PS7 - STAT 243

Alexander Fred Ojala
Student ID: 26958060
Collaborators: Guillame Baquiast, Milos Atz and Alexander Brandt

November 16th 2015

1 Problem 1

Answer submitted to Github Nov 9:

What are the goals of their simulation study and what are the metrics that they consider in assessing their method?
The goal of the study is to present their new method for Hypothesis testing on the order of the normal mixture. They present an expectation-maximization (EM) test for testing the null hypothesis and they consider an arbitrary order m0 under a finite normal mixture model. The simulation study assesses the accuracy of the proposed asymptotic approximation in finite samples and to examine the power of the EM test. They also look at the significance in the metric of p values in assessing their method. Their method applies a penalty function on the component variance to obtain a bounded penalized likelihood. It then assesses the improvement over the null models.

What choices did the authors have to make in designing their simulation study? What are the key aspects of the data generating mechanism that likely a ect the statistical power of the test?
The choices they made for the simulation study was sample size (200-400), the number of repetitions (5000), signficance levels (5 percent and 1 percent). Moreover the EM test that they carry out is calculated based on the recommendations for the terms B, K, as well as the penalty functions. And the power of the EM, under each alternative model is calculated, based on 1000 repetitions. The key aspects of the data generating mechanism - random numbers from the normal mixture - that will affect the statistical power of the test is the sample size (larger will increase the power). Also the significance level has an affect: the more stringent (lower) the significance level, the lower the power. Their penalty function contains a tuning parameter that affects the precision of the test. They solve the tuning problem via a novel computer experiment and provide an easy-to-use data-dependent formula. Another aspect of the data generating mechanism that will affect the power is how they choose parameters such as proportions, std and component mean.

Suggest some alternatives to how the authors designed their study. Are there data-generating scenarios that they did not consider that would be useful to consider?
Alternatives would be to increase the sample size significantly in order to affect the statistical power. We could also increase the number of replicates to further increase the simulation error. In the data generating scenarios they exclude the case where two normal mixture components have the same mean, because it is more challenging technically (it would be interesting to see). Also they could further change the component mean, std and other proportions, that would be useful and interesting to see. As well as larger models (with more parameters).

Give some thoughts on how to set up a simulation study for their problem that uses principles of basic experimental design (see the Unit 10 notes) or if you think it would be di cult, say why.
When setting up an experimental design analysis we could implement the standard strategy and discretize the inputs. Since the component mean varies this might be difficult however. If the number of inputs was small we could carry out a full factorial design. However the levels needs to be chosen in a reasonable way

to not have too many treatment combinations. If we have a large a large number of inputs we could use the Latin hypercube approach (each input sampled uniformly).

Do their gures/tables do a good job of presenting the simulation results and do you have any alternative suggestions for how to do this? Do the authors address the issue of simula- tion uncertainty/simulation standard errors and/or do they convince the reader they've done enough simulation replications?
Their figures and tables present boxplots indicating the significance level of the tests as well as the parameter selections and the powers of the EM for different models and sample sizes. They could present more metrics, such as the true p-values and some summary statistics. However what I really lack is they also should asses the magnitude of impact from different inputs by showing a decomposition of sums of squares, instead of only statistical significance. They do not clearly state that the simulation uncertainty / standard errors is satisfactory beyond doubt, even though they do state the simulation repetitions and the sample sizes.

Interpret their tables on power (Tables 4 and 6) - do the results make sense in terms of how the power varies as a function of the data generating mechanism?
The tables 4 and 6 make sense in how the power varies, since the significance becomes greater for larger sample sizes, and also that it is more accurate for models with less parameters (when m is smaller). Moreover they state that 'when the component means under the alternative models become far away from one another, the power of the test increases.' Which also can be seen in the tables. However I would like to know how likely is it for EM to be of power 100 and why it varies quite drastically (maybe could have read the whole paper to figure this out). Moreover in generating the data for the simulation study they could have been more clear and presented the structure of real data and talk more about their distributional assumptions, dependence structure, outliers and random effects.

Discuss the extent to which they follow JASA's guidelines on simulation studies (see the end of the Unit 10 class notes for the JASA guidelines).
The JASA guidelines state that:
'Results Based on Computation - Papers reporting results based on computation should provide enough information so that readers can evaluate the quality of the results. Such information includes estimated accuracy of results, as well as descriptions of pseudorandom-number generators, numerical algorithms, computers, programming languages, and major software components that were used.'

