Problem 1

Let λ be an eignvalue of matrix AA^T, and $V \neq 0$ be the corresponding eigenvector.

$(AA^T)V = \lambda V$ where V is a nonzero vector in R^n

$A^T(AA^T)V = A^T(\lambda V)$

$(A^TA)(A^TV) = \lambda(A^TV)$

if $A^TV \neq 0$, then A^TV is an eigenvector of A^TA corresponding to the eigenvalue λ.

Problem 3

$E_X[X] = E_Y[E_{X|y}[X|Y]]$

$E_X[X] = \int_{-\infty}^{\infty} x \cdot P_X(x)\, dx$

$P_X(x) = \int_{-\infty}^{\infty} P(x, y)\, dy$

$E_X[X] = \int_{-\infty}^{\infty} x \cdot \left(\int_{-\infty}^{\infty} P(x, y)\, dy\right) dx$

$E_{X|Y}[X|Y = y] = \int_{-\infty}^{\infty} x \cdot P(x|y)\, dx$

$P(x|y) = \frac{P(x, y)}{P_Y(y)}$

$\therefore E_X[X] = E_Y[E_{X|Y}[X|Y]]$

$E_{X|Y}[X|Y = y] = \int_{-\infty}^{\infty} x \cdot \frac{P(x, y)}{P_Y(y)}\, dx$

$$E_Y[E_{X|Y}[X|Y]] = \int_{-\infty}^{\infty} P_Y(y) \left(\int_{-\infty}^{\infty} x \cdot \frac{P(x,y)}{P_Y(y)}\, dx\right) dy$$
$$= \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x \cdot P(x, y)\, dx\, dy$$

$$\int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x \cdot P(x, y)\, dx\, dy = \int_{-\infty}^{\infty}\int_{-\infty}^{\infty} x \cdot P(x, y)\, dy\, dx$$
$$= \int_{-\infty}^{\infty} x \left(\int_{-\infty}^{\infty} P(x, y)\, dy\right) dx = \int_{-\infty}^{\infty} x \cdot P_X(x)\, dx = E_X[X]$$

Problem 4

(a) $P(x|k,a) = \frac{\alpha k^\alpha}{x_i^{\alpha+1}} \quad x \in [k, \infty)$

$$L(x_1, \dots x_N; k, \alpha) = \prod_{i=1}^{N} P(x_i | k, \alpha)$$
$$= \prod_{i=1}^{N} \frac{\alpha k^\alpha}{x_i^{\alpha+1}}$$

$$l(x_1, \dots x_N; k, \alpha) = \log L(x_1, \dots x_N; k, \alpha)$$

$$\because \log\left(\prod_{i=1}^{N} f(x_i)\right) = \sum_{i=1}^{N} \log f(x_i)$$

$$\therefore l(x_1, \dots x_N; k, \alpha) = \sum_{i=1}^{N} \log\left(\frac{\alpha k^\alpha}{x_i^{\alpha+1}}\right)$$
$$= \sum_{i=1}^{N} (\log(\alpha) + \alpha \log(k) - (\alpha+1)\log(x_i))$$
$$= N\log(\alpha) + N\alpha\log(k) - (\alpha+1)\sum_{i=1}^{N}\log(x_i)$$

$$l(x_1, \dots x_N; k, \alpha) = N\log(\alpha) + N\alpha\log(k) - (a+1)\sum_{i=1}^{N}\log(x_i)$$

(b) $l(\alpha) = N\log(\alpha) + N\alpha\log(k) - (a+1)\sum_{i=1}^{N}\log(x_i)$

$$\frac{\partial}{\partial\alpha} = \frac{N}{\alpha} + N\log(k) - \sum_{i=1}^{N}\log(x_i)$$

Set equal to 0

$$\frac{N}{\alpha} + N\log(k) - \sum_{i=1}^{N}\log(x_i) = 0$$
$$\frac{N}{\alpha} = \sum_{i=1}^{N}\log(x_i) - N\log(k)$$
$$\alpha = \frac{N}{\sum_{i=1}^{N}\log(x_i) - N\log(k)}$$

