Assignment 9: Exploring the Optimal Proposal Distribution in the Metropolis-Hastings Algorithm

yifan

2025-04-14

Introduction

In class, we learned about the method of sampling from a known distribution using the random walk Metropolis algorithm, which is a special case of the Metropolis-Hastings algorithm, and this in turn is a special case of MCMC. The basic approach is:

Starting with $X^{(0)} := (X_1^{(0)}, \dots, X_p^{(0)})$ and using a symmetric distribution g, iterate for $t = 1, 2, \dots$

1. Draw $\epsilon \sim g$ and set $X = X^{(t-1)} + \epsilon$.
2. Compute

$$\alpha(X \mid X^{(t-1)}) = \min\left\{1, \frac{f(X)}{f(X^{(t-1)})}\right\}.$$

3. With probability $\alpha(X \mid X^{(t-1)})$ set $X^{(t)} = X$, otherwise set $X^{(t)} = X^{(t-1)}$.

Thus, the essence of this algorithm is choosing the distribution of g, which isn't a trivial issue. Intuitively, we would want the distribution of g to be as similar as f as possible. Therefore, in this assignment, I will use an example from the textbook to explore this issue.

Scenario description

Consider the example of a medical study on infections resulting from birth by Cæsarean section (taken from Fahrmeir and Tutz, 2001). We try to model the influence of three factors: an indicator whether the Cæsarian was planned or not (z_{i1}), an indicator of whether additional risk factors were present at the time of birth (z_{i2}), and an indicator of whether antibiotics were given as a prophylaxis (z_{i3}). The response Y_i is the number of infections that were observed amongst n_i patients having the same influence factors (covariates). The observed data is shown in the table below.

Number of births		planned	risk factors	antibiotics
with infection	total			
y_i	n_i	z_{i1}	z_{i2}	z_{i3}
11	98	1	1	1
1	18	0	1	1
0	2	0	0	1
23	26	1	1	0
28	58	0	1	0
0	9	1	0	0
8	40	0	0	0

Statistical Modeling

As suggested by the textbook, we model this example by assuming:

$$Y_i \sim Bin(n_i, \pi_i), \pi_i = \Phi(z_i'\beta) \tag{1}$$

Where $z_i = (1, z_1, z_2, z_3)$, $\beta = (\beta_0, \beta_1, \beta_2, \beta_3)$ being the parameters to estimate, $\Phi(\cdot)$ being the CDF of the $N(0, 1)$ distribution.

We assume the prior distribution of β as $\beta \sim N(0, I/\lambda)$. We can easily obtain the posterior distribution:

$$\begin{aligned} f(\beta|y_1, \cdots, y_n) &= \frac{f(\beta, y_1, \cdots, y_n)}{f(y_1, \cdots, y_n)} \propto f(\beta, y_1, \cdots, y_n) = f(y_1, \cdots, y_n|\beta)f(\beta) \\ &\propto \left(\prod_{i=1}^{N} \Phi(z_i'\beta)^{y_i} \cdot (1 - \Phi(z_i'\beta))^{n_i - y_i} \right) \cdot \exp\left(-\frac{\lambda}{2} \sum_{j=0}^{3} \beta_j^2 \right) \end{aligned} \tag{2}$$

Now, we try to sample from the posterior distribution using the random walk Metropolis algorithm. Since the elements in β can be both positive or negative, we choose $\epsilon \sim N(0, \Sigma)$ (here, ϵ is the same as the aforementioned g). In the textbook, we chose $\Sigma = 0.08I$ without explaining why. Moreover, we were told that we could estimate the asymptotic covariance of the MLE of β using the frequentist theory. Therefore, in this assignment, we will explore whether $\alpha = 0.08$ is the optimal choice if we assume $\Sigma = \alpha I$, and whether if estimating the asymptotic covariance of $\hat{\beta}$ could indeed enhance the performance of the random walk Metropolis algorithm.

Optimal choice of α

In this section, we assume that $\Sigma = \alpha I$, i.e the random variables in ϵ are independent.

First, we have to do some preparations (input data into R):

```r
y <- c(11, 1, 0, 23, 28, 0, 8)
n <- c(98, 18, 2, 26, 58, 9, 40)
Z <- matrix(c(
  1, 1, 1, 1, 1, 1, 1,
  1, 0, 0, 1, 0, 1, 0,
  1, 1, 0, 1, 1, 0, 0,
  1, 1, 1, 0, 0, 0, 0
), nrow = 7, byrow = FALSE)
```

Now, we establish the function for calculating the posterior value given β:

```r
post_value <- function(N, Z, beta, y, n, lambda) {
  # Compute linear predictor: Z %*% beta (N x 1 vector)
  linear_pred <- Z %*% beta

  # Calculate Phi(Z_i'beta) and 1-Phi(Z_i'beta) (N x 1 vectors)
  Phi <- pnorm(linear_pred)        # Standard normal CDF
  inv_Phi <- 1 - Phi               # Complementary CDF

  # Calculate likelihood term: prod[ Phi^{y_i} * (1-Phi)^{n_i-y_i} ]
  likelihood <- prod(Phi^y * inv_Phi^(n - y))

  # Calculate regularization term: exp(-lambda/2 * sum(beta^2))
  penalty <- exp(-lambda/2 * sum(beta^2))
  # Return the joint posterior value
  return(likelihood * penalty)
}
```

