Homework 1

Claire Kelling, Statistics 540

February 8, 2018

Problem 1 **Consider an nxn covariance matrix with its (i,j)th element defined as follows: $\Sigma_{ij} = exp(-|i-j|/(n\phi))$, with $\phi = 0.2$**

(a) Write computer code to simulate M = 100 draws from a multivariate normal distribution with mean 0 (vector of 0s) and covariance matrix as above for n = 1000. Write your own code to also evaluate the multivariate normal pdf at each simulated value. (Recall that you may use some ideas discussed in lecture.) Write pseudocode for both algorithms. You do not have to provide details for standard operations like eigendecompositions, you can simply write, for instance: "Perform a spectral decomposition of M."

For this problem, I tried three different algorithms for simulating from a multivariate normal. In this case, I found the most efficient algorithm to be using the Choleski decomposition. This is especially useful because I can reuse the Choleski decomposition in evaluating the multivariate normal pdf. Since this is the main computational task in evaluating the pdf, this is extremely advantageous.

Below, find my pseudocode for simulating a multivariate normal distribution using this method:

1. Define a matrix, X, that is M rows and n columns, all univariate standard normal variables.
2. Define the matrix Sigma as above.
3. Construct the Choleski decomposition of Σ, where $CC^T = \Sigma$
4. Simulate 100 vectors of 1x1000 multivariate normals by constructing a Mxn matrix of μ, which is in case a vector of 0's, and add it to the matrix X*L. This will return a matrix whose rows are M=100 simulations of an n-dimensional multivariate normal distribution.

Below, find my pseudocode for evaluating a multivariate normal pdf at each simulated value:

1. Use the Choleski decomposition of Σ, where $CC^T = \Sigma$, **from above**
2. Calculate $r = C^{-1}(x-\mu)$, where X are your simulated values (so that $r^T r = (x-\mu)\Sigma^{-1}(x-\mu)$)
3. Calculate the log multivariate normal pdf using
   $log(f(x_1, ..., x_n)) = -0.5nlog(2\pi) - log(det(\Sigma)) - 0.5r^T r$
   $= log(f(x_1, ..., x_n)) = -0.5nlog(2\pi) - log(\prod_{i=1}^{n} C_{ii}) - 0.5r^T r$
   $= log(f(x_1, ..., x_n)) = -0.5nlog(2\pi) - \sum_{i=1}^{n} log(C_{ii}) - 0.5r^T r.$
   This is because we know from class that $|\Sigma|^{1/2} = \prod_{i=1}^{n} C_{ii}$.
4. Repeat for all 100 draws from the multivariate normal found above.

(b) Plot the pdf values (you can do this on the log scale) for the samples. Report the wall time for the simulation algorithm and the evaluation of the pdf (jointly) using, say, system.time.

Before I decided on my final algorithm, I simulated a couple of runs from each of the following techniques: using Choleski decomposition, singular value decomposition, and spectral decomposition. I also used two ways to calculate the multivariate normal pdf: directly (using solve(Σ)) and through the $r^T r$ technique described in part a. After performing this exercise, I see that, at least at size 1,000, the combination of Choleski technique for simulating multivariate normal and the $r^T r$ technique for evaluation of the pdf seems to be the fastest. Actually, using the diagonal of the Choleski decomposition, C, is vastly preferred to the determinant because if I use det(Σ), it returns the value 0 in this case.

Below, in Figure 1, we see the histogram of the evaluations of the pdf on the log scale. We see they mostly lie between 600 and 1,000.

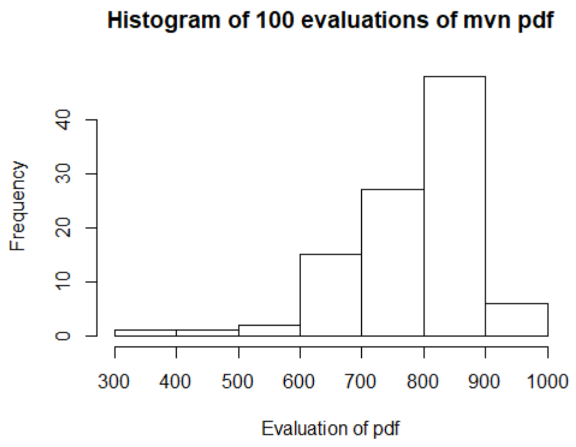


Figure 1: pdf Values from M=100 simulations with n=1,000, log scale

After simulating this several times, I find that the wall time for evaluating the simulation algorithm **and** the evaluation of the pdf, jointly, tends to be approximately 40 seconds when n is 1,000. This is more than 15 seconds shorter than if we were to calculate the Choleski decomposition twice. It is also considerably shorter than the SVD or spectral decomposition methods at n=1,000, 2,000 and 3,000. Below, in Table 1, we can see the comparison between the methods for relatively small values of n. This table includes just the time for the simulation of the random vector, not the evaluation of the pdf. However, all of the techniques scale at about the same rate, so I decided to keep using the Choleski method.

Method	n = 1,000	n = 2,000	n=3,000
Choleski	0.3	2.5	7.5
Spectral Decomposition	3.22	25.5	81.5
SVD	4.7	37.6	129.7

Table 1: Time (seconds) to simulate 100 draws from n-dimensional multivariate normal distribution

The data in this tale represent the main reason for proceeding with the Choleski method.