Overall they fulfill the criteria, however what I really lack is that they do not explicitly describe how they generate the pseudorandom numbers to obtain the simulated Type I errors. The data generating mechanism is only briefly described as sampling from the normal mixture distribution.

2 Problem 2

2.1 2 a)

The number of operations (multiplications and divisions) for the Cholesky decomposition for order n^3 and n^2 is:

$$\frac{1}{6}n^3 + \frac{1}{2}n^2 + \mathcal{O}(n)$$

The derivation of this result can be found in the Appendix (the last pages) in Figure 4, which is a picture of the written calculations that led up to the result.
Source: `https://mediatum.ub.tum.de/doc/625604/625604.pdf` (pg. 9)

The result has the same coefficient in fron of n^3 as presented in the unit 9 notes (that it should be $\frac{1}{6}n^3 + \mathcal{O}(n^2))$).

2.2 2 b)

If the Cholesky decomposition is implemented as presented with pseudo-code in figure 1, then we can see that if we instead of assigning a new matrix U we can indeed update our A matrix in every step. That will yield that A is successively transformed into the upper-triangular matrix that we want to acheive from the Cholesky decompotion (sequentially in every step, beginning with the rows and working out the calculations for each column).

This is true since we first update the first row (we will not need this information from the original matrix later) and then in the for loop over all rows i, we begin with updating the diagonal element (that will be the first element of the upper triangular matrix for row i when $2 \leq i \leq n$). Then we have the for loop over the columns, which updates the value sequentially only using the values from the original matrix (that we have not changed yet) and new values that are stored for the updated indicies in the A matrix. Therefore we can directly update A and obtain the correct value for all indicies that we are updating without losing any information.

Hence we can store the Cholesky upper triangular matrix in the same storage space as used for the original matrix, as we go along with the calculations.

1. $U_{11} = \sqrt{A_{11}}$
2. For $j = 2, \ldots, n$, $U_{1j} = A_{1j}/U_{11}$
3. For $i = 2, \ldots, n$,
   - $U_{ii} = \sqrt{A_{ii} - \sum_{k=1}^{i-1} U_{ki}^2}$
   - for $j = i+1, \ldots, n$: $U_{ij} = (A_{ij} - \sum_{k=1}^{i-1} U_{ki}U_{kj})/U_{ii}$

Figure 1: Pseudo code for how to carry out the Cholesky decomposition

2.3 2 c)

For problem 2 c) the memory used for the Cholesky decomposition was computed and compared to that of only storing the original matrix X in memory. This was done for six square matrices of the size 1000, 2000, 3000, 4000, 5000 and 6000.

As can be seen from the result in Figure 2 the same amount of memory is used for the Cholesky decomposition of the matrix X as for the original matrix (the memory use exactly doubles when we store the Cholesky decomposition). It was also confirmed that the function $chol(X)$, did not require any additional memory use (or less memory use) by checking that $mem\_change(chol(X))$ only was around 1kb (which is the same as $mem\_change(NULL)$). This confirms the result in 2b) that R updates the original matrix and stores that new one in memory, in order to create the new Cholesky decomposition of the matrix. It also indicates that R stores the zeros in the upper triangular Cholesky matrix as floating numbers that take up 8 bytes of memory.

See more comments on the results (time and memory use) after the code chunk below.

The code implemented as seen below, and by running that in a clean R session I obtained the memory change as seen in Figure 2 and the time it took to carry out the Cholesky decomposition in Figure 3.

```
gc(TRUE) #Turn on garbage collector to further eval memory use
iter=6
ns<-rep(0,iter) #vector for n's
memX<-rep(0,iter) #vector for memory of storing X
memChol<-rep(0,iter) #vector for memory use of storing chol
```

```r
time<-rep(0,iter) #vector for the time to compute chol(X)


for (i in 1:iter) {
  n=1000*i
  ns[i]=n
  first<-mem_used() #The memory use before we store X or chol(X)

  X<-crossprod(matrix(rnorm(n^2), n)) # pos. def matrix
  second<-mem_used() #Increase of memory use for storing X

  #Calculate memory use for only storing X
  memX[i]=round((second-first)/10^6,2) #Present the results as MB, with two decimal digits

  t<-system.time(X2<-chol(X)) #time to compute chol(X)
  time[i]=t[3] #only look at elapsed time (system time is much larger than user time)

  #Calculate memory use for storing X2 = chol(X)
  third<-mem_used() #Memory use of storing chol(X)
  memChol[i]=round((third-first)/10^6,2) #Memory
  rm(X) #remove X to free up memory for next iteration
  rm(X2) #remove X2
  print(i) #to see progress
}

#Plotting
par(mfrow=c(1,2))
plot(ns,memX,type="l",main="MB mem used X",xlab="n, size of X=[n x n]",ylab="MB")
text(ns,memX,memX)
plot(ns,memChol,type="l",main="MB mem used chol(X)",xlab="n, size of X=[n x n]",ylab="MB")
text(ns,memChol,memChol)
par(mfrow=c(1,1))
plot(ns,time,main="Elapsed time to compute chol(X)",xlab="n, size of X=[n x n]",
     ylab="time [s]",type="l")
text(ns,time,round(time,3))


gc(FALSE)
```

