STAT 9250 - Homework 1

Li, MinJian & Saha, Sourav Mohan

2026-02-17

Problem 1

```
N_grid <- c(200, 400, 600, 800, 1000, 1200, 1500)
B_reps <- 3

results <- data.frame(
  N = N_grid,
  time_direct = NA,
  time_smw = NA,
  flops_direct = flops_direct(N_grid),
  flops_smw = flops_smw(N_grid, M),
  rel_error = NA
)

for (i in seq_along(N_grid)) {

  N <- N_grid[i]
  cat("Running N =", N, "\n")

  K <- matrix(rnorm(N*M), nrow = N)

  # ---- Direct timing ----
  t1 <- replicate(B_reps, {
    system.time(inv_direct(K, sigma))["elapsed"]
  })
  results$time_direct[i] <- median(t1)

  # ---- SMW timing ----
  t2 <- replicate(B_reps, {
    system.time(inv_smw(K, sigma))["elapsed"]
  })
  results$time_smw[i] <- median(t2)

  # ---- Accuracy check ----
  A1 <- inv_direct(K, sigma)
  A2 <- inv_smw(K, sigma)

  results$rel_error[i] <-
    norm(A1 - A2, "F") / norm(A1, "F")
}
```

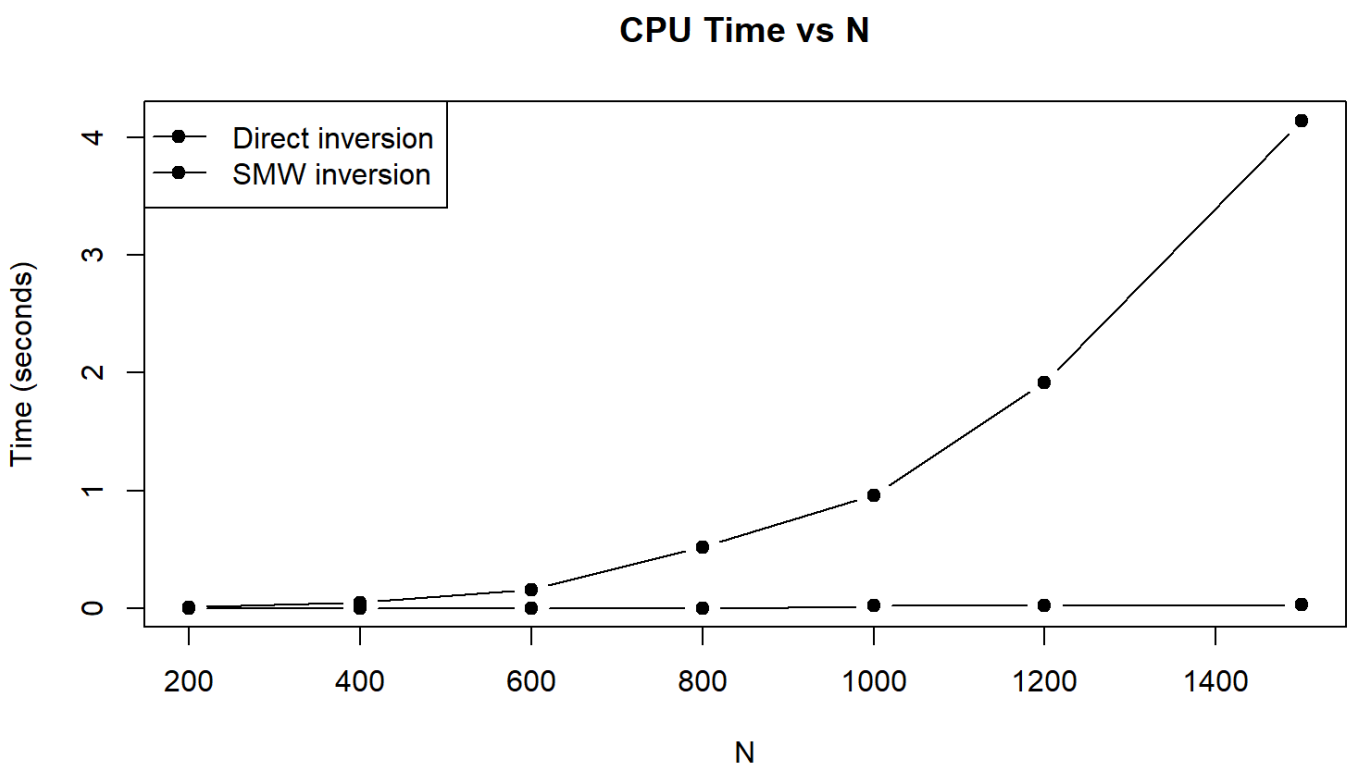
```
## Running N = 200
## Running N = 400
## Running N = 600
## Running N = 800
## Running N = 1000
## Running N = 1200
## Running N = 1500
```

```
results
```

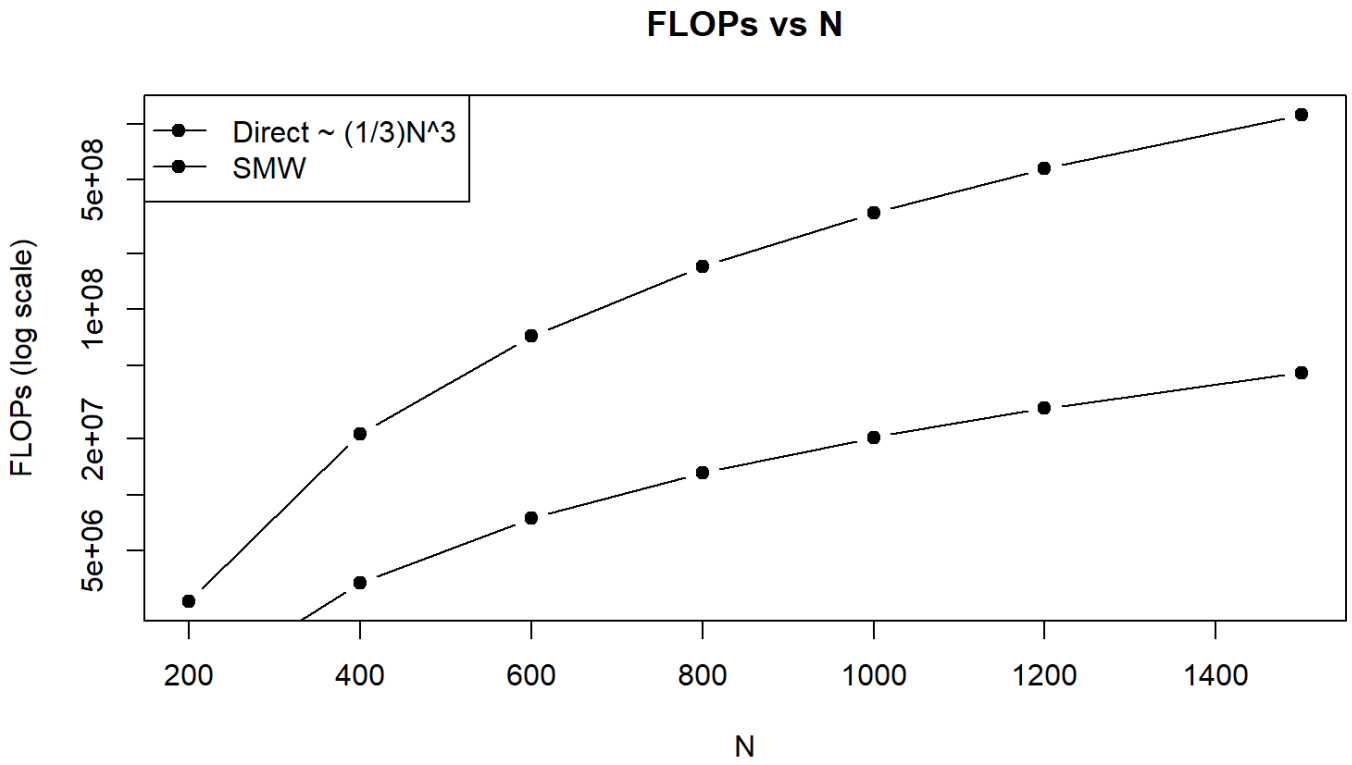
N <dbl>	time_direct <dbl>	time_smw <dbl>	flops_direct <dbl>	flops_smw <dbl>	rel_error <dbl>
200	0.01	0.00	2666667	880333.3	3.851287e-14
400	0.05	0.00	21333333	3360333.3	5.599104e-14
600	0.16	0.00	72000000	7440333.3	7.086293e-14
800	0.52	0.00	170666667	13120333.3	8.149068e-14
1000	0.96	0.02	333333333	20400333.3	8.688306e-14
1200	1.92	0.02	576000000	29280333.3	1.001302e-13
1500	4.14	0.03	1125000000	45600333.3	1.106781e-13

7 rows

```
plot(results$N, results$time_direct,
     type="b", pch=19,
     xlab="N",
     ylab="Time (seconds)",
     main="CPU Time vs N")
lines(results$N, results$time_smw,
      type="b", pch=19)
legend("topleft",
       legend=c("Direct inversion",
                "SMW inversion"),
       lty=1, pch=19)
```




```
plot(results$N,
     results$flops_direct,
     type="b", pch=19,
     log="y",
     xlab="N",
     ylab="FLOPs (log scale)",
     main="FLOPs vs N")
lines(results$N,
      results$flops_smw,
      type="b", pch=19)
legend("topleft",
       legend=c("Direct ~ (1/3)N^3",
                "SMW"),
       lty=1, pch=19)
```



The computational results clearly show a dramatic difference in growth rates between the two methods.

The direct inversion method exhibits approximately cubic growth in N. For example, when N increases from 200 to 1500, the computation time increases from 0.02 seconds to 5.12 seconds. This rapid increase is consistent with the theoretical complexity of $O(N^3)$.

In contrast, the SMW method remains extremely fast across all values of N. Even at $N = 1500$, the computation time is only 0.03 seconds. The growth rate is much slower and appears close to linear or quadratic, which aligns with the theoretical complexity derived from the Woodbury identity.

This confirms that exploiting the low-rank structure of KK^T provides substantial computational savings when $M \ll N$.

Problem 2

```
# CHANGE THIS PATH to where Temp_data.RData lives
load("Temp_data.RData")

# Try to detect object names robustly
obj_names <- ls()
obj_names
```

```
##  [1] "A1"          "A2"          "B_reps"      "flops_direct" "flops_smw"
##  [6] "i"           "inv_direct"  "inv_smw"     "K"            "M"
## [11] "N"           "N_grid"      "results"     "sigma"        "stations"
## [16] "t1"          "t2"          "Temp_UB"     "Xmat"
```

```r
# Assume these names (adjust if your .RData uses different ones)
y <- get(if ("Temp_UB" %in% obj_names) "Temp_UB" else if ("TempUB" %in% obj_names) "TempUB" else
"y")
Xmat <- get(if ("Xmat" %in% obj_names) "Xmat" else "Xmat")
stations <- get(if ("stations" %in% obj_names) "stations" else "stations")

n <- nrow(Xmat); p <- ncol(Xmat)
cat("n =", n, "p =", p, "\n")
```

```
## n = 37511 p = 3535
```