(c) Now that you have successfully implemented this for n = 1000, repeat the exercise for n = 2,000; n = 3,000 and so on, going up to the largest value you are able to handle. Make plots of how the computational cost scales with increasing n.

Below, in Figure 2, I have included a plot that shows how the computational cost (wall time in this case) scales as the dimension of the multivariate normal distribution (and therefore the dimension of Σ increases). The computational cost scale quite significantly with n, seemingly on the order of n^3. The highest value I was able to handle was n =10,000, but this took quite a long time, as you can see on the graph where the y-axis is seconds (over 42,000 seconds or about 12 hours to simulate).

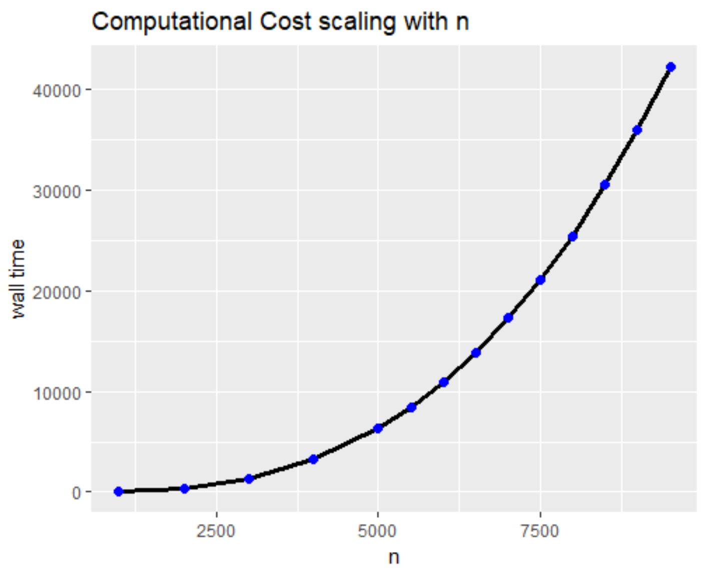


Figure 2: n vs Wall Time (computational cost)

(d) Write out the computational complexity of your algorithm. Show the details of your calculation. If you have used an R function (e.g. eigen), you need to find out the cost of that algorithm by looking through the documentation.

For this problem, there are two parts to my computational complexity. The first part comes with the simulation of a multivariate normal distribution. The second comes with the evaluation of a multivariate normal pdf. I will break this into two parts, and write out the computational complexity.

First, I will discuss the **simulation of multivariate normal** distribution. I identified several potential algorithms that I could use here. As stated above, I chose Choleski decomposition method, as opposed to SVD or spectral decomposition. This is due to the scale at n = 1,000, 2,000 and 3,000. Below, in Table 2 you can see the break down of the computational complexity in terms of flops. I have omitted the addition of mu due to the fact that mu is 0 (vector of 0's) in our case.

I tried to look into the computational complexity of the rnorm function and was not able to find much. The source code is not readily available online and the source code that I was able to find was not intuitive in terms of determining the number of flops. Therefore, to determine the number of flops for this function, I test out a couple different cases and then compare the theoretical cost with rnorm to the CPU time calculated in part c. I find that the flop count for rnorm is probably not of the order n^3 or n^2 but more likely just $n * M$.

However, I think in total, it is most important to recognize that the total theoretical cost for this simulation and evaluation of the pdf is dominated by the Choleski decomposition, or the order of n^3 flops.

Operation	Flop Count
$C = chol(\Sigma)$	$n^3/3$ flops
simulating n*M random normal variables	n*M
matrix multiplication $X\% * \%C$	(2n-1)M*n
Total	$(2n-1)M * n + n * m + n^3/3$

Table 2: Flop count for simulation of multivariate normal random variable

Second, I will discuss the **evaluation of a multivariate normal pdf**. When I evaluate the multivariate normal pdf, first I take the choleski decomposition of Sigma. Then, I use the method of solving quadratic forms that we used during class. When I have something of the form $(x-\mu)\Sigma^{-1}(x-\mu)$, I can use Choleski decomposition (so that $CC^T = \Sigma$). Then I set $r = C^{-1}(x-\mu)$. Then, I have that $r^T r = (x-\mu)\Sigma^{-1}(x-\mu)$. Secondly, I also take advantage of Choleski decomposition for the calculation of the determinant, since these are my two main costs. I know that the square root of the determinant of Σ, or $|\Sigma|^{1/2}$, is equal to the product of the diagonal elements of C, so $|\Sigma| = \prod_{i=1}^{n} C_{ii}$. Therefore, these are my main operations.

So, in full for the second part, I have the following computation: $-0.5*n*log(2*\pi) - 0.5*log(prod(diag(C))) - 0.5 * t(r)\% * \%r$. This has 9 different "operations" to consider, but I have not accounted for the fact that some of these operations involve matrices/vectors. Specifically, diag(C) involves n flops as it is the multiplication of diagonal elements of a matrix. Also, we know that r is a nx1 vector. Therefore, (1xn)x(nx1) involves n flops.