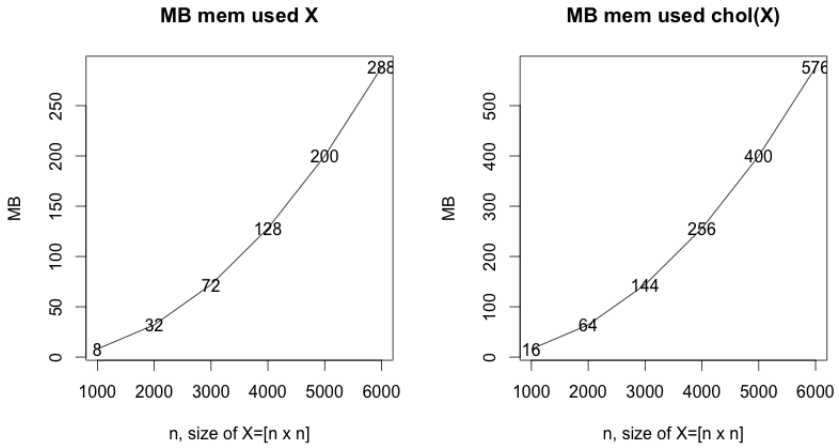

Figure 2: Extra memory use when storing the Cholesky decompoisition of a square matrix of size n times n. As can be seen the memory use doubles every time, hence the Cholesky decomposition only updates the original matrix in order to obtain and store the Cholesky.

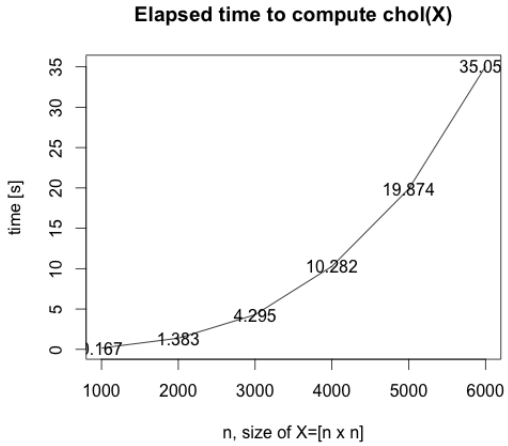

Figure 3: The time it took in R to carry out the Cholesky decomposition. The result is that the time to carry out the Cholesky decomposition is increasing by (polynomial) power of three, which is accurate according to the theory of the order of calculations as n is increasing.

Results for 2c)

Memory use did not change (as expected) except for storing the new Cholesky matrix (the same size as the original matrix, as they are of the same dimensions), which indicates that the original matrix is updated sequentially and then stored in memory as a new object when the Cholesky decomposition is carried out.
Memory use scale with n as expected ($n \cdot n \cdot 8$ equals the number of bytes used in memory for storing the matrix, where n is the size of the square matrix).
The time it takes to carry out the Cholesky decomposition is empircally increasing 'cubically' (by the order of n^3, which is true according to theory, since the number of calculations that needs to be carried out are of order n^3). This is verified in Figure 3. This was also confirmed by plotting n against $(time)^{\frac{1}{3}}$ which produced a straight line, hence the plot approximately represents a polynomial function where the greatest exponent is to the power of three.