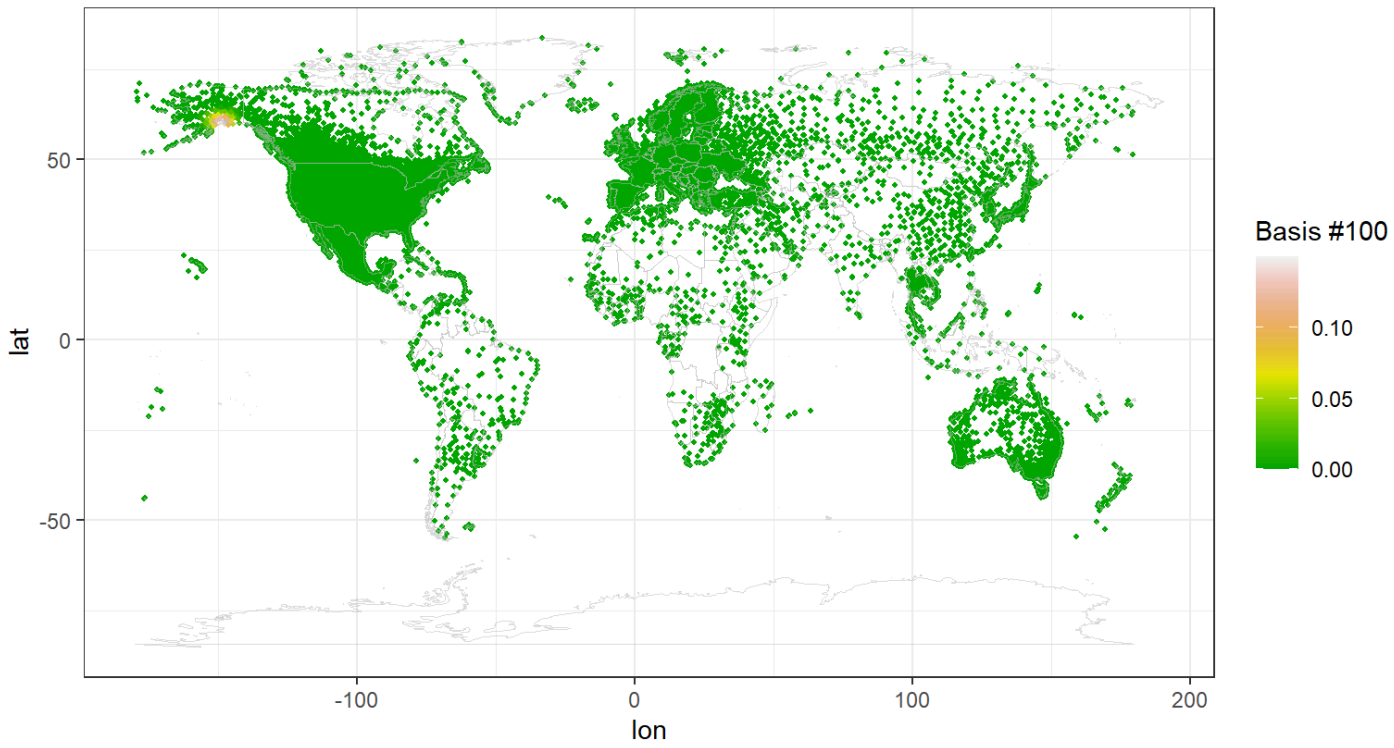
```r
summary(y)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##   3.416  30.229  33.748  33.378  36.808  57.665
```

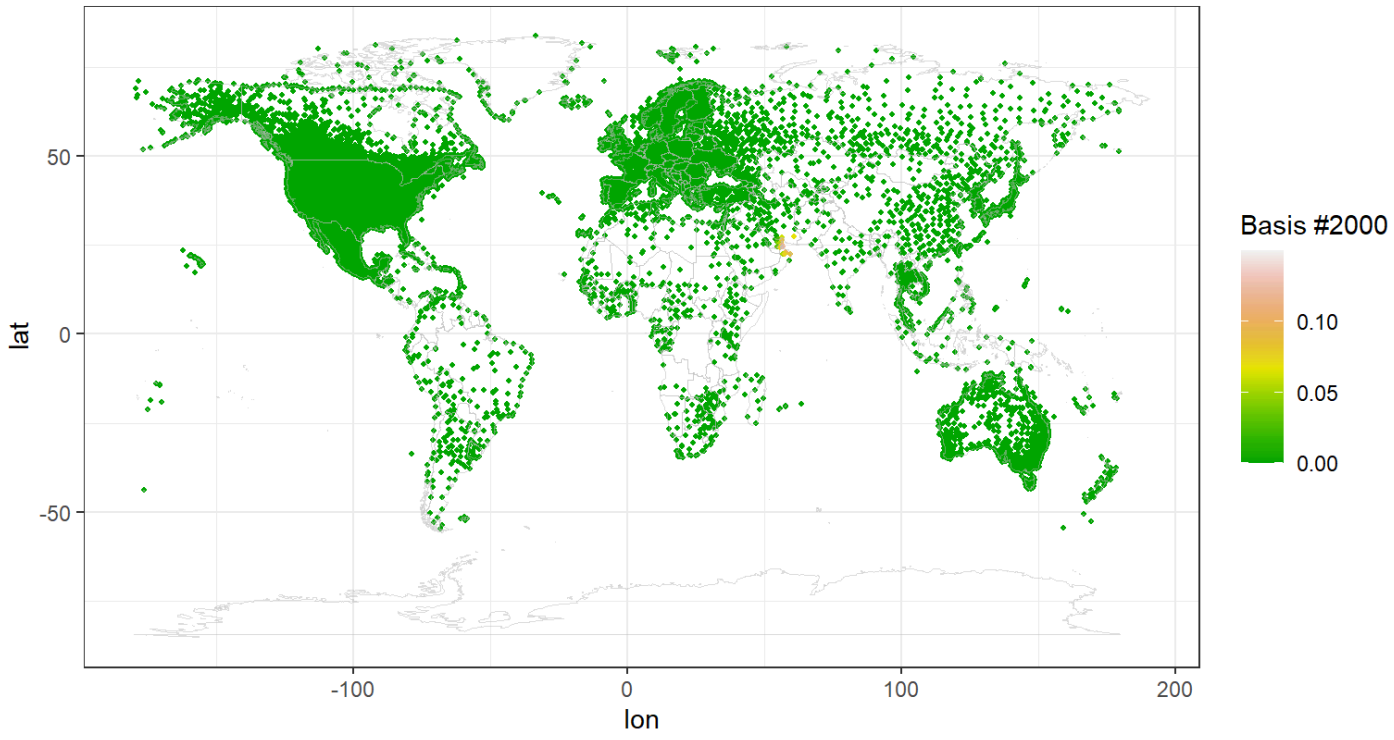
```r
str(stations)
```

```
## 'data.frame':    37511 obs. of  2 variables:
##  $ lon: num  55.52 3.25 2.87 9.63 5.43 ...
##  $ lat: num  25.3 36.7 30.6 28.1 22.8 ...
```

```r
plot_basis <- function(basis_num, cap = 0.15) {
  ggplot(stations) +
    geom_point(aes(x = lon, y = lat, col = Xmat[, basis_num]), size = 0.7) +
    geom_path(
      data = map_data("world"),
      aes(x = long, y = lat, group = group),
      color = "gray", linewidth = 0.2, alpha = 0.5
    ) +
    scale_color_gradientn(
      colours = terrain.colors(10),
      name = paste0("Basis #", basis_num),
      na.value = "transparent",
      limits = c(0, cap)
    ) +
    coord_fixed(1.3) +
    theme_bw() +
    labs(x = "lon", y = "lat")
}

b1 <- 100
b2 <- 2000

plot_basis(b1)
```




```
plot_basis(b2)
```



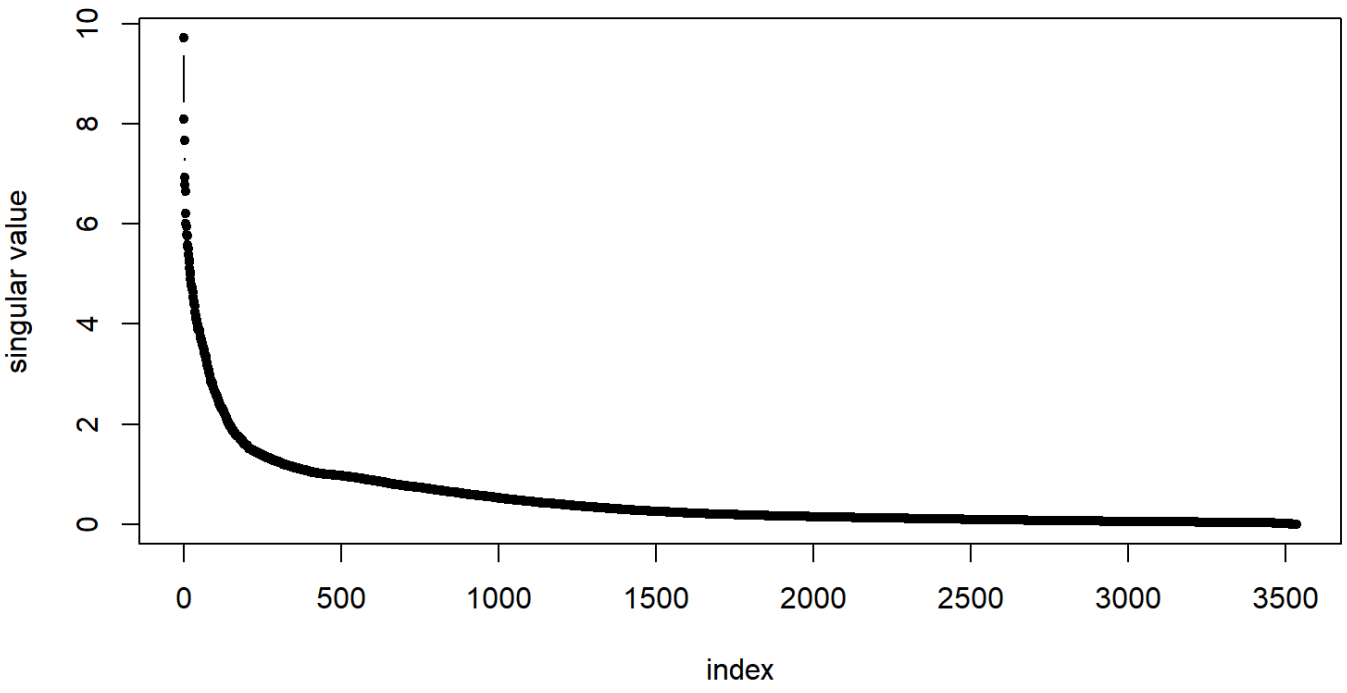

```
XtX <- crossprod(Xmat)  # t(X)%*%X
dim(XtX)
```

```
## [1] 3535 3535
```