Operation	Flop Count
$chol(\Sigma)$	used from above
$-0.5 * n * log(2 * \pi) -$	4 flops
$0.5 * log(prod(diag(C))) -$	1 + n +1 = n+2 flops
$0.5 * r^T r$	$1 + (2n - 1) = 2n$ flops
Total per X	$3n + 6$ flops
Total for M simulations	$M(3n + 6)$ flops

Table 3: Flop count for evaluation of MVN pdf

Bringing these together, we get the idea of the total computational cost of this algorithm. The total number of flops for the algorithm that combines these two tasks is approximately $M(3n+6)+n^3/3+(2n-1)Mn+n*M$.

(e) Plot the theoretical cost of your algorithm as you increase n. Do this in terms of flops (floating point operations). How does the computation scale here when compared to the plot you produced above? (Try to make the plots easy to compare.) Do you have any explanation for differences between this plot and the above plot? Note: the discussion becomes more interesting as n gets large.

Based on our calculations in part 1d, we plot the number of flops as n increases below in Figure 3. Compared to Figure 2 where we plotted the CPU time vs. n instead of the flops, we see that there is a very similar and almost identical scaling issue. There are not many differences between the two plots at all, but potential differences may be due to the fact that memory use is not taken into account with the flop count. However, it appears that the CPU time scales very similarly to the theoretical cost determined by flops.

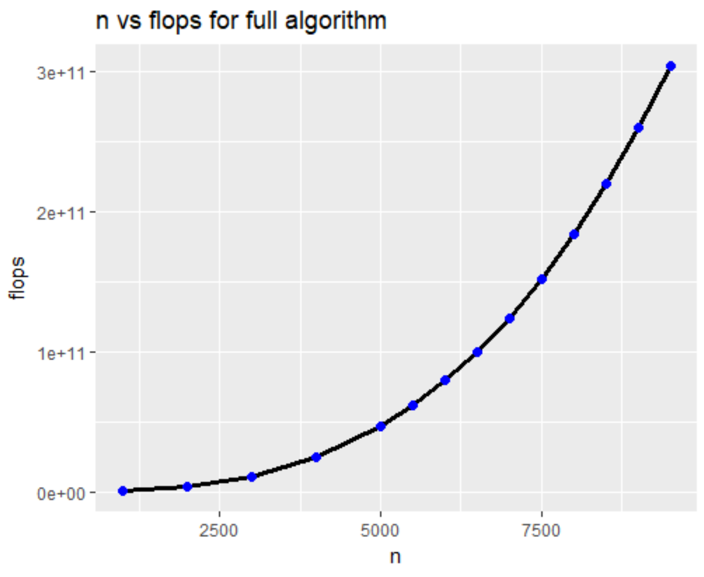


Figure 3: n vs flops

Problem 2 **Monte Carlo methods for random matrices: Consider the following approach for generating random covariance matrices of dimension mxm: Simulate m^2 N(0, 1) random variates to obtain an mxm matrix R. Obtain the positive definite covariance matrix $\Sigma = RR^T$. Note that this is a way to simulate a matrix with a particular Wishart distribution. Read through all the parts below before beginning to work on this problem.**

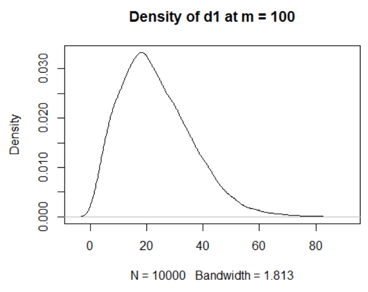
a) Let $d_i = \lambda_i - \lambda_{i+1}$, where λ_i is the ith eigenvalue. What are the expected values of d_1 and d_2? Report Monte Carlo standard errors and the sample size you used to approximate the expected value in each case. Also plot approximate density plots for d_1 and d_2. Do this for m = 100, m = 1000, m = 10,000. Write a brief (1-2 sentence) summary of what you have learned.

In Table 4 below, we see the given size of the matrix, which is specified by m where the matrix is mxm, and the Monte Carlo sample size, which is specified by n. As m increases, it becomes less feasible to have multiple Monte Carlo simulations. That is why n is not constant throughout the three different levels of m.

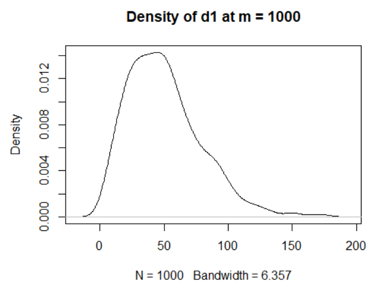
m	n	E(d_1)	MC std err, d_1	E(d_2)	MC std err, d_2
100	10,000	23.41694	0.1270963	17.3578	0.09431562
1,000	1,000	50.34244	0.9126475	39.95161	0.6864529
10,000	100	95.46987	6.079568	91.83154	4.672586

Table 4: Expected Value of d_1 and d_2 with Monte Carlo standard errors

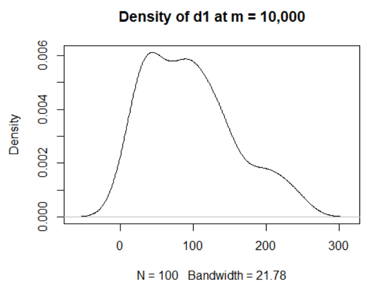
Below, I have included the approximate density plots for d_1 for all three values of m in Figure 4 and for d_2 in Figure 5.