$\therefore$ The MLE is $\hat{\alpha} = \frac{N}{\sum_{i=1}^{N}\log(x_i) - N\log(k)}$

Problem 2

Haonan LI

2025-02-06

```
# Problem 2
# (a)
N <- 9

# Construct the matrix C using nested loops
C <- matrix(0, nrow = N, ncol = N)
for (i in 1:N) {
  for (j in 1:N) {
    C[i, j] <- min(i, j) / N
  }
}

# Compute eigenvalues and eigenvectors
eig <- eigen(C)
lambda <- eig$values
V <- eig$vectors

# Display eigenvalues and eigenvectors
cat("Eigenvalues (lambda_1 to lambda_9):\n")
```

```
## Eigenvalues (lambda_1 to lambda_9):
```

```
print(round(lambda, 6))
```

```
## [1] 4.073377 0.460942 0.172149 0.092855 0.060556 0.044606 0.035913 0.031052
## [9] 0.028551
```

```
cat("\nFirst two eigenvectors (v_1 and v_2):\n")
```

```
##
## First two eigenvectors (v_1 and v_2):
```

```
print(round(V[, 1:2], 6))
```

```
##            [,1]      [,2]
##  [1,] -0.075521 -0.218380
##  [2,] -0.148982 -0.384118
##  [3,] -0.218380 -0.457264
##  [4,] -0.281820 -0.420186
##  [5,] -0.337573 -0.281820
##  [6,] -0.384118 -0.075521
##  [7,] -0.420186  0.148982
##  [8,] -0.444791  0.337573
##  [9,] -0.457264  0.444791
```

```
#(b)
# Construct the diagonal matrix Lambda
Lambda <- diag(lambda)

# Reconstruct matrix C using V * Lambda * V^T
C_reconstructed <- V %*% Lambda %*% t(V)

# Verify if the reconstructed matrix is close to the original C
is_close_b <- all.equal(C, C_reconstructed)

# Output the results
cat("\nReconstructed Matrix C:\n")
```

```
##
## Reconstructed Matrix C:
```

```
print(round(C_reconstructed, 6))
```

```
##           [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
##  [1,] 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111
##  [2,] 0.111111 0.222222 0.222222 0.222222 0.222222 0.222222 0.222222 0.222222
##  [3,] 0.111111 0.222222 0.333333 0.333333 0.333333 0.333333 0.333333 0.333333
##  [4,] 0.111111 0.222222 0.333333 0.444444 0.444444 0.444444 0.444444 0.444444
##  [5,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.555556 0.555556 0.555556
##  [6,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.666667 0.666667
##  [7,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.777778
##  [8,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.888889
##  [9,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.888889
##           [,9]
##  [1,] 0.111111
##  [2,] 0.222222
##  [3,] 0.333333
##  [4,] 0.444444
##  [5,] 0.555556
##  [6,] 0.666667
##  [7,] 0.777778
##  [8,] 0.888889
##  [9,] 1.000000
```

```
cat("\nOriginal Matrix C:\n")
```

```
##
## Original Matrix C:
```

```r
print(round(C, 6))
```

```
##           [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
##  [1,] 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111
##  [2,] 0.111111 0.222222 0.222222 0.222222 0.222222 0.222222 0.222222 0.222222
##  [3,] 0.111111 0.222222 0.333333 0.333333 0.333333 0.333333 0.333333 0.333333
##  [4,] 0.111111 0.222222 0.333333 0.444444 0.444444 0.444444 0.444444 0.444444
##  [5,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.555556 0.555556 0.555556
##  [6,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.666667 0.666667
##  [7,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.777778
##  [8,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.888889
##  [9,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.888889
##           [,9]
##  [1,] 0.111111
##  [2,] 0.222222
##  [3,] 0.333333
##  [4,] 0.444444
##  [5,] 0.555556
##  [6,] 0.666667
##  [7,] 0.777778
##  [8,] 0.888889
##  [9,] 1.000000
```

```r
cat("\nDoes C = V * Lambda * V^T (within tolerance)?\n")
```

```
##
## Does C = V * Lambda * V^T (within tolerance)?
```

```r
print(is_close_b)
```