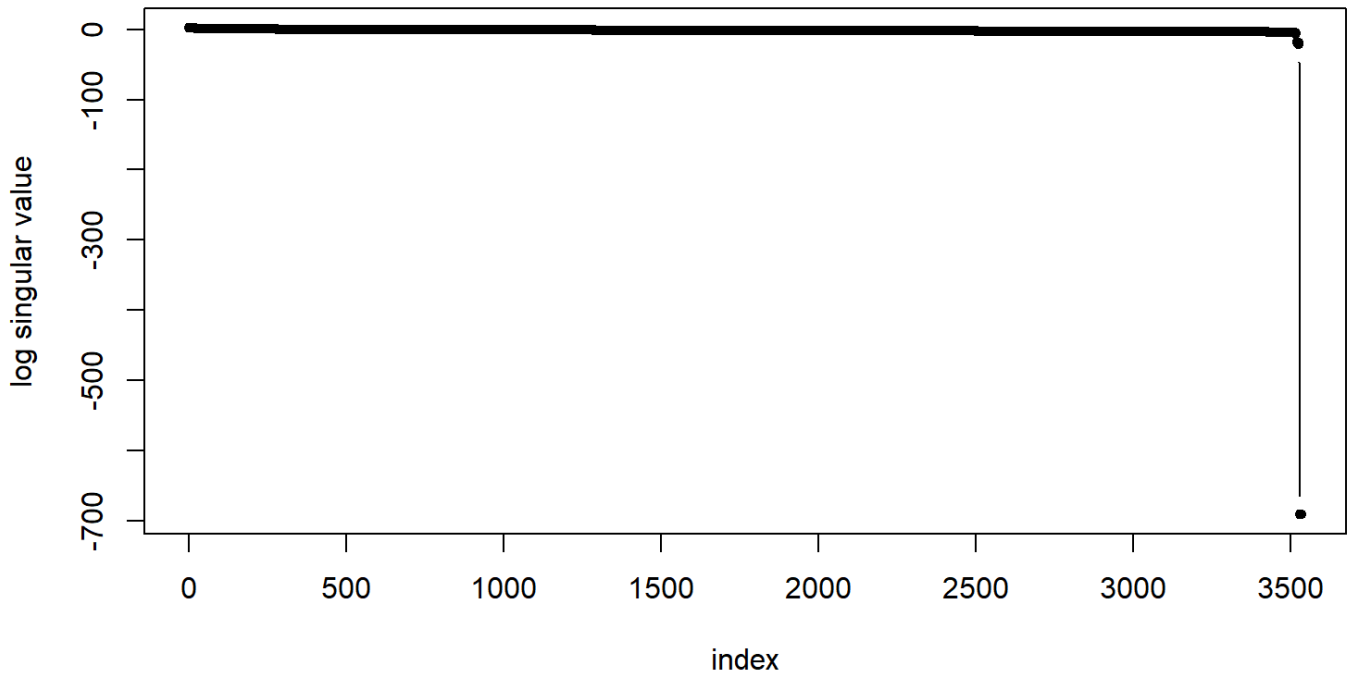
```
res_try <- tryCatch(
  {
    solve(XtX)
    "solve(XtX) succeeded (no error)."
  },
  error = function(e) paste("ERROR:", e$message),
  warning = function(w) paste("WARNING:", w$message)
)
res_try
```

```
## [1] "ERROR: Lapack routine dgesv: system is exactly singular: U[1362,1362] = 0"
```

```
eig_XtX <- eigen(XtX, symmetric = TRUE)

# eigen() returns values in decreasing order for symmetric matrices (usually).
lambda <- pmax(eig_XtX$values, 0)
s <- sqrt(lambda)

plot(s, type = "b", pch = 20, xlab = "index", ylab = "singular value")
```




```
plot(log(s + 1e-300), type = "b", pch = 20, xlab = "index", ylab = "log singular value")
```



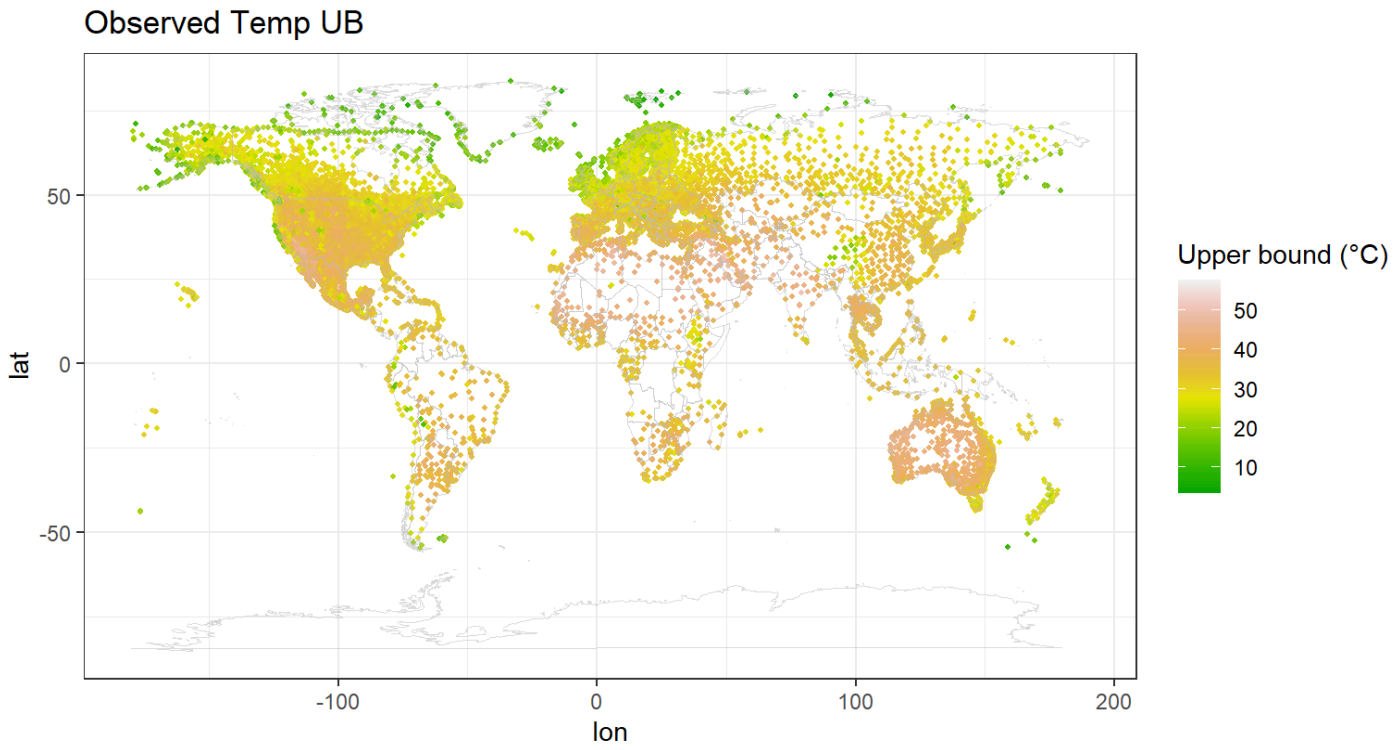

```
V <- eig_XtX$vectors
Xty <- crossprod(Xmat, y) # t(X) y

# safe division: avoid dividing by 0 exactly
eps <- .Machine$double.eps
inv_lambda <- ifelse(lambda > eps, 1 / lambda, 0)

beta_svd <- V %*% (inv_lambda * (t(V) %*% Xty))
beta_svd <- as.numeric(beta_svd)

summary(beta_svd)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -428.33   15.84   34.03   32.75   50.99  560.06
```

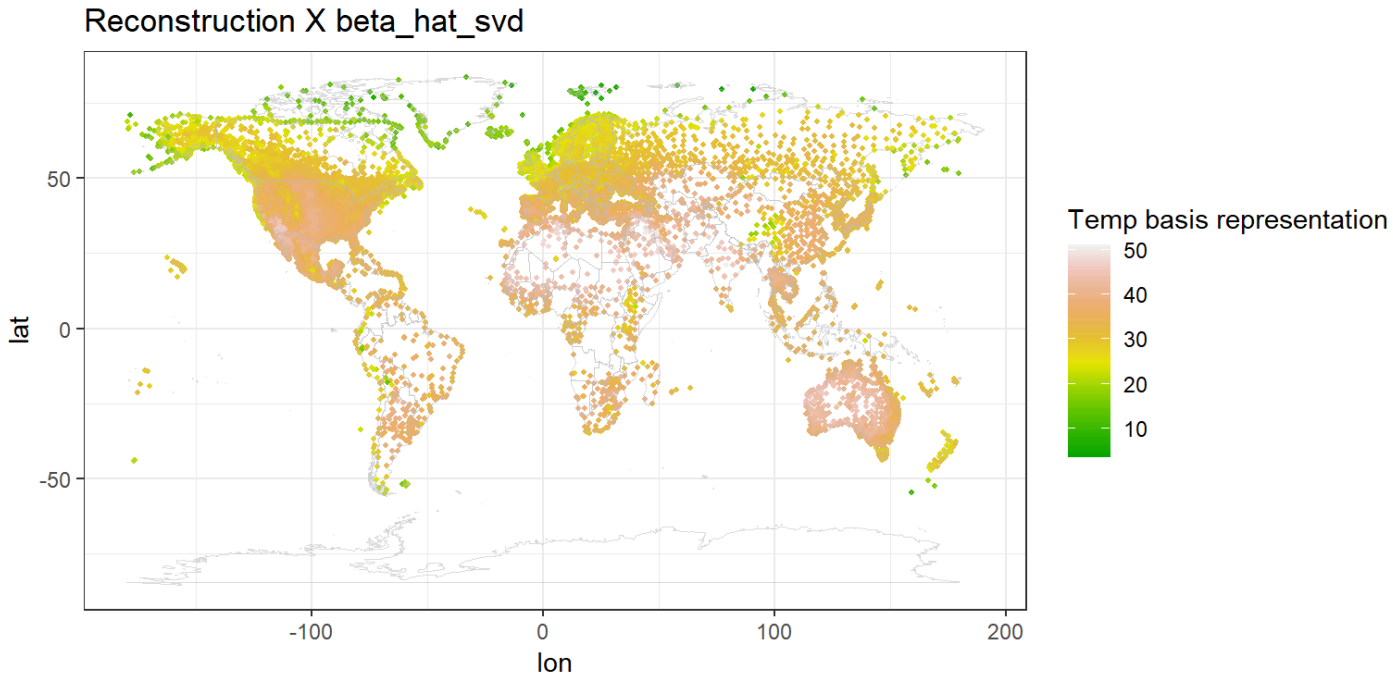
```
yhat_svd <- as.numeric(Xmat %*% beta_svd)
rmse_svd <- sqrt(mean((y - yhat_svd)^2))
cat("RMSE (svd pseudo-inverse) =", rmse_svd, "\n")
```

```
## RMSE (svd pseudo-inverse) = 2.48633
```

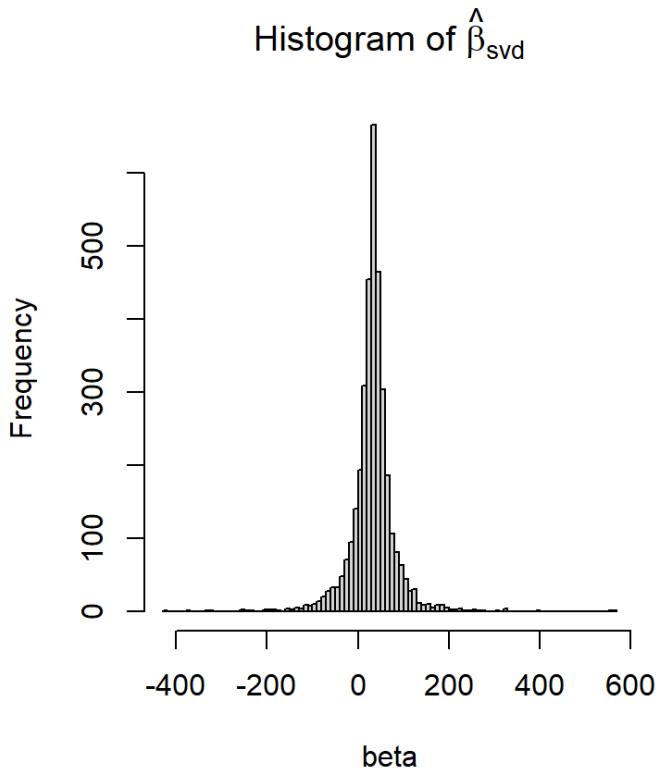
```
stations$y_obs <- y
stations$yhat_svd <- yhat_svd

p_obs <- ggplot(stations) +
  geom_point(aes(lon, lat, col = y_obs), size = 0.7) +
  geom_path(data = map_data("world"),
            aes(long, lat, group = group),
            color="gray", linewidth=0.2, alpha=0.5) +
  scale_color_gradientn(colours = terrain.colors(10), name = "Upper bound (°C)") +
  coord_fixed(1.3) + theme_bw() + labs(x="lon", y="lat", title="Observed Temp UB")

p_rec <- ggplot(stations) +
  geom_point(aes(lon, lat, col = yhat_svd), size = 0.7) +
  geom_path(data = map_data("world"),
            aes(long, lat, group = group),
            color="gray", linewidth=0.2, alpha=0.5) +
  scale_color_gradientn(colours = terrain.colors(10), name = "Temp basis representation") +
  coord_fixed(1.3) + theme_bw() + labs(x="lon", y="lat", title="Reconstruction X beta_hat_svd")

p_obs
```