(a) m=100

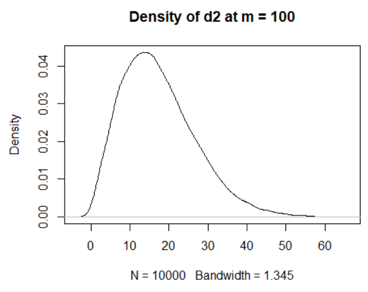


(b) m=1,000

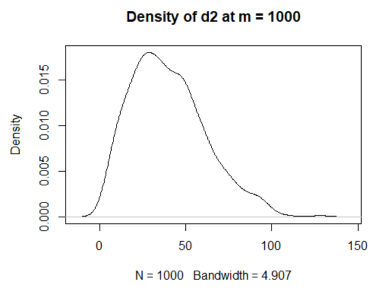


(c) m=10,000

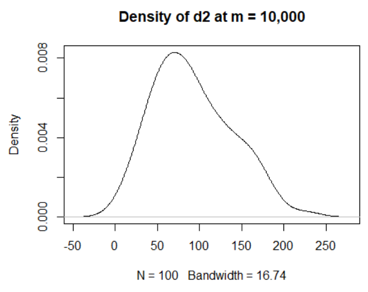
Figure 4: Approximate Densities for d_1



(a) m=100



(b) m=1,000



(c) m=10,000

Figure 5: Approximate Densities for d_2

After examining Figures 4 and 5 and Table 4, we see that as m increases, or the size of the matrix R and therefore Σ, so does the difference between the difference between the largest two eigenvalues (as well as the values of the eigenvalues themselves). This is also true for the difference between the second eigenvalue and the third (d_2) as it also increases with m. We also notice that the value of d_2 (the difference between the second and third eigenvalues) is consistently smaller than d_1 (the difference between the first and second eigenvalues) regardless of the value of m. It is hard to draw too many conclusions based on the standard error because we used different numbers of Monte Carlo samples, but the distributions of d_1 and d_2 look roughly the same throughout, being slightly right-skewed.

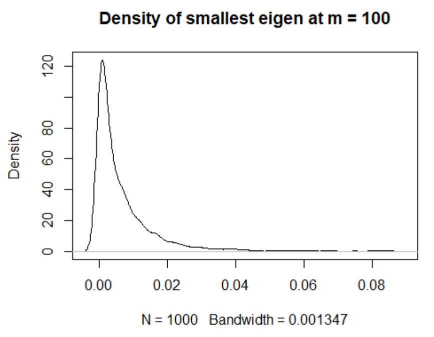
b) What is the expected value of the smallest eigenvalue of Σ? Plot approximate density plots for the smallest eigenvalue. Report Monte Carlo standard errors and the sample size you used in each case. Do this for m = 100 and m = 1,000.

Below, in Table 5, I have included the expected value of the smallest eigenvalue of Σ as well as the Monte Carlo standard error, calculated as the sample standard error over the square root of the Monte Carlo sample size. In both of these cases, I have used a Monte Carlo Sample size of 1,000, and the MC standard error is relatively low for both of the estimates.

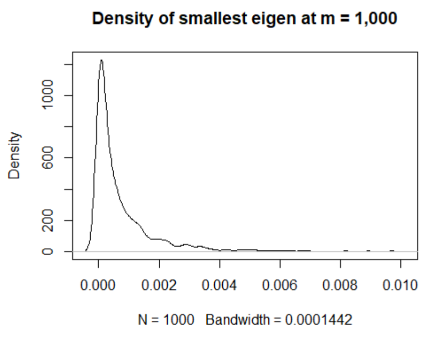
m	E(λ_m)	MC std err, λ_m
100	0.006928206	0.000334488
1,000	0.0007337275	3.578432e-05

Table 5: Expected Value of d_1 and d_2 with Monte Carlo standard errors

I have also included the density plots for λ_m for both values of m, 100 and 1,000. From comparing Figure 6a to Figure 6b, we see that generally the smallest eigenvalue of Σ is smaller when m is larger. We seek to verify this is the following problem, part c, when we analyze the relationship between m and n.



(a) m=100



(b) m=1,000

Figure 6: Approximate Densities for λ_m

c) Now study how the expected smallest eigenvalue changes as a function of m. (i) Plot approximate E(λ_m), expected value of smallest eigenvalue, versus m. (ii) Report your grid of m values. (iii) What is the largest m value for which you approximated the expectation?

For each of these estimates, I used 100 Monte Carlo samples. Below, in Figure 7, I have included a plot of E(λ_m), expected value of smallest eigenvalue, versus m. In terms of the grid of m values, I used values from 10 to 1,000, every 10 digits, and then from 1,000 to 3,000 every 500 digits, as these get to be much more tedious computations, especially when finding the smallest eigenvalue. The largest m value for which I approximated the expectation, using again the 100 Monte Carlo samples, was 3,000. However, we see in Figure 7 that the values for the smallest eigenvalue decay quite rapidly with m being greater than a couple hundred.