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

```r
# (c)
# Compute the square root of the diagonal matrix Lambda
Lambda_sqrt <- diag(sqrt(lambda))

# Compute A = V * Lambda^(1/2)
A <- V %*% Lambda_sqrt

# Compute AA^T
AA_T <- A %*% t(A)

# Verify if AA^T equals C
is_close_c <- all.equal(C, AA_T)

# Output results
cat("\nMatrix A (first few rows for brevity):\n")
```

```
##
## Matrix A (first few rows for brevity):
```

```r
print(round(A[1:5, 1:5], 6))
```

```
##           [,1]      [,2]      [,3]      [,4]      [,5]
## [1,] -0.152421 -0.148264  0.140062 -0.128039  0.112524
## [2,] -0.300685 -0.260788  0.189723 -0.102866  0.018584
## [3,] -0.440747 -0.310449  0.116929  0.045398 -0.109455
## [4,] -0.568787 -0.285275 -0.031334  0.139338 -0.036662
## [5,] -0.681311 -0.191335 -0.159374  0.066545  0.103400
```

```r
cat("\nMatrix AA^T:\n")
```

```
## 
## Matrix AA^T:
```

```r
print(round(AA_T, 6))
```

```
##           [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
##  [1,] 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111
##  [2,] 0.111111 0.222222 0.222222 0.222222 0.222222 0.222222 0.222222 0.222222
##  [3,] 0.111111 0.222222 0.333333 0.333333 0.333333 0.333333 0.333333 0.333333
##  [4,] 0.111111 0.222222 0.333333 0.444444 0.444444 0.444444 0.444444 0.444444
##  [5,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.555556 0.555556 0.555556
##  [6,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.666667 0.666667
##  [7,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.777778
##  [8,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.888889
##  [9,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.888889
##           [,9]
##  [1,] 0.111111
##  [2,] 0.222222
##  [3,] 0.333333
##  [4,] 0.444444
##  [5,] 0.555556
##  [6,] 0.666667
##  [7,] 0.777778
##  [8,] 0.888889
##  [9,] 1.000000
```

```r
cat("\nOriginal Matrix C:\n")
```

```
## 
## Original Matrix C:
```

```r
print(round(C, 6))
```

```
##           [,1]     [,2]     [,3]     [,4]     [,5]     [,6]     [,7]     [,8]
##  [1,] 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111 0.111111
##  [2,] 0.111111 0.222222 0.222222 0.222222 0.222222 0.222222 0.222222 0.222222
##  [3,] 0.111111 0.222222 0.333333 0.333333 0.333333 0.333333 0.333333 0.333333
##  [4,] 0.111111 0.222222 0.333333 0.444444 0.444444 0.444444 0.444444 0.444444
##  [5,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.555556 0.555556 0.555556
##  [6,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.666667 0.666667
##  [7,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.777778
##  [8,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.888889
##  [9,] 0.111111 0.222222 0.333333 0.444444 0.555556 0.666667 0.777778 0.888889
##           [,9]
##  [1,] 0.111111
##  [2,] 0.222222
##  [3,] 0.333333
##  [4,] 0.444444
##  [5,] 0.555556
##  [6,] 0.666667
##  [7,] 0.777778
##  [8,] 0.888889
##  [9,] 1.000000
```

```
cat("\nDoes AA^T equal C (within tolerance)?\n")
```

```
##
## Does AA^T equal C (within tolerance)?
```

```
print(is_close_c)
```

