatrixXd matB = simmat(size, size, 0, 1);
    MatrixXd matA = matB.transpose() * matB;
    MatrixXd eigenmat = eigeninv(matA);
    MatrixXd cholmat = cholinv(matA);
    MatrixXd ideneigenmat = matA * eigenmat - MatrixXd::Identity(size, size);
    MatrixXd idencholmat = matA * cholmat - MatrixXd::Identity(size, size);
    finalvect[i] = getnorm(ideneigenmat);
    finalvect[i+reps] = getnorm(idencholmat);
  }
  return (finalvect);
}
```

```
set.seed(1213)

acc <- foreach::foreach(m = c(10L, 100L, 500L, 1000L, 2000L), .combine = rbind) %do% {

  # Accuracy simulation
  accuracy(m, 10)

}

set.seed(1213)

# Time unit for benchmarking
timeunit <- "microseconds"

foreach::foreach(m = c(10L, 100L, 500L, 1000L, 2000L), .combine = rbind) %do% {
```

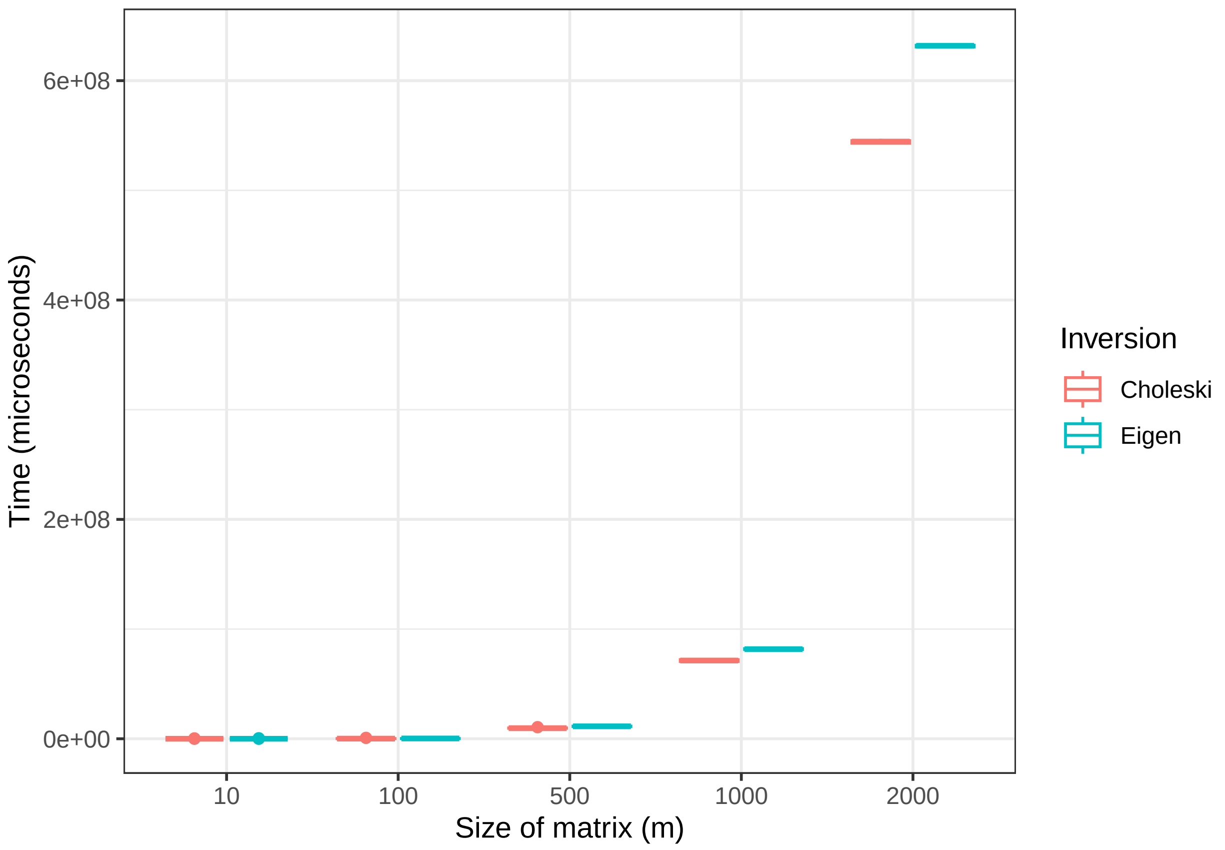
```r
  # Simulate symmetric random matrix
  X <- matrix(rnorm(m*m), m, m)
  X <- t(X) %*% X

  # Conduct benchmark analysis
  microbenchmark(
    eigInv = eigeninv(X),
    cholInv = cholinv(X),
    times = 10,
    unit = timeunit
  ) -> bmark
  # return results
  data.frame(m = m, as.data.frame(bmark))
} -> simres
```