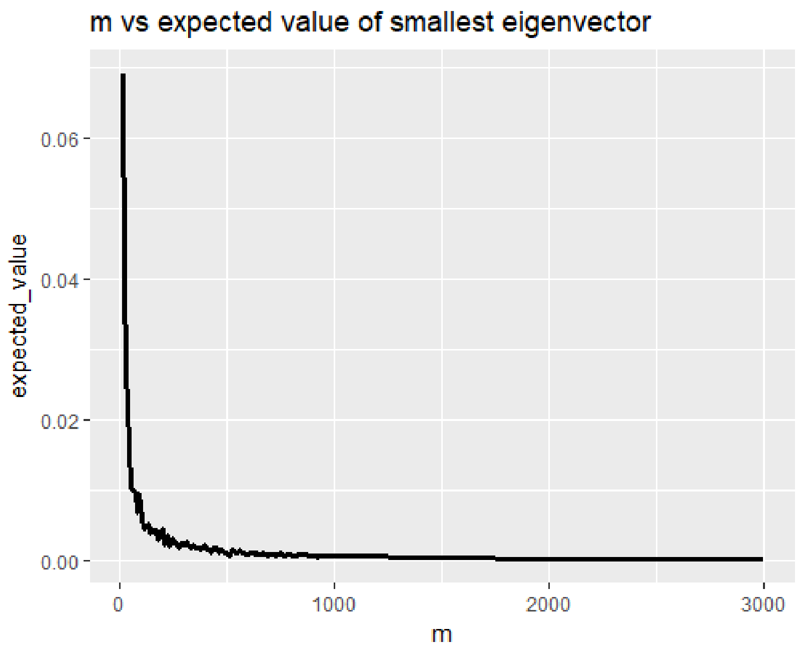
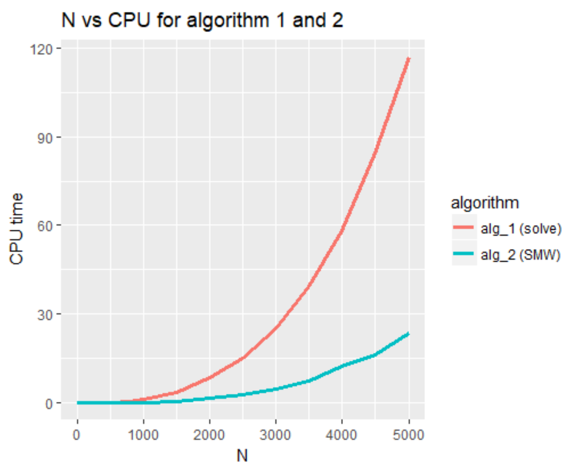


Figure 7: Relationship between m and λ_m

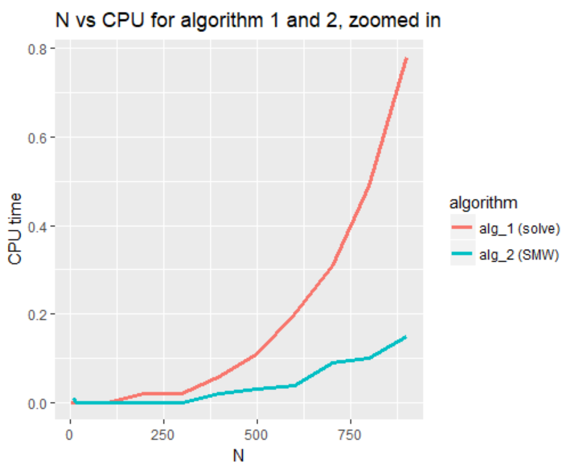
Problem 3 **Matrix inversion: Consider matrices of the form $\Sigma = \sigma I + KK^T$ where K is an NxM matrix of iid N(0,1) random variates and σ= 0.2. Fix M = 10. Compute the inverse of the matrix using two different algorithms: (i) directly by using the solve function and (ii) using the Sherman-Morrison-Woodbury identity discussed in class. I have provided example code for simulating the matrix and timing the solve function online. Report the following:**

(a) Plot the CPU time versus N for algorithm 1 and algorithm 2 (you will have to determine the grid and how large you can go with N);

Below, in Figure 8, we see that solve definitely takes more time than the Sherman-Morrison-Woodbury method. We see generally the same trend when we consider only 0 to 1,000 for N as if we use 0 to 5,000. They generally take minimal time when N is very small (less than 100).