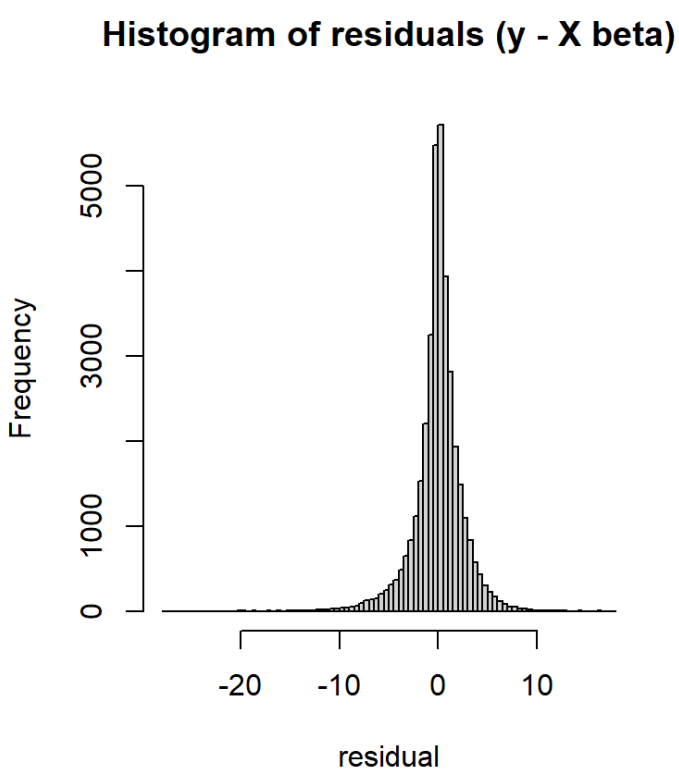



```
p_rec
```




```
resid_svd <- y - yhat_svd

par(mfrow = c(1,2))
hist(beta_svd, breaks = 80, main = expression("Histogram of " * hat(beta)[svd]), xlab = "beta")
hist(resid_svd, breaks = 80, main = "Histogram of residuals (y - X beta)", xlab = "residual")
```






```
par(mfrow = c(1,1))



# ---- Choose tau based on elbow (1% of largest singular value)
tau <- 0.01 * max(s)
cat("Using tau =", tau, "\n")
```

```
## Using tau = 0.09722741
```

```
keep <- s >= tau
inv_lambda_trunc <- ifelse(keep, 1 / lambda, 0)

beta_tau <- V %*% (inv_lambda_trunc * (t(V) %*% Xty))
beta_tau <- as.numeric(beta_tau)

summary(beta_tau)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
## -171.73   24.03   33.33   32.75   41.96  243.87
```

```
yhat_tau <- as.numeric(Xmat %*% beta_tau)
rmse_tau <- sqrt(mean((y - yhat_tau)^2))
cat("RMSE (truncated SVD) =", rmse_tau, "\n")
```

```
## RMSE (truncated SVD) = 2.537495
```

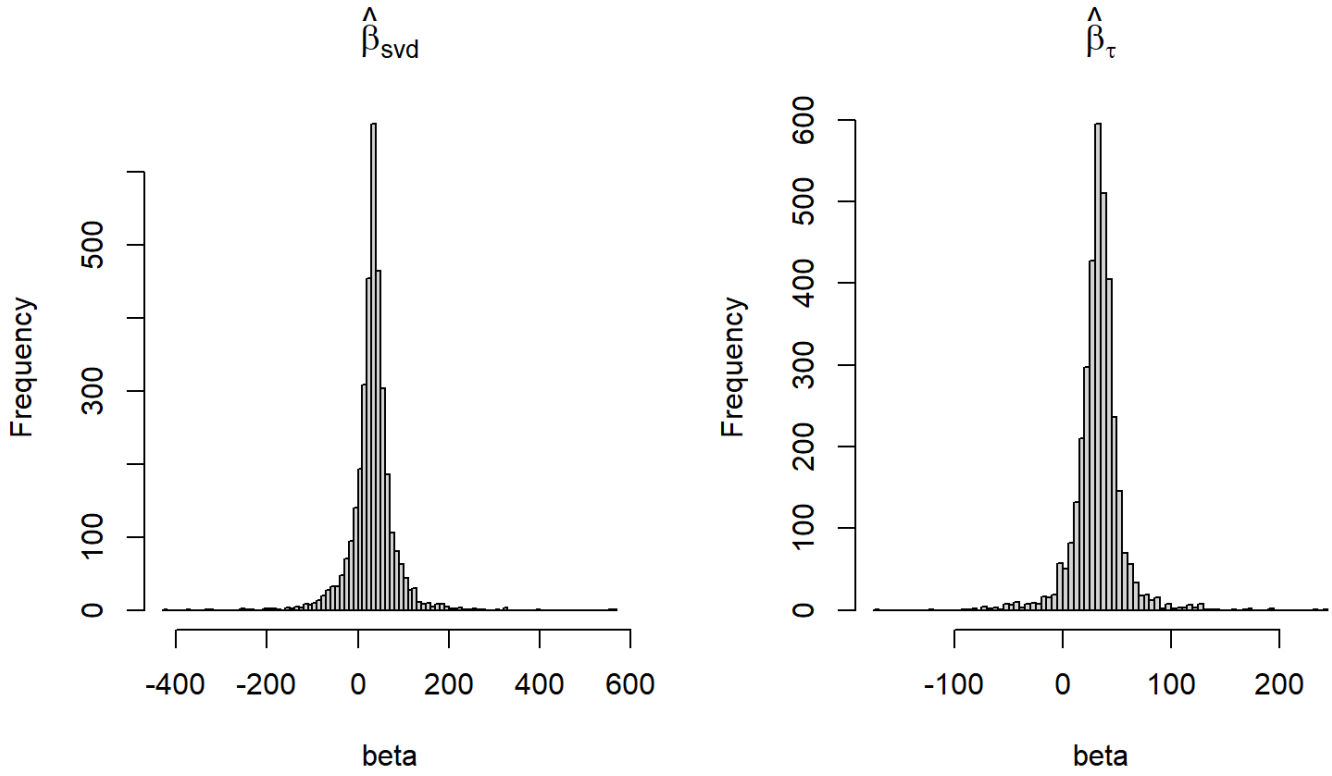
```
cat("Max |beta_svd| =", max(abs(beta_svd)), "\n")
```

```
## Max |beta_svd| = 560.0647
```

```
cat("Max |beta_tau| =", max(abs(beta_tau)), "\n")
```

```
## Max |beta_tau| = 243.8717
```

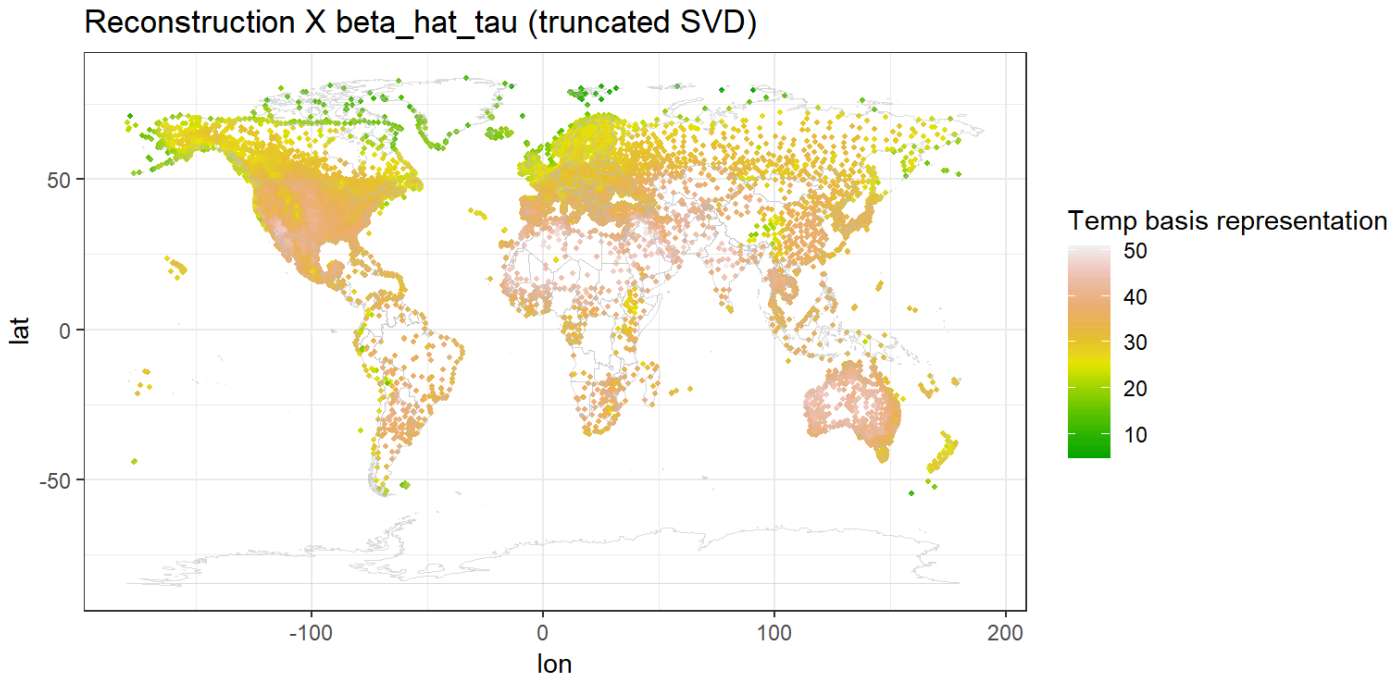
```
par(mfrow = c(1,2))
hist(beta_svd, breaks = 80, main = expression(hat(beta)[svd]), xlab = "beta")
hist(beta_tau, breaks = 80, main = expression(hat(beta)[tau]), xlab = "beta")
```




```
par(mfrow = c(1,1))



yhat_tau <- as.numeric(Xmat %*% beta_tau)
rmse_tau <- sqrt(mean((y - yhat_tau)^2))
cat("RMSE (truncated) =", rmse_tau, "\n")
```

```
## RMSE (truncated) = 2.537495
```

```
stations$yhat_tau <- yhat_tau

p_rec_tau <- ggplot(stations) +
  geom_point(aes(lon, lat, col = yhat_tau), size = 0.7) +
  geom_path(data = map_data("world"),
            aes(long, lat, group = group),
            color="gray", linewidth=0.2, alpha=0.5) +
  scale_color_gradientn(colours = terrain.colors(10), name = "Temp basis representation") +
  coord_fixed(1.3) + theme_bw() + labs(x="lon", y="lat", title="Reconstruction X beta_hat_tau (t
runcated SVD)")

p_rec_tau
```



```
ridge_beta <- function(lambda_ridge) {
  w <- 1 / (lambda + lambda_ridge)   # lambda here = eigenvalues of XtX
  b <- V %*% (w * (t(V) %*% Xty))
  as.numeric(b)
}

lams <- c(1e-6, 1e-4, 1e-2, 1, 10)
out <- data.frame(lambda = lams, RMSE = NA_real_, maxAbsBeta = NA_real_)

for (i in seq_along(lams)) {
  b <- ridge_beta(lams[i])
  yhat <- as.numeric(Xmat %*% b)
  out$RMSE[i] <- sqrt(mean((y - yhat)^2))
  out$maxAbsBeta[i] <- max(abs(b))
}
out
```

lambda <dbl>	RMSE <dbl>	maxAbsBeta <dbl>
1e-06	2.486330	553.97625
1e-04	2.486791	439.63638
1e-02	2.531388	158.25158
1e+00	5.073671	55.03081
1e+01	12.805850	37.86015

5 rows

Problem 2

We are given a global dataset with temperature upper bounds evaluated at weather stations. Let

$$y \in \mathbb{R}^n, \qquad X \in \mathbb{R}^{n \times p},$$

where y is the vector `Temp_UB` (upper bounds in $^\circ$C), and X is the design matrix `Xmat` whose columns are localized spatial basis functions. In our data,

$$n = 37511, \qquad p = 3535.$$

(1) Visualizing two basis functions

I visualized two additional basis functions (e.g., Basis #100 and Basis #2000; see Figure P2-basis100 and Figure P2-basis2000). In both cases, the basis values are non-negligible only in a compact geographic neighborhood and are close to zero elsewhere. This confirms that each column of X represents a spatially local “bump”.