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

Problem 5

Haonan LI

2025-02-04

```r
# Problem 5
# (a)

train_file <- "C:/R code/stat385a1_train_data.csv"
test_file <- "C:/R code/stat385a1_test_data.csv"

# Load training data
train_data <- read.csv(train_file)
test_data <- read.csv(test_file)
x_train <- as.numeric(train_data$X)
y_train <- as.numeric(train_data$Y)
x_test <- as.numeric(test_data$X_test)
y_test <- as.numeric(test_data$Y_test)

# Building a Design Matrix
design_matrix <- function(x, degree) {
  n <- length(x)
  Phi <- matrix(1, nrow = n, ncol = degree)
  if (degree > 1) {
    for (i in 2:degree) {
      Phi[, i] <- x^(i - 1)
    }
  }
  return(Phi)
}

# Compute the weight vector for the maximum likelihood estimate w
mle_weights <- function(Phi, y, lambda = 1e-3) {
  Phi <- as.matrix(Phi)
  y <- as.numeric(y)
  n <- ncol(Phi)
  w <- solve(t(Phi) %*% Phi + lambda * diag(n)) %*% t(Phi) %*% y
  return(w)
}

# Choose the order of the polynomial n
n <- 3
Phi_train <- design_matrix(x_train, n)

# Calculate the maximum likelihood estimated weights w
w <- mle_weights(Phi_train, y_train)
```

```r
# Calculate the training set prediction based on w
y_pred <- Phi_train %*% w

# output result
cat("Weight vector w:\n")
```

```
## Weight vector w:
```

```r
print(w)
```

```
##             [,1]
## [1,]  6.00705855
## [2,]  0.08126199
## [3,] -1.21579709
```

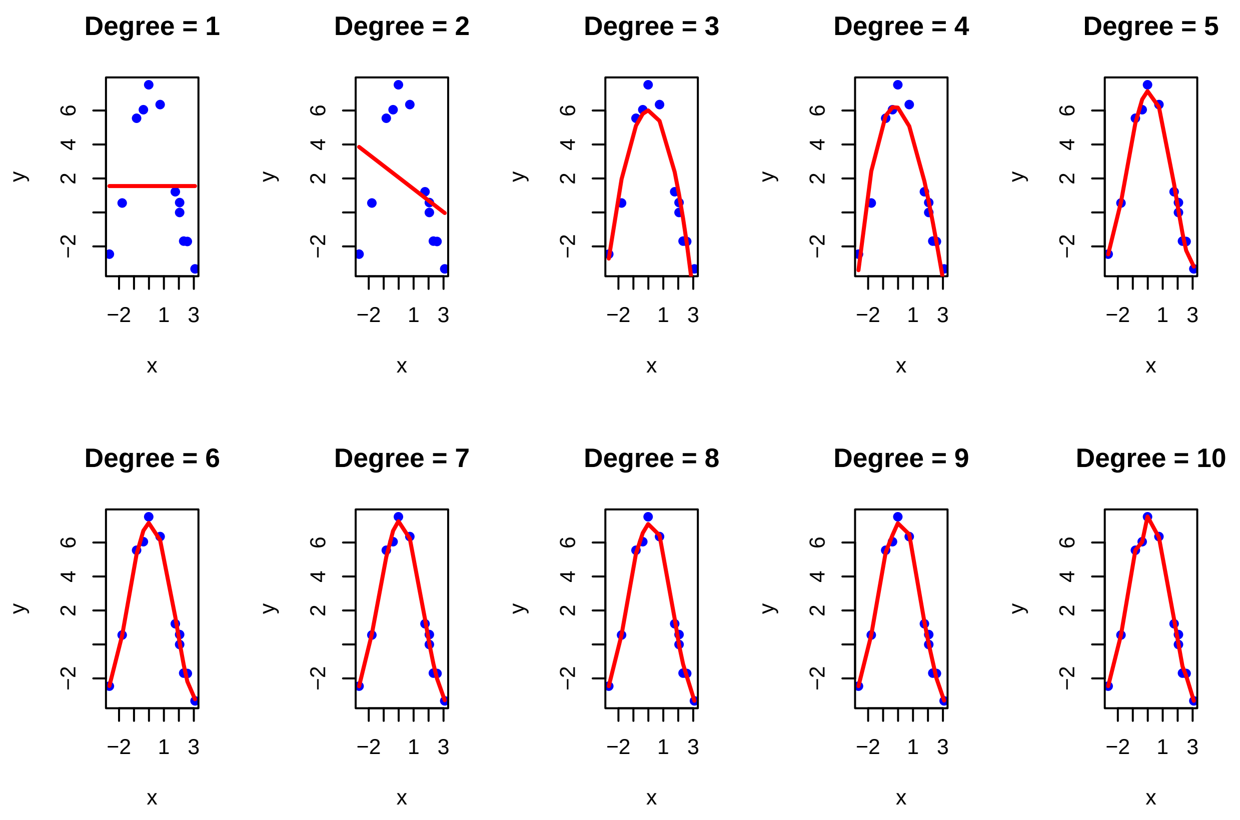
```r
cat("\nPredicted values y_pred:\n")
```

```
##
## Predicted values y_pred:
```