(a) Full Scope

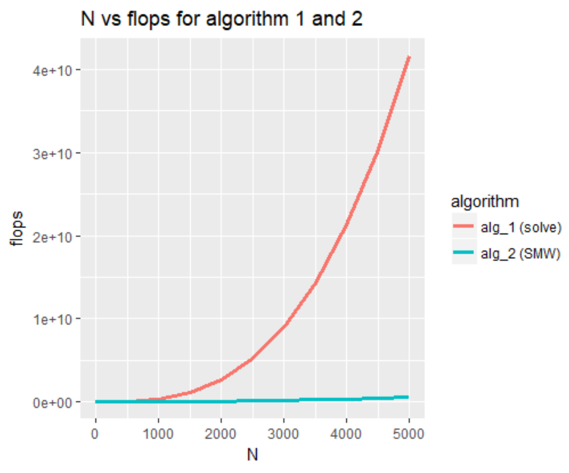


(b) Zoomed in

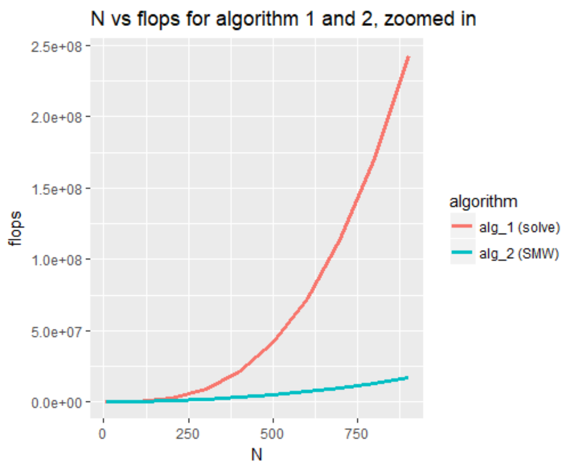
Figure 8: N vs CPU time

(b) a plot of floating point operations (flops) versus N for algorithm 1 and algorithm 2;

In Figure 9 we see that once again solve tends to be much slower than the Sherman-Morrison-Woodbury methods, especially at large values of N, or high dimensions of the matrix.



(a) Full Scope



(b) Zoomed in

Figure 9: N vs flops

In Tables 6 and 7 below, you can see exactly how Figure 9 was created. I calculated the flops for the Sherman Morrison Woodbury method through matrix multiplication, addition/subtraction and inversion. The calculations are shown in detail, and I believe them to be exact.

For the solve function, although I was unable to find very detailed documentation, I did find that in general most sources agree that to take an inverse of a dense matrix, there are $n^3/3$ flops.

Operation	Flop Count
$solve(\Sigma)$	$n^3/3$ flops
Total	$n^3/3$ flops

Table 6: Flop count for Algorithm 1 (solve)

Operation	Details	Flop Count
$S_A = A^{-1}$	inverse of diagonal	N
$S_C = C^{-1}$	inverse of diagonal	M
first = S_A*U	diagonal$_{N\times N}$ $*$ dense$_{N\times M}$	N*M
middle = S_C + (V*S_A*)*U	diagonal$_{M\times M}$+ (dense$_{M\times N}$ $*$ diagonal$_{N\times N}$) $*$ dense$_{N\times M}$	$M^2 + M * N + M^2 * (2N-1)$
last = V*S_A	dense$_{M\times N}$ $*$ diagonal$_{N\times N}$	M*N
X = solve(middle)	$solve(dense_{M\times M})$	$M^3/3$
Y = first*X*last	$dense_{N\times M} * dense_{M\times M} * dense_{M\times N}$	$(2M-1) * N * M + (2M-1)N^2$
S_A - Y	$diagonal_{N\times N} - dense_{N\times N}$	N^2
Total		$N + M + 3NM + M^2 + N^2 + M^3/3 +$ $(2N-1)M^2 + (2M-1)N^2 + (2M-1)NM$

Table 7: Flop count for Algorithm 2 (SMW)

(c) briefly summarize what you observe based on a comparison of the computational costs for the two algorithms using flops and using CPU time.

We see with Figure 8 and Figure 9, that the computational costs in terms of flops and the computational costs in terms of CPU time seems to be somewhat similar. We see that the CPU time seems to grow faster for Sherman-Morrison-Woodbury in terms of CPU time than in terms of flops. This may because of computation restraints such as memory that are not reflected in the number of flops. My computer performs slower when it is using up quite a bit of memory, which it does when N is large.

However, I recognize that I also might not have the exact right calculations for the flop count for the Sherman-Morrison-Woodbury method. If I add another multiple of N^2 to the flop count of the SMW method (about $100N^2$), then it results in similar image as Figure 8 in terms of the comparison between the two methods. Therefore, I believe it is safe to say that the flop count for the SMW method is of the order N^2 while the flop count for the solve method is of the order N^3. Therefore, we conclude that when we can deconstruct the matrix into a form to use SMW, it is much more efficient.

Problem 4 **A simple simulation study: Let $X_1, ..., X_n$ iid with common distribution Poi(λ). Now consider constructing 95% confidence intervals for λ. Denote $\hat{\lambda} = \sum_{i=1}^n X_i/n$ and $\hat{\sigma}^2$ is the sample variance. Here are three confidence intervals:**

1. $\hat{\lambda} \pm 1.96\sqrt{\hat{\lambda}}/\sqrt{n}$ using the sample mean to directly estimate the variance/standard error.
2. $\hat{\lambda} \pm 1.96\hat{\sigma}/\sqrt{n}$ using sample standard deviation to estimate the variance/standard error.
3. Using the fact that we can calculate quantiles for the Poisson distribution (R command: qpois), construct an "exact" 95% interval for λ.

Construct the above confidence intervals for two different sample sizes, n = 10; 200, and two different λ values, λ = 2; λ = 50. You should report a 4x3 table (rows are the different scenarios and the columns are the three different confidence intervals) that contains the estimated coverage rates along with Monte Carlo standard errors for each coverage rate. Write a brief summary of what you have found, and which confidence interval approach you would recommend.