3 Question 3

3.1 3 a)

The full inversion metho is of order $n^3 + \mathcal{O}(n^2))$ calculations, R's $solve(X, y)$ method which makes use of the LU decomposition is of order $\frac{1}{3}n^3 + \mathcal{O}(n^2))$ calculations and the Cholesky decomposition method is of order $\frac{1}{6}n^3 + \mathcal{O}(n^2))$ calculations (in order to calculate b). For the decompositions we need to do a back/forward solve after X has been decomposed in order to solve for b. For the full inversion we need to calculate the matrix multiplication of Xy, which is a slower method than the forward/backsolve. Therefore what is expected from theory is that the full inversion should take the longest, the R solver less than one third of that time and the quickest one should be the Cholesky decomposition that should take less than one sixth of the time of the full inversion.

Running the code below shows that indeed the slowest operation was the full inversion (250seconds). However what is interesting to note is that the LU decomposition (28 seconds) is more than 10 times faster than the full inversion. This probably has to do with how R has optimized the calculations in their built in solve function (because if we used LU straight away it would not be so close in timing to the Cholesky solution also). The Cholesky solution only took (20 seconds) about 8 percent of the time of the full inversion.

Overall this clearly shows that the decomposition methods, and avoiding carrying out matrix multiplications is much quicker in R than doing full inversion and matrix multiplying. Even quicker, than what we expect from theory.

```
#Setup the matrix
n=5000
X<-crossprod(matrix(rnorm(n^2), n)) #pos def matrix

y<-rnorm(n)

#Full inversion method, n^3
system.time(b1<- solve(X) %*%y)

##    user  system elapsed
## 244.125   1.228 250.973

b1<-b1[,1] #to get rid of matrix structure

#R's implemented solve function, based on LU, n^3/3
system.time(b2<-solve(X,y)) #based on LU decomposition

##    user  system elapsed
##  27.787   0.104  27.984

#Cholesky solution, n^3/6
system.time({
  U<-chol(X)
  b3<-backsolve(U, backsolve(U, y, transpose = TRUE))
})

##    user  system elapsed
##  19.828   0.079  19.972
```

3.2 3 b)

As can be seen from the results below, the resulting b vectors are not the same for the different methods up to machine precision. When comparing the full inversion and the R solver, they are approximately numerically the same up to decimal digit 10^{-12}. When comparing the Cholesky to the full inversion and the R solver it is approximately numerically the same up to 10^{-7}. That is how many decimal places that agree for the result b, when comparing the result obtained from the different methods (note that we did not use a method with full precision in order to compare the results with the true value of b (e.g. to compute it in Maple), since we only know X and y).

Machine precision is 10^{-16}, why it differs is because of the condition number of the matrix, $cond(X)$ causes the output to to have less accuracy, namely if $cond(X) \approx 10^t$ then we will have accuracy for our results that are around $cond(X) \cdot 10^{-16}$ (the decimal places that are accurate). So the smaller the condition number is, the better the accuracy of our result is.

The reason that the calculations differ more for the Cholesky decomposition than when comparing the R solver to the full inversion, is that the Cholesky decomposition uses a different method for evaluating the result, while the full inversion and the R solve, both make use of the function solve. For the Cholesky decomposition, we also obtain obatin a new, lower, condition for the Cholesky decomposed U matrix that might affect the results. The R solver and the full inversion are directly extracted from the X matrix (at least up to my knowledge for how the R solve function works).

The results from the calculations on precision are shown below. The condition number was also calculated, and the result indicates that we should expect approximate precision of result up to the decimal digit around 10^{-9} (or close to 10^{-8}). We can see that our results when comparing the Cholesky to the full inversion / R solver approximately yields this result. Here the condition number is calculated from the matrix 2-norm (function $norm(X, type = 2)$) of the original matrix X.

```
### Question 3 b) ----

max(abs(b1-b2)) #Compare result from full inversion and R solver

## [1] 7.105427e-13

max(abs(b1-b3)) #full inversion vs Cholesky

## [1] 6.342167e-08

max(abs(b2-b3)) #R solver vs Cholesky

## [1] 6.342164e-08

K<-norm(X,type="2")*norm(solve(X),type="2") #Condition number
10^-16*K #the approximate numerical precision we can expect from our results

## [1] 4.029049e-09
```

4 Question 4

The Generalized Linear Regression is given by:

$$\hat{\beta} = (X^T \Sigma^{-1} X)^{-1} X^T \Sigma^{-1} Y$$

Pseudo-code for efficient GLS estimator

In order to efficiently obtain $\hat{\beta}$, and not just compute the inverses, we can implement a solution following the pseudo-code below. Here the specific order of operations is presented in the two pseudo-code blocks below, along with the significant order of operations:

1. Do an eigendecomposition (the matrix is pos. semi-def): $\Sigma^{-1} = C\Lambda C^T$, **Order:** $\mathcal{O}(n^3)$
2. Define p: $p = C\Lambda^{1/2}$ (Define p matrix, [n x n]), **Order:** $\mathcal{O}(n^3)$
3. $\Sigma^{-1} = pp^T$
4. Then $\hat{\beta} = (X^T pp^T X)^{-1} X^T pp^T Y$
5. Set $p^T X = X_*$ and $y_* = p^T Y$, **Order:** $\mathcal{O}(n^3)$ and $\mathcal{O}(n^2)$
6. At last we obtain $\hat{\beta} = (X_*^T X_*)^{-1} X_*^T Y_*$

Then, in order to efficiently compute $\hat{\beta}$ we can now use a QR decomposition, which efficiently calculates the Generalized Linear Regression estimate, as done in the pseudo-code below:

1. Use $X_*^T X_* \hat{\beta} = X_*^T Y_*$
2. Do QR decomposition $X_* = QR$, **Order:** $np^2 + \frac{1}{3}p^3$
3. We get $R^T Q^T QRhat\beta = R^T Q^T Y_*$
4. Q is orthogonal and R is invertible: $R\hat{\beta} = Q^T Y_*$, **Order:** $\mathcal{O}(n^2)$

In order to solve for $\hat{\beta}$ in the last line of the pseduo-code below, we can use a back-solve in R.

The algorithm was implemented (and confirmed to work on several example matricies) in an R function called *gls*, that can be seen below. In order to speed up the computations even more, the QR decomposition could also be carried out on the tall-skinny matrix X with parallellization in order to greatly speed up the process and make the method even more efficient:

```
library(matrixcalc) #used for defensive programming

gls = function(X,y,sig) {

  if(!is.positive.definite(sig, tol=1e-8))
    stop("Sigma needs to be pos. definite")
  if(!isTRUE(dim(X)[1] == dim(sig)[1]) &
     !isTRUE(length(y) == dim(X)[1]))
    stop("Check matrix dimensions")

  #Obtain sigma inverse
  sigInv<-solve(sig)

  e<-eigen(sigInv) #eigen-decomposition of sigma inverse
  c<-e$vectors #extract eigenvectors
  lambda<-diag(e$values) #form diagonal matrix with eigen values on diagonal

  p<-c%*%sqrt(lambda) #define p

  X1 <- t(p)%*%X
  Y1 <- t(p)%*%y

  #QR decomposition
```

```
  X.qr<-qr(X1)
  Q<-qr.Q(X.qr)
  R<-qr.R(X.qr)

  beta <- backsolve(R,t(Q)%*%Y1) #solve for beta
  #Ok, same as:
  #beta = solve(t(X)%*%solve(sig)%*%X) %*% t(X) %*% solve(sig) %*% y

  return(beta)
}

#CHECK

n=5
p=2
X<-matrix(runif(n*p),n,p)
y<-runif(n)
sig<-crossprod(matrix(rnorm(n^2), n))

gls(X,y,sig)

##           [,1]
## [1,] 0.1766335
## [2,] 0.8074778
```

5 Question 5

5.1 5a) and 5b)

The mathematical proofs for these two questions can be found in Figure 5 and Figure 6 in the Appendix (the last pages).

The number of operations in Question 5 b) is exactly n summations (we only add the constant c to every eigenvalue of the matrix X in order to find the eigenvalues of matrix Z, which makes the operation exactly Order n). N.B. I did not include the eventual multiplication of the scalar c and the Identity matrix (c times 1 n times, in order to obtain the eigenvalues I only added the scalar c to every eigenvalue of X, that is given by the eigenvalue decomposition).