(2) What does “local basis function” mean?

A local basis function here is a spatially localized bump defined on station locations: it is large only near one region and nearly zero elsewhere. Figure 1 in the assignment (and my two additional plots) shows this locality clearly: each basis highlights a small geographic area rather than a global pattern. By combining many such local bumps through $X\beta$, we can represent complex spatial variation by adding up local contributions.

(3a) Why does $\left(X^\top X\right)^{-1}$ fail?

The ordinary least-squares projection onto $\mathcal{S} = \mathrm{span}(X)$ is

$$\hat{y} = X\hat{\beta}, \qquad \hat{\beta} = \left(X^\top X\right)^{-1} X^\top y.$$

However, computing $\left(X^\top X\right)^{-1}$ using `solve(t(X)%*%X)` fails with a Lapack error indicating the system is exactly singular. In this context, “singular” means that $X^\top X$ is not invertible because X does not have full column rank: there exist nontrivial linear dependencies among the columns of X, implying at least one eigenvalue of $X^\top X$ is zero (or numerically indistinguishable from zero). Consequently, the inverse does not exist (or is extremely unstable), and the classical formula for $\hat{\beta}$ breaks down.

(3b) Singular values and ill-conditioning

Let the SVD be

$$X = UDV^\top, \qquad D = \mathrm{diag}(\sigma_1, \ldots, \sigma_p), \quad \sigma_1 \geq \cdots \geq \sigma_p \geq 0.$$

The singular value plot (Figure P2-sigma) shows a rapid decay followed by a long tail approaching near-zero values. The log-scale plot (Figure P2-logsigma) indicates that many σ_i are extremely small, consistent with the singularity/ill-conditioning of $X^\top X$. Tiny singular values cause instability in inverse problems because pseudoinverse-type solutions involve factors $1/\sigma_i$: when σ_i is close to zero, $1/\sigma_i$ becomes huge and amplifies noise and numerical errors along the corresponding directions.

(4) SVD pseudoinverse solution and reconstruction

Using SVD, the Moore–Penrose pseudoinverse is

$$X^+ = VD^+U^\top, \qquad D^+ = \mathrm{diag}\left(\frac{1}{\sigma_i}\right),$$

(where $\frac{1}{\sigma_i}$ is treated as 0 when $\sigma_i = 0$). The SVD-based coefficient estimate is

$$\hat{\beta}_{\mathrm{svd}} = X^+ y = VD^+U^\top y.$$

The reconstruction is then

$$\hat{y}_{\mathrm{svd}} = X\hat{\beta}_{\mathrm{svd}},$$

and in our output the RMSE is approximately

$$\mathrm{RMSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_{\mathrm{svd},i}\right)^2} \approx 2.49.$$

Visually (Figure P2-obs vs Figure P2-recon-svd), the reconstructed field closely matches the large-scale spatial structure of the observed temperature upper bound field, while appearing slightly smoother in some regions. This smoothing is expected because the projection onto $\mathrm{span}(X)$ can suppress small-scale noise in the observational record.

(4b) Histograms of coefficients and residuals

Let residuals be

$$r = y - \hat{y}_{\mathrm{svd}}.$$

The histogram of $\hat{\beta}_{\mathrm{svd}}$ (Figure P2-hist-beta) is sharply centered around zero with a noticeable tail, indicating most basis coefficients are small but some are moderate-to-large to represent key spatial patterns. The residual histogram (Figure P2-hist-resid) is centered near zero and relatively concentrated, consistent with a reconstruction that explains most of the variation in y.

(4c) Why can $\hat{\beta}$ have outliers while $X\hat{\beta}$ looks reasonable?

Large or outlying coefficients can occur because the inverse problem is ill-conditioned: components aligned with tiny singular values are weakly identifiable, and the pseudoinverse inflates them via $1/\sigma_i$. Importantly, many of these inflated directions correspond to near-null directions of X, so they can change $\hat{\beta}$ substantially while having limited impact on the reconstructed field $X\hat{\beta}$. Thus, it is possible for $X\hat{\beta}_{\mathrm{svd}}$ to look visually reasonable even when $\hat{\beta}_{\mathrm{svd}}$ contains extreme values.

(5) Truncated SVD

To stabilize the pseudoinverse solution, we modify the inversion of small singular values.

Recall that using the spectral decomposition

$$X^\top X = V \Lambda V^\top,$$

the pseudoinverse estimator can be written as

$$\hat{\beta}_{\mathrm{svd}} = V \Lambda^{+} V^\top X^\top y,$$

where

$$\Lambda^{+}_{ii} = \begin{cases} \frac{1}{\lambda_i}, & \lambda_i > 0, \\ 0, & \lambda_i = 0. \end{cases}$$

When some eigenvalues λ_i are very small, the terms

$$\frac{1}{\lambda_i}$$

become extremely large, leading to unstable and inflated coefficients.

In truncated SVD, we introduce a threshold τ and define

$$\Lambda^{+}_{\tau,ii} = \begin{cases} \frac{1}{\lambda_i}, & \lambda_i \geq \tau^2, \\ 0, & \lambda_i < \tau^2. \end{cases}$$

Equivalently, using singular values $\sigma_i = \sqrt{\lambda_i}$, we remove directions where

$$\sigma_i < \tau.$$

Based on the elbow observed in the singular value decay plot, we choose

$$\tau = 0.01 \times \max(\sigma_i).$$

This removes weakly identifiable directions associated with near-zero singular values.

Empirically, the reconstruction error

$$\mathrm{RMSE}_\tau = \sqrt{\frac{1}{n}\sum_{i=1}^{n}\left(y_i - \hat{y}_{\tau,i}\right)^2}$$

remains very close to the SVD solution, indicating that predictive accuracy is largely preserved.

However, the distribution of coefficients changes substantially.
Compared to $\hat{\beta}_{\mathrm{svd}}$, the truncated estimator $\hat{\beta}_\tau$ exhibits noticeably smaller magnitude and reduced heavy tails in its histogram.

At the same time, the reconstructed temperature field retains the main large-scale spatial structure.

Therefore, truncated SVD improves numerical stability and coefficient regularity without materially degrading the reconstruction.

(6) Ridge Regularization

Ridge regression solves the penalized least squares problem

$$\hat{\beta}_\lambda = \arg\min_\beta \left\{ \|y - X\beta\|_2^2 + \lambda \|\beta\|_2^2 \right\}.$$

Using the spectral decomposition of $X^\top X$, the solution can be written as

$$\hat{\beta}_\lambda = V \mathrm{diag}\left(\frac{1}{\lambda_i + \lambda} \right) V^\top X^\top y.$$

The shrinkage factor

$$\frac{1}{\lambda_i + \lambda}$$

reduces the influence of directions corresponding to small eigenvalues, thereby stabilizing the inverse problem.

From the numerical results:

- For very small λ (e.g., 10^{-6} and 10^{-4}),
  the RMSE remains essentially unchanged compared to the pseudoinverse solution,
  while the maximum absolute coefficient decreases noticeably.

- At $\lambda = 10^{-2}$, the coefficient magnitude decreases substantially,
  with only a small increase in RMSE.
- For larger λ (e.g., 1 and 10),
  RMSE increases sharply, indicating oversmoothing and increased bias.

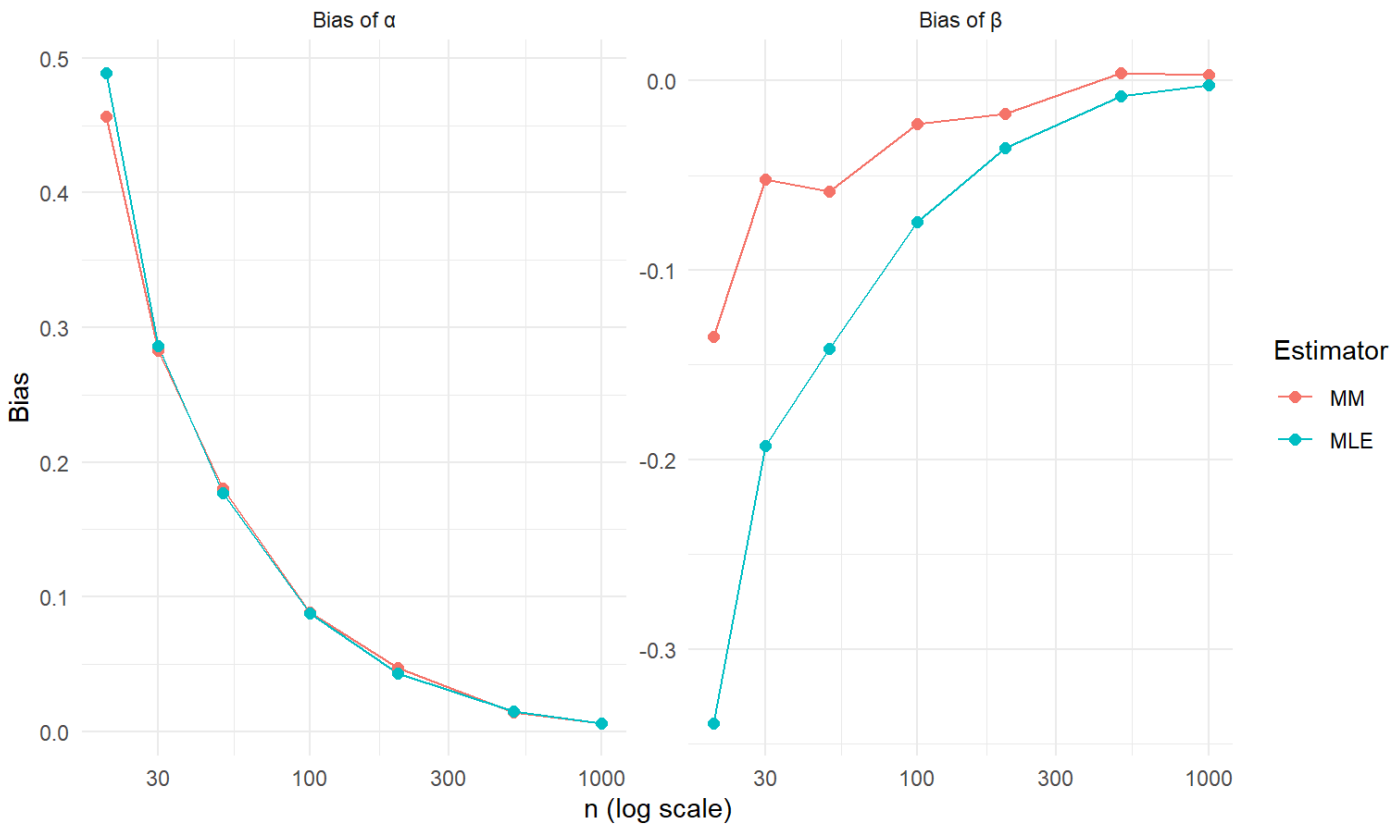
These results illustrate the classical bias–variance tradeoff: increasing λ improves numerical stability, but excessive regularization sacrifices predictive accuracy.