```
## Warning in microbenchmark(eigInv = eigeninv(X), cholInv = cholinv(X), times =
## 10, : less accurate nanosecond times to avoid potential integer overflows
```

```r
# Graphical summary
acc1 <- as.data.frame(t(acc))
v1 <- c(rep("Eigen", 10), rep("Choleski", 10))
acc1$Inversion <- v1
acc1 <- acc1 %>% rename('10' = result.1, '100' = result.2,
                '500' = result.3, '1000' = result.4, '2000' = result.5)

acc2 <- acc1 %>% gather()
acc2 <- acc2[1:100,]
v2 <- c(rep("Eigen", 10), rep("Choleski", 10), rep("Eigen", 10), rep("Choleski", 10),
        rep("Eigen", 10), rep("Choleski", 10), rep("Eigen", 10), rep("Choleski", 10),
        rep("Eigen", 10), rep("Choleski", 10))
acc2$Inversion <- v2
acc2$value <- as.numeric(acc2$value)
acc2$key <- factor(acc2$key, levels=unique(acc2$key))
# Remove outliers
acc2 <- acc2 %>% filter(value < 1e-08)

acc2 %>%
  ggplot2::ggplot(aes(x = as.factor(key), y = value, col = Inversion)) +
  ggplot2::geom_boxplot() +
  ggplot2::theme_bw() +
  ggplot2::xlab("Size of matrix (m)") +
  ggplot2::ylab("Accuracy (L Infinity Norm)")
```



```r
simres1 <- simres %>% mutate(Inversion = ifelse(expr == "cholInv",
                                                "Choleski", "Eigen"))
simres1 %>%
  ggplot2::ggplot(aes(x = as.factor(m), y = time, col = Inversion)) +
  ggplot2::geom_boxplot() +
  ggplot2::theme_bw() +
  ggplot2::xlab("Size of matrix (m)") +
  ggplot2::ylab(paste0("Time (", timeunit,")"))
```



We can see from the plots that the numerical accuracy for both methods decreases as you increase the starting matrices size, however they both have similar accuracy overall for the different sizes. The inversions take longer for both methods as matrix size increases, however the Choleski inversion is noticeably quicker than the Eigen method as the size of the matrices increase.

The Choleski decomposition is only appropriate for these matrices if they are symmetric and positive definite. We make the matrix symmetric and positive semidefinite when we multiply it by its transpose, however we do not guarantee that it is positive definite. It is only positive definite if it is a full rank matrix.

A numerically more efficient approach to simulating symmetric matrices may be to generate a symmetric random matrix by adding the transpose of the matrix to itself instead of having to multiply the transpose of the matrix (as long as the matrix is square, as it is in this case).

Problem 3

Consider the bootstrapping example from the tutorial and explain why the simulation replicates in the example below are not reproducible. Provide an approach to make the boostrap replicates reproducible.

```
#include <Rcpp.h>
#include <RcppEigen.h>

using Eigen::VectorXd;

//[[Rcpp::depends(RcppEigen)]]
//[[Rcpp::export]]

VectorXd bootvec(VectorXd vecx) {
    // This function samples the entries of vecx with replacement

    // Create an output vector equal to the lenth of vecx
    int nr = vecx.rows();
    VectorXd vecxboot(nr);

    // Simulate nr replicates uniformly over {0,..., nr-1} with replacement
    Rcpp::NumericVector bootindex(nr);
    bootindex=Rcpp::floor(Rcpp::runif(nr)*nr);

    // Index the output using the bootstrap indices
    // Eigen 3.4 introduces new slicing features to provide a more
    // elegant solution
    for(int i = 0; i < bootindex.size(); ++i){
        vecxboot[i] = vecx(bootindex[i]);
    }

    return(vecxboot);
}
```

```
set.seed(2121)
# Simulate random noise vector from Normal(0,1)
vecx <- rnorm(100, 0, 1)
# Generate two bootstrap replicates
boot1 <- bootvec(vecx)
boot2 <- bootvec(vecx)
# Why are the bootstrap replicates not reproducible?
set.seed(2121)
identical(boot1, bootvec(vecx))
```

```
## [1] FALSE
```

```
identical(boot2, bootvec(vecx))
```

```
## [1] FALSE
```

The replicates are not reproducible because the bootvec function has randomness to it. When we generate boot1 and boot2 initially, we did not set a seed, so they will not be identical to when bootvec is called on vecx. We can remedy this by setting the same seed right before creating boot1 and boot2, as shown below, and get reproducible bootstrap replicates.

```
set.seed(2121)
# Simulate random noise vector from Normal(0,1)
vecx <- rnorm(100, 0, 1)
# Generate two bootstrap replicates
set.seed(2121)
boot1 <- bootvec(vecx)
boot2 <- bootvec(vecx)
# Why are the bootstrap replicates not reproducible?
set.seed(2121)
identical(boot1, bootvec(vecx))
```

```
## [1] TRUE
```

```
identical(boot2, bootvec(vecx))
```

```
## [1] TRUE
```