For this problem, we simulated 10 million Monte Carlo samples using the given Poisson sample sizes and λ values. We found that in some instances, namely when using the Poisson quantiles, the coverage rate was 100%. This can be seen in Table 1, below. The Monte Carlo standard error is very very low (essentially 0) for many of these simulations due to the large number of Monte Carlo samples.

		Confidence Interval Type					
		$\hat{\lambda} \pm 1.96\sqrt{\hat{\lambda}}/\sqrt{n}$		$\hat{\lambda} \pm 1.96\hat{\sigma}/\sqrt{n}$		Poisson quantiles	
n	λ	cov rate	MC std err	cov rate	MC std err	cov rate	MC std err
10	2	0.9474357	1e-04	0.9099012	1e-04	0.999745	1e-04
200	2	0.9729287	1e-04	0.9479918	1e-04	1	0
10	50	0.9779149	1e-04	0.9178852	1e-04	1	0
200	50	0.9756092	1e-04	0.9486439	1e-04	1	0

Table 8: Coverage Rates and MC Standard Errors for 3 CI Types

In order to determine which confidence interval approach we would recommend, we also consider the width of the confidence interval. If a confidence interval is too wide, then it might not be informative. In Table 9, we see the average width across all 10 million Monte Carlo samples in each scenario. We see that the confidence intervals are pretty wide for most of the cases when n =10 as opposed to n = 200. However, when we use the Poisson quantiles, the width pretty much stays the same (as this does not depend on n).

Average Length of Confidence Interval				
		Confidence Interval Type		
n	λ	$\hat{\lambda} \pm 1.96\sqrt{\hat{\lambda}}/\sqrt{n}$	$\hat{\lambda} \pm 1.96\hat{\sigma}/\sqrt{n}$	Poisson quantiles
10	2	2.1106515	1.6976533	5.082966
200	2	0.4731303	0.3913922	5.022727
10	50	35.371847	8.5246524	27.658477
200	50	7.9096434	1.9575740	27.663272

Table 9: Average Length for 3 CI Types

After considering the data presented in these two tables, we consider the first type of confidence interval to be the most effective, especially when we are able to make many draws from the distribution (in this case Poisson) to create our estimate of λ. We see that this type has a much higher coverage rate than the second type consistently, though being slightly longer in terms of average width. However, the Poisson quantiles are not particularly informative as they are so wide. Therefore, we suggest using the first method of creating confidence intervals with as many as possible samples form the distribution to create the estimate of lambda.

Problem 5

Let $X_1, ..., X_n$ iid from Gamma(α, β) distribution. Consider Bayesian inference for α, β with prior distributions $\alpha \sim$ N(0, 3), $\beta \sim$N(0, 3). Use a Laplace approximation (as discussed in class) to approximate the posterior expectation of α, β. Show your work, and provide pseudocode for your algorithm. You will likely need to use the R function optim to carry out numerical optimization. The function is well documented.

For this problem, I follow the class notes and the paper *Accurate Approximations for Posterior Moments and Marginal Densities* (Tierney and Kadane, 1986) that was mentioned in class. In this paper, they mention the approximation to $E_n[g]$ in the multi-parameter case:

$$\hat{E}_n[g] = \left(\frac{det(\Sigma^*)}{\det(\Sigma)}\right)^{1/2} exp(n(L^*(\hat{\theta}^*) - L(\hat{\theta}).$$

This is where $\hat{\theta}^*$ and $\hat{\theta}$ maximize L^* and L respectively and Σ^* and Σ are "minus the inverse Hessians of L^* and L at $\hat{\theta}^*$ and $\hat{\theta}$ respectively.

I will be referring to the structure of the problem that we developed in class, where we are trying to approximate $E_\pi(g(\theta))$ where $\pi(\theta|x) = \frac{L(x|\theta)}{\int L(x|\theta)p(\theta)d\theta}$, or the posterior. So, we are trying to approximate $\int g(\theta)\pi(\theta|x)d\theta = \frac{\int g(\theta)exp(l_n(\theta)+l_p(\theta))d\theta}{\int exp(l_n(\theta)+l_p(\theta))d\theta}$ where $l_n(\theta) = log(L(x|\theta))$ and $l_p(\theta) = log(p(\theta))$. We will also re-parameterize $h(\theta) = log(g(\theta))$ in the next step.

We know that the numerator,

$$\int exp(h(\theta) + l_n(\theta) + l_p(\theta))d\theta$$

is approximated by

$$exp(r(\widetilde{\theta}))|(r''(\widetilde{\theta}))^{-1}|^{-1/2} * (2\pi)^{n/2} = *$$

where $r(\theta) = h(\theta) + l_n(\theta) + l_p(\theta)$ and $\widetilde{\theta}$ maximizes $r(\theta)$.

Similarly, we know that the denominator ,

$$\int exp(l_n(\theta) + l_p(\theta))d\theta$$

is approximated by

$$exp(l_n(\hat{\theta}) + l_p(\hat{\theta}))|(l''_n(\hat{\theta}) + l''_p(\hat{\theta}))^{-1}|^{-1/2} * (2\pi)^{n/2} = **$$

$$\text{where } \hat{\theta} \text{ maximizes } l_n(\theta) + l_p(\theta).$$

So, the Laplace approximation just is the estimate for th enumerator divided by the estimate for the denominator, or $\frac{*}{**}$.