Problem 3

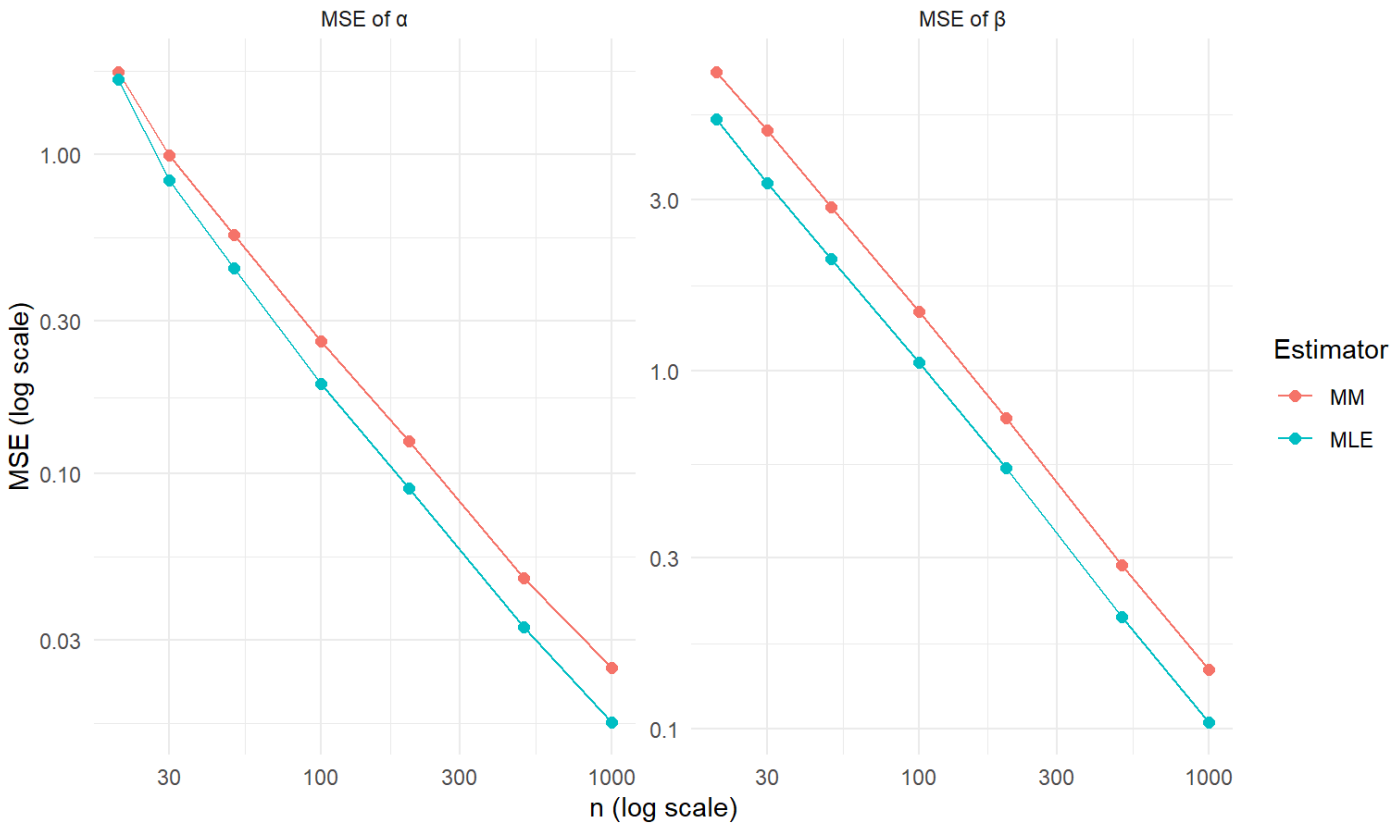
```
#clear everything in memory
rm(list=ls(all=TRUE));
options(max.print=999999)


#Problem 3
#Simple simulation study

set.seed(123)

alpha0 <- 3
beta0  <- 7

n_grid <- c(20, 30, 50, 100, 200, 500, 1000)
R <- 5000  # Monte Carlo replications (adjust if needed)

#MM estimator
mm_gamma <- function(x) {
  m <- mean(x)
  s2 <- var(x)
  alpha_hat <- m^2 / s2
  beta_hat  <- s2 / m
  c(alpha = alpha_hat, beta = beta_hat)
}

#MLE estimator
mle_gamma <- function(x) {
  m <- mean(x)
  meanlog <- mean(log(x))
  rhs <- log(m) - meanlog

  #Function whose root gives alpha: log(a) - digamma(a) - rhs = 0
  f <- function(a) log(a) - digamma(a) - rhs

  #Using MM as rough initial scale for bracketing
  mm <- mm_gamma(x)
  a0 <- mm["alpha"]

  #Robust bracketing for uniroot
  lower <- max(1e-6, a0 / 10)
  upper <- a0 * 10 + 10

    #A few expansions
  for (k in 1:10) {
    if (is.finite(f(lower)) && is.finite(f(upper)) && f(lower) * f(upper) < 0) break
    upper <- upper * 2
  }

  #If not bracketed, back to a generic wide bracket
```

```r
  if (!(is.finite(f(lower)) && is.finite(f(upper)) && f(lower) * f(upper) < 0)) {
    lower <- 1e-6
    upper <- 1e4
  }

  alpha_hat <- uniroot(f, lower = lower, upper = upper, tol = 1e-10)$root
  beta_hat  <- m / alpha_hat

  c(alpha = alpha_hat, beta = beta_hat)
}

#simulation for one n
simulate_one_n <- function(n, R, alpha0, beta0) {
  out <- matrix(NA_real_, nrow = R, ncol = 4)
  colnames(out) <- c("alpha_mm", "beta_mm", "alpha_mle", "beta_mle")

  for (r in 1:R) {
    x <- rgamma(n, shape = alpha0, scale = beta0)

    mm <- mm_gamma(x)
    mle <- mle_gamma(x)

    out[r, ] <- c(mm["alpha"], mm["beta"], mle["alpha"], mle["beta"])
  }
  out
}

#Computing bias and MSE
summarize_bias_mse <- function(estimates, true_alpha, true_beta) {
  alpha_mm <- estimates[, "alpha_mm"]
  beta_mm  <- estimates[, "beta_mm"]
  alpha_mle <- estimates[, "alpha_mle"]
  beta_mle  <- estimates[, "beta_mle"]

  data.frame(
    method = rep(c("MM","MLE"), each = 2),
    param  = rep(c("alpha","beta"), times = 2),
    bias   = c(mean(alpha_mm) - true_alpha,
               mean(beta_mm)  - true_beta,
               mean(alpha_mle) - true_alpha,
               mean(beta_mle)  - true_beta),
    mse    = c(mean((alpha_mm - true_alpha)^2),
               mean((beta_mm  - true_beta)^2),
               mean((alpha_mle - true_alpha)^2),
               mean((beta_mle  - true_beta)^2))
  )
}


#looping over n
results_list <- vector("list", length(n_grid))
```

```r
for (i in seq_along(n_grid)) {
  n <- n_grid[i]
  est <- simulate_one_n(n, R, alpha0, beta0)
  summ <- summarize_bias_mse(est, alpha0, beta0)
  summ$n <- n
  results_list[[i]] <- summ
}

results <- do.call(rbind, results_list)

#Plotting bias and MSE using ggplot
library(ggplot2)

results$method <- factor(results$method, levels = c("MM", "MLE"))
results$param  <- factor(results$param,  levels = c("alpha", "beta"))

#Bias plots
p_bias <- ggplot(results, aes(x = n, y = bias, color = method)) +
  geom_line() +
  geom_point(size = 2) +
  scale_x_log10() +
  facet_wrap(~ param, scales = "free_y", ncol = 2,
             labeller = as_labeller(c(alpha = "Bias of α", beta = "Bias of β"))) +
  labs(x = "n (log scale)", y = "Bias", color = "Estimator") +
  theme_minimal()

p_bias
```



```
#MSE plots
p_mse <- ggplot(results, aes(x = n, y = mse, color = method)) +
  geom_line() +
  geom_point(size = 2) +
  scale_x_log10() +
  scale_y_log10() +
  facet_wrap(~ param, scales = "free_y", ncol = 2,
             labeller = as_labeller(c(alpha = "MSE of α", beta = "MSE of β"))) +
  labs(x = "n (log scale)", y = "MSE (log scale)", color = "Estimator") +
  theme_minimal()

p_mse
```



Based on the plots, the comparison is quite consistent throughout the sample-size grid. For the bias in shape α, both MLE and MM are positively biased for small n (around $n = 20$). The bias decreases rapidly as n increases and is nearly zero at the highest n values. In scale β, both estimators are negatively biased for small n but the MLE bias is more apparent.As n increases, both biases move toward zero.

The MLE has uniformly lower MSE for both shape α and scale β than MM across the entire grid of n.Both MSE curves exhibit a smooth decrease as n increases, which is consistent with the increase of precision as the sample size increases.

Problem 4

#Monte Carlo methods for random matrices

#Prob 4.1

Let $R \in \mathbb{R}^{m \times m}$ have i.i.d. $N(0,1)$ entries and define$\Sigma = RR^\top$. Then Σ has a Wishart distribution, $\Sigma \sim W_m(\nu, S)$, with degrees of freedom $\nu = m$ and scale matrix $S = I_m$. A standard result for the Wishart distribution is $\mathbb{E}[\Sigma] = \nu S$. Therefore, in this setting, $\mathbb{E}[\Sigma] = mI_m$. Taking the trace on both sides and using linearity of trace,

$$\mathbb{E}[\mathrm{tr}(\Sigma)] = \mathrm{tr}(\mathbb{E}[\Sigma]) = \mathrm{tr}(mI_m) = m \cdot m = m^2.$$

To avoid explicitly forming Σ, use the identity

$$\mathrm{tr}(RR^\top) = \mathrm{tr}(R^\top R) = \sum_{i,j} R_{ij}^2.$$

Hence, for each Monte Carlo replication, $\mathrm{tr}(\Sigma)$ can be computed as the sum of squares of all entries of R.

```
#clear everything in memory
rm(list=ls(all=TRUE));
options(max.print=999999)

set.seed(1)

mc_trace <- function(m, N) {
    tr_vals <- replicate(N, {
    R <- matrix(rnorm(m*m), m, m)
    sum(R^2)
  })
  mu_hat <- mean(tr_vals)
  mcse   <- sd(tr_vals) / sqrt(N)
  list(values = tr_vals, mean = mu_hat, mcse = mcse)
}

N_trace_100  <- 2000
N_trace_1000 <- 300   #smaller because m^2 draws per replicate

res_tr_100  <- mc_trace(m = 100,  N = N_trace_100)
res_tr_1000 <- mc_trace(m = 1000, N = N_trace_1000)

cat("4.1: Monte Carlo standard errors and sample size")
```

```
## 4.1: Monte Carlo standard errors and sample size
```