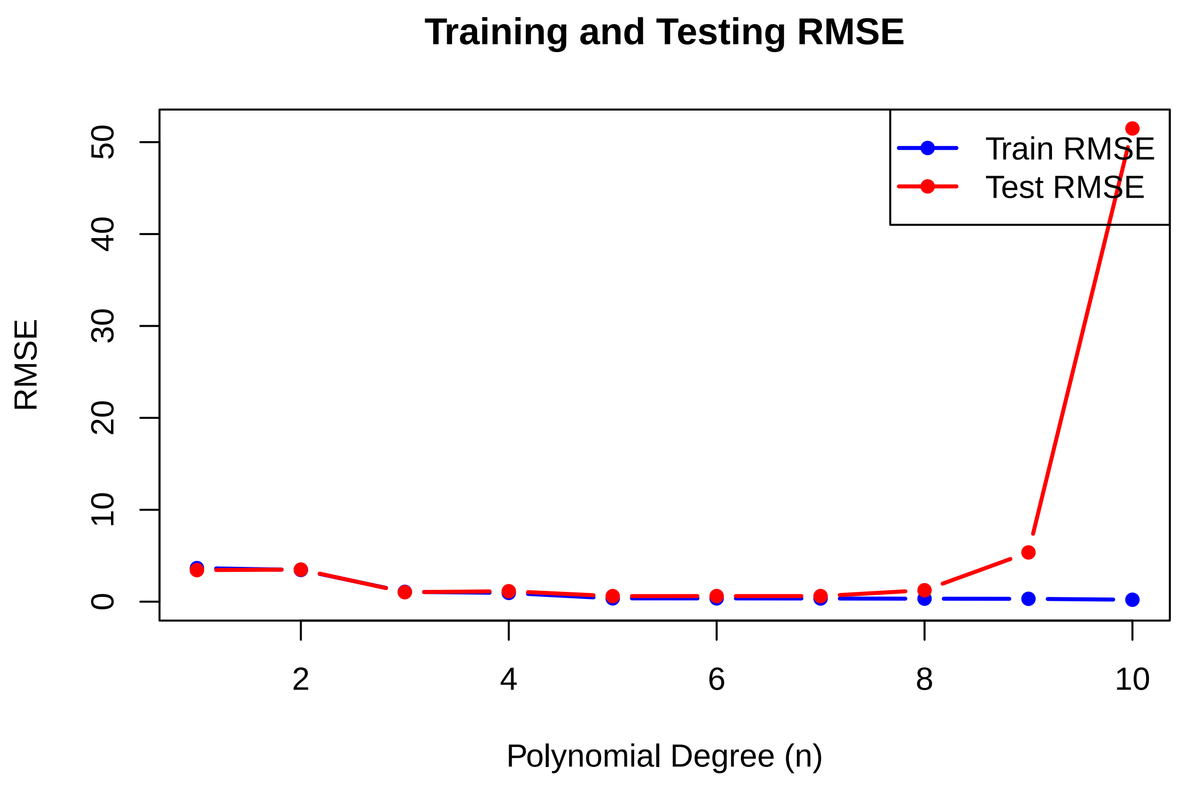
```r
print(y_pred)
```

```
##             [,1]
##  [1,]  5.3888155
##  [2,]  5.8083967
##  [3,]  1.9625804
##  [4,]  1.0531680
##  [5,] -2.7203657
##  [6,] -1.7860835
##  [7,] -0.3586529
##  [8,]  2.3774079
##  [9,]  5.1086981
## [10,] -5.2773697
## [11,]  1.0560299
## [12,]  6.0054703
```

```r
# (b)
# Setting the polynomial order range
degree_range <- 1:10

# Plotting the fitted curve
par(mfrow = c(2, 5))
for (n in degree_range) {
  Phi_train <- design_matrix(x_train, n)
  w <- mle_weights(Phi_train, y_train)
  y_pred <- Phi_train %*% w
  plot(x_train, y_train, col = "blue", pch = 16,
       main = paste("Degree =", n),
       xlab = "x", ylab = "y")
  lines(sort(x_train), y_pred[order(x_train)], col = "red", lwd = 2)
}
```



```r
# Low-Degree Polynomials: Underfitting leads to poor model performance.

# Medium-Degree Polynomials: Achieve the best balance,
# minimizing both training and testing errors.

# High-Degree Polynomials: Overfitting results in
# low training error but high testing error.


# (c)
# Calculate RMSE
rmse <- function(y_true, y_pred) {
  sqrt(mean((y_true - y_pred)^2))
}

# Initialize vectors to store RMSE for training and testing
train_rmse <- numeric(length(degree_range))
test_rmse <- numeric(length(degree_range))

# Loop through each polynomial degree to compute RMSE
for (n in degree_range) {