6 Appendix

6.1 Question 1 a)

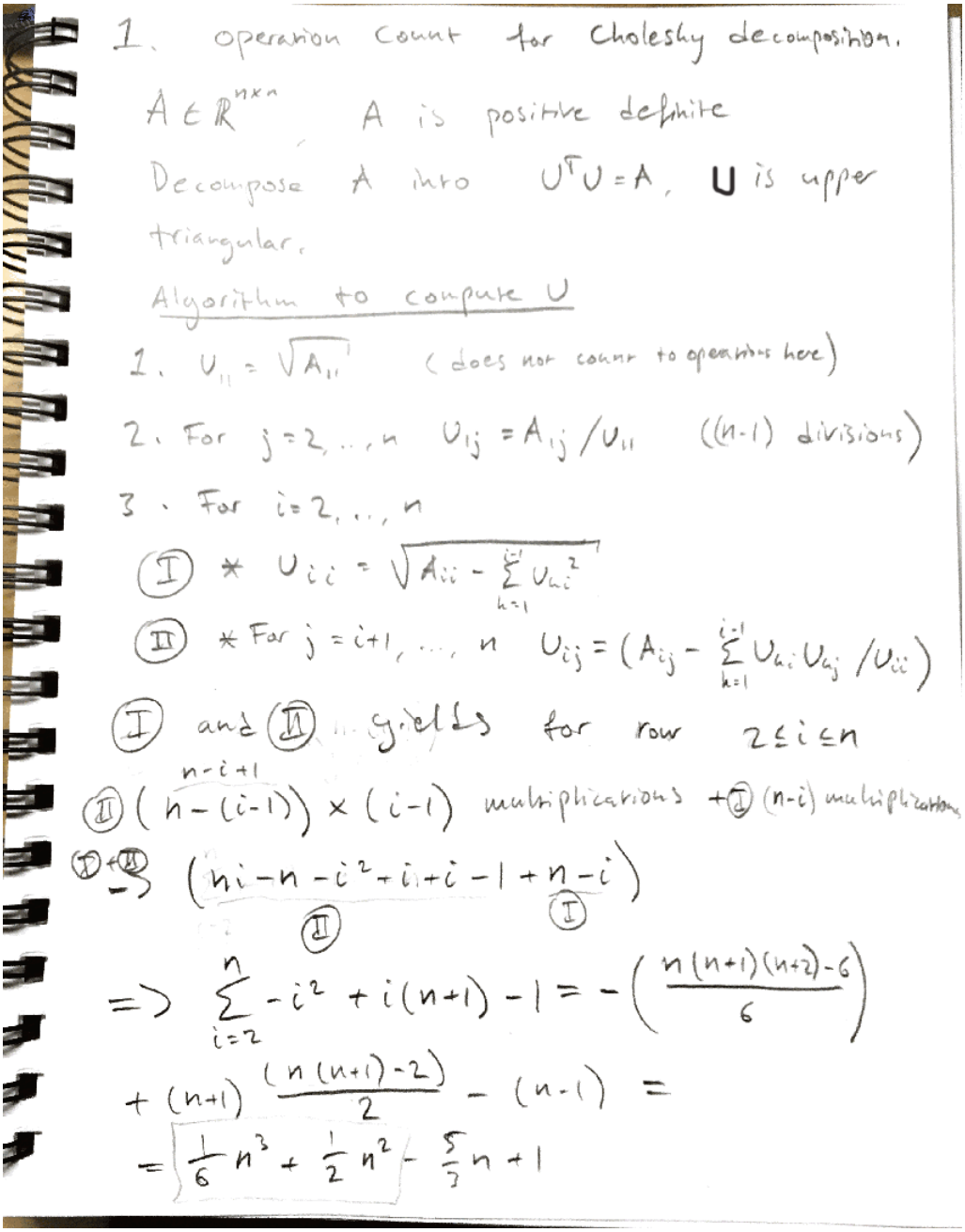

Figure 4: Operation count (multiplications and divisions) for the Cholesky decomposition