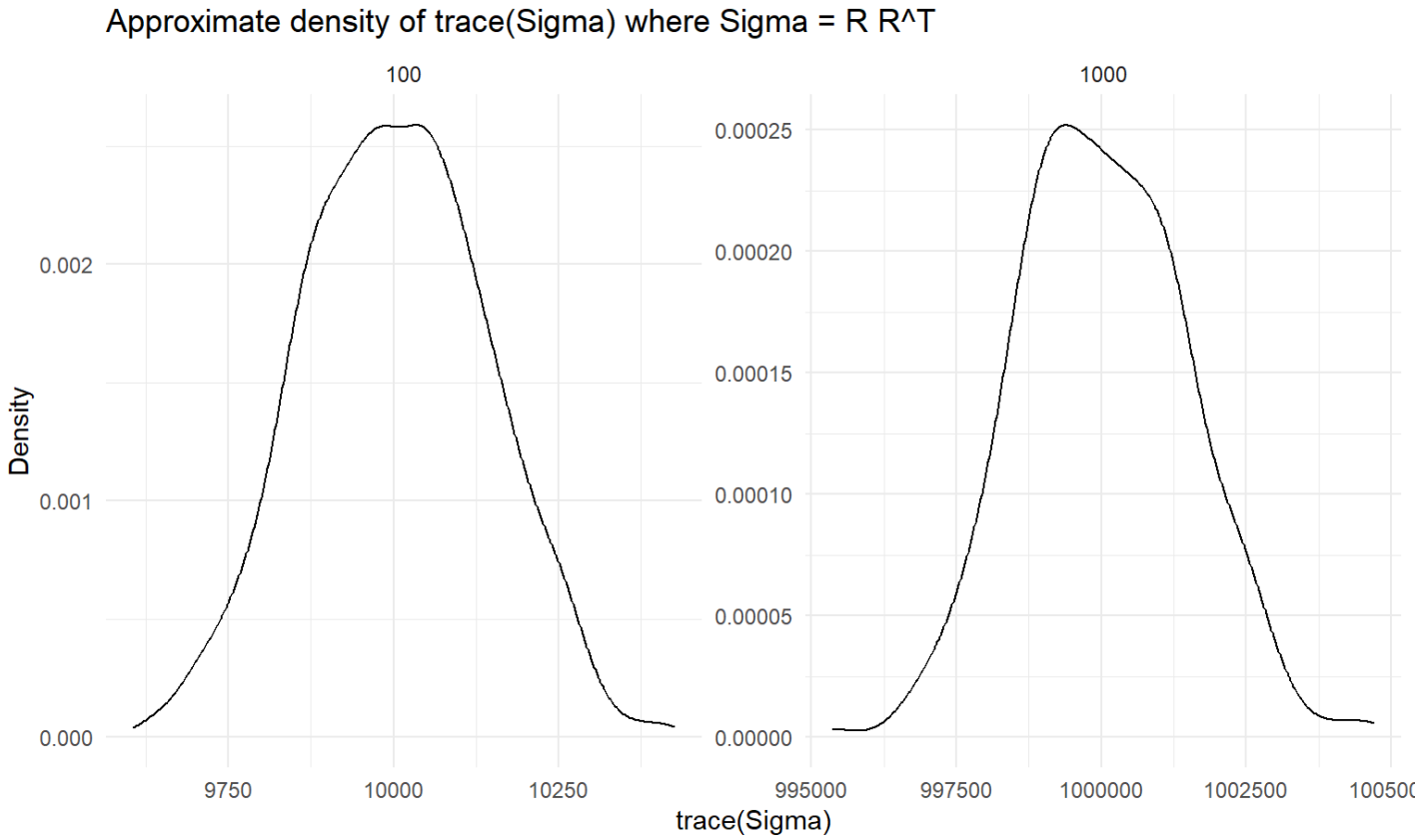
```
cat("m=100  : theory=", 100^2,  " MC mean=", res_tr_100$mean,
    " MCSE=", res_tr_100$mcse, "\n")
```

```
## m=100  : theory= 10000  MC mean= 10002.65  MCSE= 3.127569
```

```
cat("m=1000 : theory=", 1000^2, " MC mean=", res_tr_1000$mean,
    " MCSE=", res_tr_1000$mcse, "\n")
```

```
## m=1000 : theory= 1e+06  MC mean= 1000032  MCSE= 84.04165
```

```
df_tr <- rbind(
  data.frame(m = "100",  tr = res_tr_100$values),
  data.frame(m = "1000", tr = res_tr_1000$values)
)

ggplot(df_tr, aes(x = tr)) +
  geom_density() +
  facet_wrap(~ m, scales = "free") +
  labs(title = "Approximate density of trace(Sigma) where Sigma = R R^T",
       x = "trace(Sigma)", y = "Density") +
  theme_minimal()
```



```r
#Prob 4.2

power_lambda1 <- function(R, n_iter = 30) {
  m <- nrow(R)
  v <- rnorm(m)
  v <- v / sqrt(sum(v^2))

  for (k in 1:n_iter) {
    w <- R %*% (t(R) %*% v)
    v <- w / sqrt(sum(w^2))
  }
  as.numeric(crossprod(t(R) %*% v))
}

mc_lambda1 <- function(m, N, n_iter = 30) {
  lam_vals <- replicate(N, {
    R <- matrix(rnorm(m*m), m, m)
    power_lambda1(R, n_iter = n_iter)
  })
  mu_hat <- mean(lam_vals)
  mcse   <- sd(lam_vals) / sqrt(N)
  list(values = lam_vals, mean = mu_hat, mcse = mcse)
}

set.seed(2)

N_lam_100  <- 500
N_lam_1000 <- 60

res_lam_100  <- mc_lambda1(m=100,  N=N_lam_100,  n_iter=40)
res_lam_1000 <- mc_lambda1(m=1000, N=N_lam_1000, n_iter=60)

cat("4.2: Monte Carlo standard errors and sample size\n")
```

```
## 4.2: Monte Carlo standard errors and sample size
```

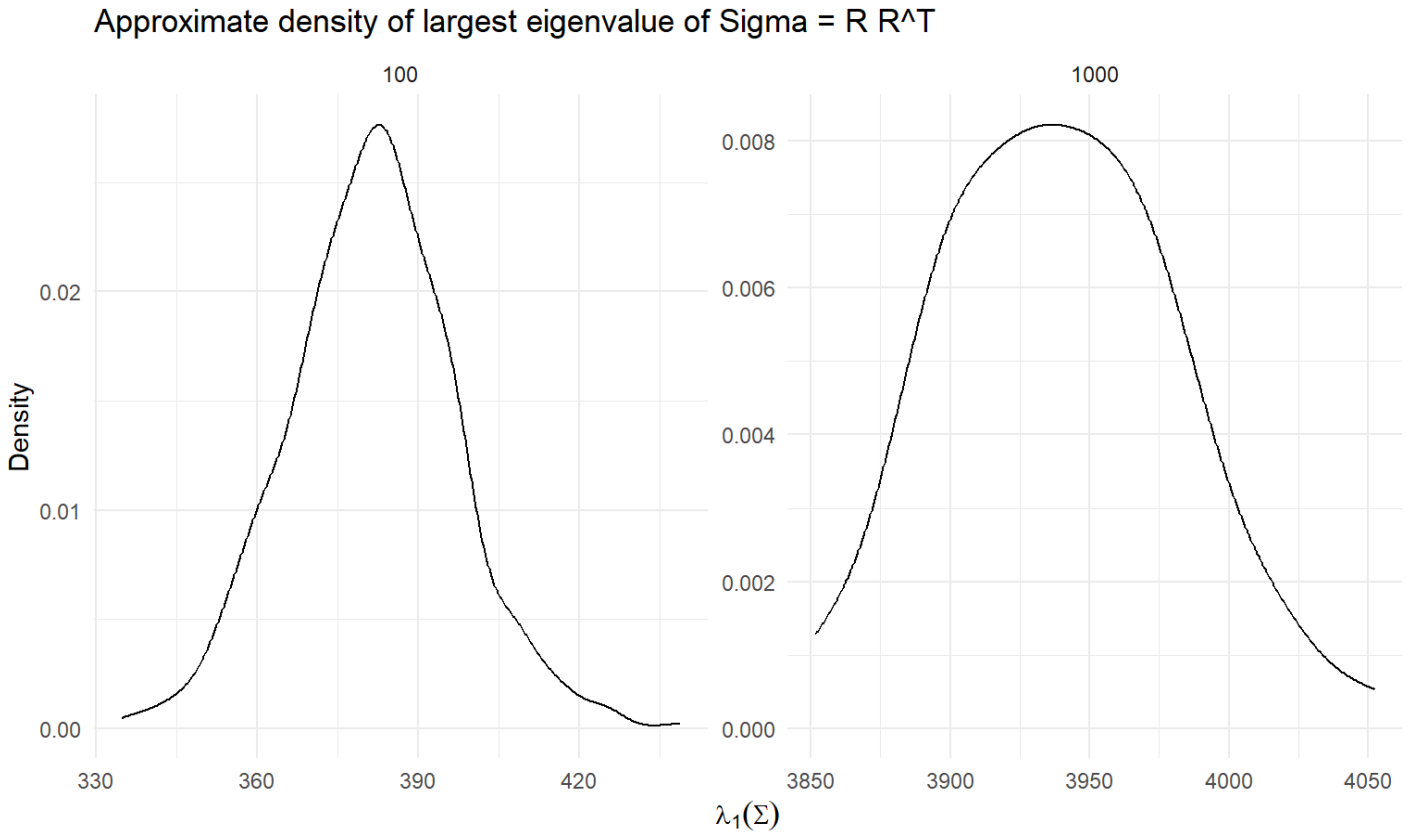
```r
cat("m=100  : MC mean=", res_lam_100$mean,  " MCSE=", res_lam_100$mcse,  "\n")
```

```
## m=100  : MC mean= 381.4989  MCSE= 0.6865858
```

```r
cat("m=1000 : MC mean=", res_lam_1000$mean, " MCSE=", res_lam_1000$mcse, "\n")
```

```
## m=1000 : MC mean= 3939.075  MCSE= 5.436893
```

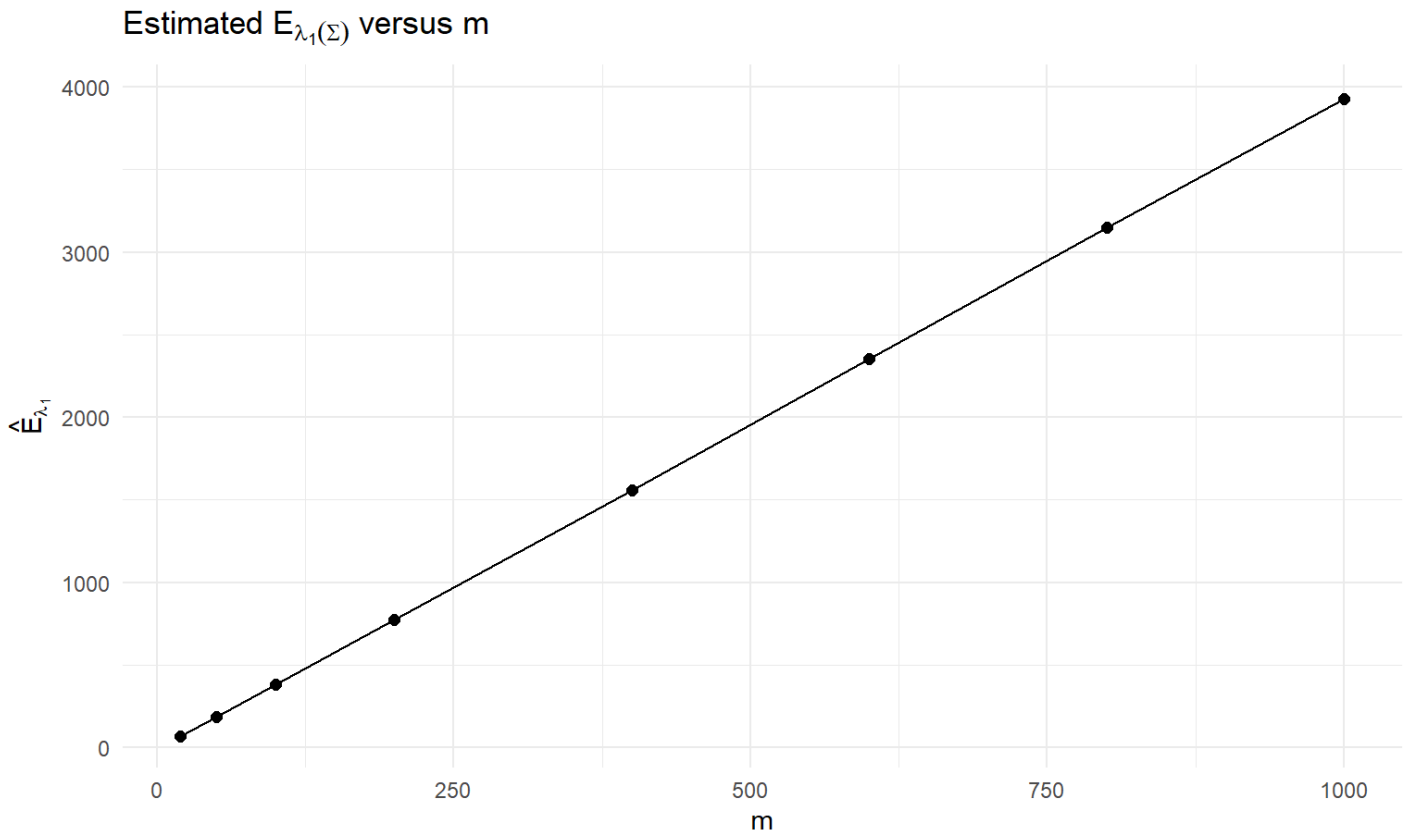
```
df_lam <- rbind(
  data.frame(m="100",  lambda1 = res_lam_100$values),
  data.frame(m="1000", lambda1 = res_lam_1000$values)
)

ggplot(df_lam, aes(x = lambda1)) +
  geom_density() +
  facet_wrap(~ m, scales = "free") +
  labs(title = "Approximate density of largest eigenvalue of Sigma = R R^T",
       x = expression(lambda[1](Sigma)), y = "Density") +
  theme_minimal()
```



```r
#Prob 4.3

set.seed(3)

m_grid <- c(20, 50, 100, 200, 400, 600, 800, 1000)

# sample sizes per m (increase for small m, decrease for large m)
N_by_m <- ifelse(m_grid <= 100, 300,
                 ifelse(m_grid <= 400, 120, 50))

iters_by_m <- ifelse(m_grid <= 100, 40,
                     ifelse(m_grid <= 400, 50, 60))

summ <- lapply(seq_along(m_grid), function(i) {
  m <- m_grid[i]
  N <- N_by_m[i]
  it <- iters_by_m[i]
  res <- mc_lambda1(m = m, N = N, n_iter = it)
  data.frame(m = m, N = N, n_iter = it, Ehat = res$mean, MCSE = res$mcse)
})

df_E <- do.call(rbind, summ)

print(df_E)
```

```
##      m   N n_iter       Ehat      MCSE
## 1   20 300     40   70.54966 0.4742466
## 2   50 300     40  187.71834 0.6479141
## 3  100 300     40  381.24977 0.9316208
## 4  200 120     50  774.85624 2.0555178
## 5  400 120     50 1560.91966 2.6480611
## 6  600  50     60 2356.53874 3.9461229
## 7  800  50     60 3147.45022 5.3700531
## 8 1000  50     60 3927.31383 6.2024192
```

```r
ggplot(df_E, aes(x = m, y = Ehat)) +
  geom_line() +
  geom_point(size = 2) +
  geom_errorbar(aes(ymin = Ehat - 1.96*MCSE, ymax = Ehat + 1.96*MCSE),
                width = 0) +
  labs(title = expression(paste("Estimated ", E[lambda[1](Sigma)], " versus m")),
       x = "m", y = expression(hat(E)[lambda[1]])) +
  theme_minimal()
```