The following pseudocode represents the algorithm that I used to implement a Laplace approximation to approximate the posterior expectation of α and β.

1. Give initial values for α and β.
2. Find the values of the parameters (α, β) that maximize the denominator, given the data and initial values (using optim), $(\hat{\alpha}, \hat{\beta})$.
3. Find the values of the parameters (α, β) that maximize the numerator, given the data and initial values (using optim), $(\widetilde{\alpha}, \widetilde{\beta})$.
4. Using the Hessian estimates for each, find $|(r''(\widetilde{\alpha}, \widetilde{\beta}))^{-1}|^{-1/2}$ and $|l_p''(\hat{\alpha}, \hat{\beta}))^{-1}|^{-1/2}$ for the numerator and denominator, respectively.
5. Using $g(\alpha, \beta) = \alpha$, find the estimate of α using the ratio described above, $\frac{*}{**}$, by using the Hessian and the evaluation of the approximation of the numerator and the denominator at $(\widetilde{\alpha}, \widetilde{\beta})$ and $(\hat{\alpha}, \hat{\beta})$, respectively.
6. Similarly, using $g(\alpha, \beta) = \beta$, find the estimate of β using the ratio described above, $\frac{*}{**}$, by using the Hessian and the evaluation of the approximation of the numerator and the denominator at $(\widetilde{\alpha}, \widetilde{\beta})$ and $(\hat{\alpha}, \hat{\beta})$, respectively.

After running this algorithm, I find the posterior estimate of α to be approximately 2.953634 and the posterior estimate of β to be approximately 0.5165954.

We were also asked to consider the computational costs for this problem. At this dimension, when the dimension of y is 200 data points, the computational cost is quite small. It takes between 0.10 and 0.14 seconds to approximate each of α and β.

However, when you increase the dimension of y, the computation time increases. We simulate a vector of random uniform variables that have the range of our original data, y. We then go through this optimization procedure for each length of y.

In order to find the computational complexity of this algorithm, we do some research on the L-BFGS-B method for optimization that we used in this problem. We find two papers, "Limited-Memory Reduced-Hessian Methods for Large-Scale Unconstrained Optimization" (Gill and Leonard, 2003) and "Stochastic L-BFGS: Improved Convergence Rates and Practical Acceleration Strategies" (Zhao et al., 2017) that both say that the scale of the flops is roughly linear with the size of the data. We seek to verify this with simulations.

Although we were not able to find the *exact* number of flops in the optim function and others, we can see that in Figure 10 that it is probably of the order N (linear) where N is the dimension of the data. This agrees with the two papers that we have found above. However, we were able to increase the dimension of y to be 3 million data points and the computation still took less than 3 minutes. Therefore, this seems like a pretty computationally efficient procedure.

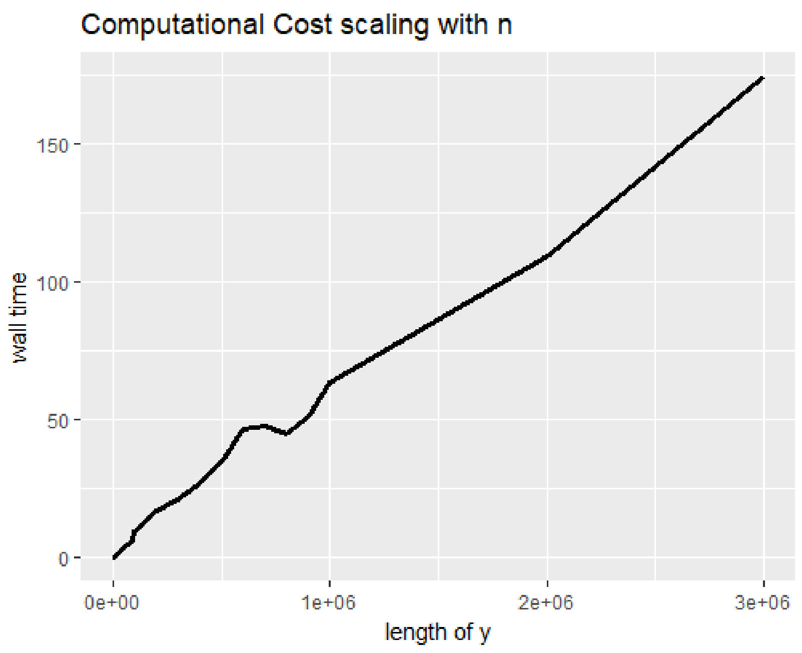


Figure 10: Relationship between length of y and CPU time

References

Philip E Gill and Michael W Leonard. 2003. Limited-memory reduced-Hessian methods for large-scale unconstrained optimization. SIAM Journal on Optimization 14, 2 (2003), 380–401.

Luke Tierney and Joseph B Kadane. 1986. Accurate approximations for posterior moments and marginal densities. Journal of the american statistical association 81, 393 (1986), 82–86.

Renbo Zhao, William Benjamin Haskell, and Vincent YF Tan. 2017. Stochastic L-BFGS: Improved Convergence Rates and Practical Acceleration Strategies. IEEE Transactions on Signal Processing (2017).