6.2 Question 5a and 5b)

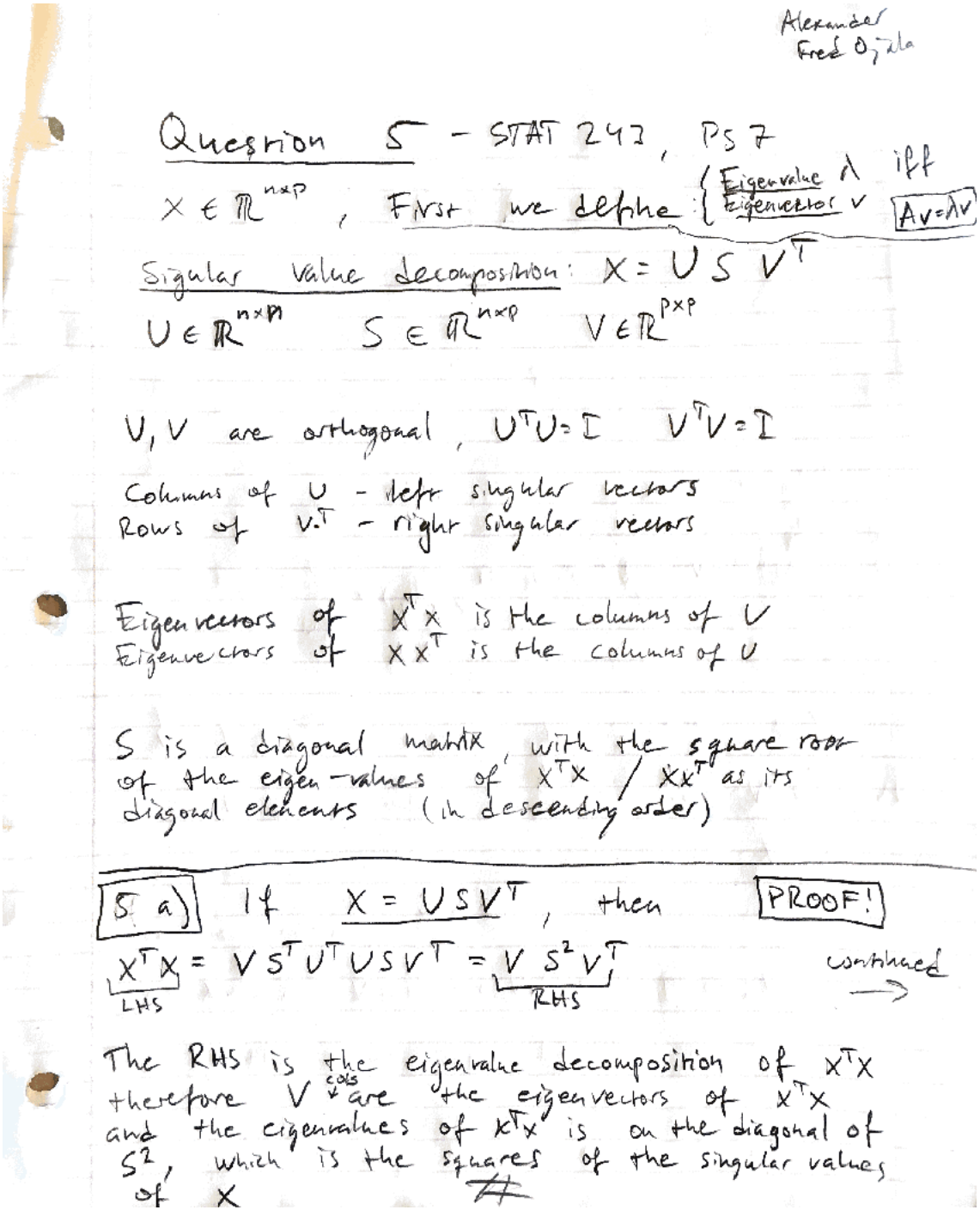

Figure 5: First page of answers for Question 5, here only the beginning of 5a)

5 a) continued X^TX pos. semidefinite

A matrix M is positive semidefinite if

$\forall z \neq \bar{0}$, $z \in \mathbb{R}^{n\times 1}$, then $z^TMz \geq 0$

We get: $z^TX^TXz = \underbrace{(Xz)^T \cdot (Xz)}_{} \geq 0$ ##

(Inner product of something by itself is always non-negative)

---

5 b) $X \in \mathbb{R}^{n\times n}$ $w^TXw \geq 0$ $\forall w$, $w \neq \bar{0}$, $w \in \mathbb{R}^{n\times 1}$

Eigendecomposition: (because pos. semi-definite)

$X = VDV^T$ where $D = \text{diag}(\lambda_1, \ldots, \lambda_n)$ eigenvalues

and V is an orthogonal matrix, i.e. $V^{-1} = V^T$, $VV^T = I$

To compute the eigenvalues of $Z = X + cI$, $c \in \mathbb{R}$

we can rewrite it as:

$Z = X + cI = VDV^T + c \cdot VV^T = \underbrace{V(D+cI)V^T}_{\text{eigendecomp of } Z}$

Thus, the number of calculations needed is

exactly n summations, where we add c to every eigenvalue of X in order compute Z's eigenvalues ##

Figure 6: Second page of answers to Question 5 a) and 5 b)