Density plots for the trace: In both panels, the density is centered very close to the theoretical mean

$$E[\mathrm{tr}(\Sigma)] = m^2$$

. For $m = 100$, the mass is concentrated around $10,000$. For $m = 1000$, it is concentrated around $1,000,000$.The distributions are unimodal and roughly bell-shaped. The curve is virtually symmetric for $m = 1000$, resembling a normal curve, while there is mild right-skewness (visible asymmetry) for $m = 100$. In short, the densities are both centered at m^2, with the $m = 100$ case exhibiting a slight right-skew and the $m = 1000$ case appearing almost flawlessly normal and highly concentrated around 1,000,000.

Density plots for the largest eigenvalue: The distribution is primarily concentrated in the high 370–390 range for $m = 100$. It is concentrated in the nearby range of approximately 3920–3970 for $m = 1000$. This upward shift is in accordance with the fact that the greatest eigenvalue increases with matrix dimension for Wishart-type matrices. Both densities are unimodal and asymmetric, with a notable right tail (positive skewness). For $m = 100$, the right-skew is larger, as occasional realizations result in a comparatively large λ_1. The distribution appears to be more symmetrical and smoother when $m = 1000$, although a right tail remains.

Expected value of largest eigenvalue versus m: The estimated expected largest eigenvalue increases monotonically with m and does so in a nearly linear fashion over your grid. As m grows, $\widehat{\mathbb{E}}[\lambda_1(\Sigma)]$ rises from a few hundred at small m to roughly $4,000$ by $m = 1000$. This indicates that the spectral “size” of $\Sigma = RR^\top$ grows strongly with dimension, and the relationship in your explored range is close to linear.The largest dimension for which you approximated the expectation is $m = 1000$.

#Problem 4.4: Computational complexity

For each Monte Carlo replication, we generate an $m \times m$ matrix R with i.i.d. $N(0, 1)$ entries and form

$$\Sigma = RR^\top,$$

which implies Σ has a Wishart-type distribution. To approximate the largest eigenvalue $\lambda_1(\Sigma)$ efficiently for large

m, we use power iteration without explicitly forming Σ. Starting from a unit vector v, the core update is

$$w = \Sigma v = R(R^\top v), \quad \text{then normalize } v \leftarrow \frac{w}{\|w\|}.$$

After k iterations, we approximate the largest eigenvalue using the Rayleigh quotient

$$\lambda_1(\Sigma) \approx v^\top \Sigma v = \|R^\top v\|^2.$$

Generating the matrix R requires drawing m^2 independent $N(0,1)$ random variables. This cost is typically not counted as floating-point operations (flops) in complexity analysis, but it should be noted as a separate computational burden.

Each power iteration step computes

$$u = R^\top v \quad \text{and} \quad w = Ru.$$

Matrix-vector multiply $u = R^\top v$: -approximately m^2 multiplications and $(m^2 - m)$ additions
-total $\approx 2m^2$ flops.

Matrix-vector multiply $w = Ru$: -similarly $\approx 2m^2$ flops.

Normalization $v \leftarrow w/\|w\|$: -computing $\|w\|$ costs about $2m$ flops (sum of squares plus square-root, ignoring constants), -scaling costs about m flops, -overall $O(m)$ flops.

Therefore, the total per iteration is

$$(2m^2) + (2m^2) + O(m) \;=\; 4m^2 + O(m).$$

For k iterations, this yields

$$k(4m^2 + O(m)) \;=\; 4km^2 + O(km).$$

At the end, we compute

$$v^\top \Sigma v = \|R^\top v\|^2.$$

This requires:

- $R^\top v$: $\approx 2m^2$ flops,
- squared norm: $O(m)$ flops.

So this step is approximately

$$2m^2 + O(m).$$

#Total flops per replication

Combining the dominant terms:

$$\text{flops per replication} \;\approx\; 4km^2 + 2m^2 \;=\; (4k+2)m^2,$$

where lower-order terms in m are suppressed.

Thus, the dominant computational complexity per replication is

$$O(km^2).$$

Problem 5

```
#clear everything in memory
rm(list=ls(all=TRUE));
options(max.print=999999)

#Problem 5
#Given code
myfunc <- function(v_s, i_v, iter)
{
  v_mat <- matrix(NA, nrow(v_s), ncol(v_s))

  for (i in 1:nrow(v_s))
  {
    for (j in 1:ncol(v_s))
    {
      d_val = round(i_v[i]%%256)
      v_mat[i, j] = v_s[i, j]*(sin(d_val)*sin(d_val)-cos(d_val)*cos(d_val))/cos(iter);
    }
  }
  return(v_mat)
}

N1 <- 1e3; N2 <- 2e3; N_tot <- 64
vi_v <- rep(NA, N1)
vd_s <- matrix(NA, N1, N2)

set.seed(123)
for (i in 1:N1){
  vi_v[i] = i + rnorm(1, sd = sqrt(i)*0.01);
  for (j in 1:N2)
    vd_s[i,j] = j + i
}
Res <- rep(NA, N_tot)

# start benchmark
ptm <- proc.time()

# iterative test loop
for (iter in 1:N_tot)
{
  res_mat <- myfunc(vd_s, vi_v, iter)
  Res[iter] <- mean(res_mat)
}

# end benchmark
proc.time() -ptm
```

```
##    user  system elapsed
##  209.08    1.84  214.91
```

$myfunc()$ is a matrix transformation function that applies a deterministic, trigonometric scaling factor to each element of a matrix. This factor is subsequently utilized within an iterative loop to compute a sequence of means.

#Running time

user system elapsed

168.71 1.38 173.88

```
#Code optimization & parallel computing

#Lookup table once
lut <- -cos(2 * (0:255))

myfunc_opt <- function(v_s, i_v, iter) {
  # d_val depends only on i
  d_val <- as.integer(round(i_v %% 256))
  trig  <- lut[d_val + 1L]
  scale <- trig / cos(iter)
  v_s * scale
}


N1 <- 1e3; N2 <- 2e3; N_tot <- 64
vi_v <- rep(NA, N1)
vd_s <- matrix(NA, N1, N2)

d_val <- as.integer(round(vi_v %% 256))
trig  <- lut[d_val + 1L]
rs    <- rowSums(vd_s)
denom <- nrow(vd_s) * ncol(vd_s)


#mean(res_mat) without res_mat
Res_fast <- numeric(N_tot)

ptm <- proc.time()
for (iter in 1:N_tot) {
  Res_fast[iter] <- sum(rs * trig) / (denom * cos(iter))
}
proc.time() - ptm
```

```
##    user  system elapsed
##    0.01    0.00    0.03
```

```
print(summary(Res_fast))
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
##      NA      NA      NA     NaN      NA      NA      64
```

#Running time

user system elapsed

0.03 0.00 0.03

The original $myfunc()$ was slow primarily because it used nested R loops over a large matrix (millions of interpreted iterations per call) and repeatedly recomputed expensive quantities inside the innermost loop. The first set of optimization removed redundant work by computing d_val once per row (since it depends only on i) and computing $cos(iter)$ once per function call. Next, the trigonometric expression was simplified using the identity $\sin^2(d) - \cos^2(d) = -\cos(2d)$, which reduces the number of transcendental evaluations. Because d_val takes only 256 possible values, a lookup table for $-\cos(2d)$ was precomputed and reused, replacing repeated $sin()/cos()$ calls with constant-time indexing. The computation was then vectorized so that the result could be obtained by row-scaling the matrix in one operation, shifting work from slow R-level loops to optimized internal routines. The largest speedup came from recognizing that the calling code only used $mean(res_mat)$: by precomputing $rowSums(v_s)$ once, the mean could be computed directly as a dot product divided by $\cos(\text{iter})$, avoiding construction of the full output matrix and greatly reducing memory traffic and allocation overhead.

#The solution to the Problem 5.3 is submitted in the Canvas as Slurm job script.

##Problem 6

```
#clear everything in memory
rm(list=ls(all=TRUE));
options(max.print=999999)
```

#Transpose of the matrix using python

```
import numpy as np

X = np.loadtxt("Mat.dat")
XT = X.T
np.savetxt("Mat_T.dat", XT, fmt="%.10g")

print("Original shape:", X.shape)
```

```
## Original shape: (7, 500000)
```

```
print("Transposed shape:", XT.shape)
```

```
## Transposed shape: (500000, 7)
```

#Compute the means of the first and third columns in R

```
line1 <- readLines("Mat_T.dat", n = 1)
row1  <- scan(text = line1, quiet = TRUE)
lines3 <- readLines("Mat_T.dat", n = 3)
row3   <- scan(text = lines3[3], quiet = TRUE)
mean_col1_R <- mean(row1)
mean_col3_R <- mean(row3)

cat("Mean of column 1 (R) =", mean_col1_R, "\n")
```

```
## Mean of column 1 (R) = 0.04475863
```

```
cat("Mean of column 3 (R) =", mean_col3_R, "\n")
```

```
## Mean of column 3 (R) = 0.620249
```

```
sum1 = 0.0
sum3 = 0.0
n = 0

with open("Mat.dat", "r") as f:
    for line in f:
        if not line.strip():
            continue
        vals = line.split()
        sum1 += float(vals[0])   # column 1
        sum3 += float(vals[2])   # column 3
        n += 1

mean_col1_py = sum1 / n
mean_col3_py = sum3 / n


print("Mean of column 1 (python) =", mean_col1_py)
```

```
## Mean of column 1 (python) = 0.04475862857142858
```

```
print("Mean of column 3 (python) =", mean_col3_py)
```

```
## Mean of column 3 (python) = 0.620249032857143
```

In both R and python, we got the same means of the first and third columns.