  # Build design matrices for training and testing data
  Phi_train <- design_matrix(x_train, n)
  Phi_test <- design_matrix(x_test, n)

  # Calculate weights
  w <- mle_weights(Phi_train, y_train)

  # Predict values for training and testing data
  y_pred_train <- Phi_train %*% w
  y_pred_test <- Phi_test %*% w

  # Calculate RMSE for training and testing
  train_rmse[n] <- rmse(y_train, y_pred_train)
  test_rmse[n] <- rmse(y_test, y_pred_test)
}

# Plot RMSE
par(mfrow = c(1, 1))
plot(degree_range, train_rmse, type = "b", col = "blue", pch = 16, lwd = 2,
     ylim = c(0, max(train_rmse, test_rmse)), xlab = "Polynomial Degree (n)",
     ylab = "RMSE", main = "Training and Testing RMSE")
lines(degree_range, test_rmse, type = "b", col = "red", pch = 16, lwd = 2)
legend("topright", legend = c("Train RMSE", "Test RMSE"),
       col = c("blue", "red"), pch = 16, lty = 1, lwd = 2)
```



```
# Output RMSE values
cat("\nTraining RMSE for each degree:\n")
```

```
##
## Training RMSE for each degree:
```

```r
print(train_rmse)
```

```
##  [1] 3.6536923 3.4480673 1.0717612 0.9508293 0.3592377 0.3567345 0.3435260
##  [8] 0.3239865 0.3097750 0.2110863
```

```r
cat("\nTesting RMSE for each degree:\n")
```

```
##
## Testing RMSE for each degree:
```

```r
print(test_rmse)
```

```
##  [1]  3.4329892  3.4941840  1.0296472  1.1379610  0.6048368  0.6080990
##  [7]  0.6092413  1.2432137  5.3596792 51.4885920
```