Here, we notice that according to the algorithm, all that matters is the value $\frac{f(\beta|y_1,\cdots,y_n)}{f(\beta^{(t-1)}|y_1,\cdots,y_n)}$. Thus, to ensure numerical stability, we use the log posterior function instead:

```r
# Posterior function (log-scale for numerical stability)
log_posterior <- function(beta, Z, y, n, lambda) {
  linear_pred <- Z %*% beta
  log_Phi <- pnorm(linear_pred, log.p = TRUE)
  log_inv_Phi <- pnorm(linear_pred, log.p = TRUE, lower.tail = FALSE)
  log_lik <- sum(y * log_Phi + (n - y) * log_inv_Phi)
  log_prior <- -lambda/2 * sum(beta^2)
  return(log_lik + log_prior)
}
```

And when we have to calculate the fraction of the posterior values, we calculate the exponential value of the fraction of the log posterior values instead. Next, we implement the M-H algorithm:

```r
# Metropolis-Hastings algorithm
run_mcmc <- function(n_iter, Z, y, n, lambda, Sigma) {
  p <- ncol(Z)
  beta_samples <- matrix(NA, nrow = n_iter, ncol = p)
  beta_current <- rep(0, p)  # Initialize at zero

  for (t in 1:n_iter) {
    # 1. Propose new beta
    epsilon <- rmvnorm(1, mean = rep(0, p), sigma = Sigma)[1, ]
    beta_proposed <- beta_current + epsilon

    # 2. Compute acceptance ratio
    log_alpha <- log_posterior(beta_proposed, Z, y, n, lambda) -
                 log_posterior(beta_current, Z, y, n, lambda)
    alpha <- min(1, exp(log_alpha))

    # 3. Accept/reject
    if (runif(1) < alpha) {
      beta_current <- beta_proposed
    }

    beta_samples[t, ] <- beta_current
  }

  return(beta_samples)
}
```

Next, we have to choose the value of λ (this part isn't included in the textbook). Since we don't want to include too much information in the posterior distribution, we would want the posterior distribution of β to be flat, i.e $1/\lambda$ should be large enough. Thus, we choose $\lambda = 0.1$.

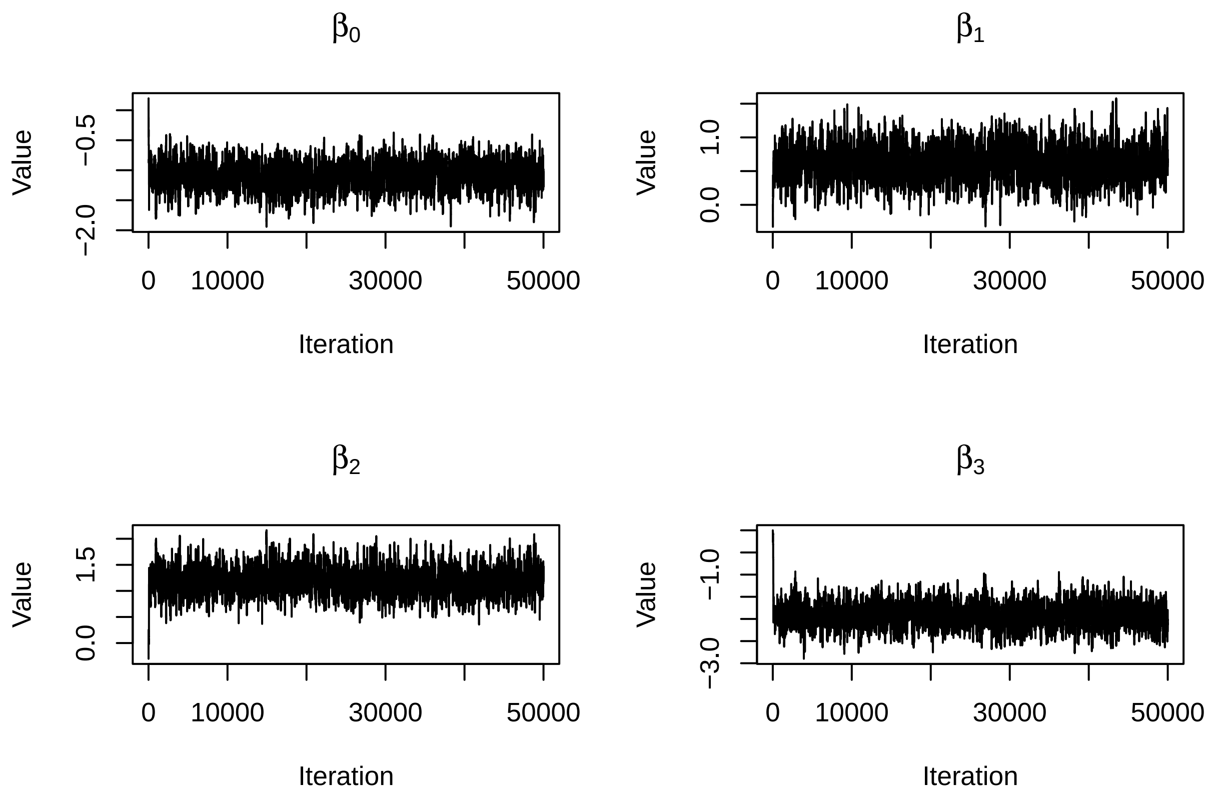
```r
lambda <- 0.1
Sigma <- diag(0.08, nrow = 4) # Since Σ = 0.08I
```

Finally, we can run the algorithm (with 50000 iterations):

```r
# Run MCMC
set.seed(123)
n_iter <- 50000
beta_samples <- run_mcmc(n_iter, Z, y, n, lambda, Sigma)
```

We can first plot the traces of β:

```r
# Plot the traces
par(mfrow = c(2, 2))
for (j in 1:4) {
  plot(beta_samples[, j], type = "l",
       main = bquote(beta[.(j-1)]),
       xlab = "Iteration", ylab = "Value")
}
```



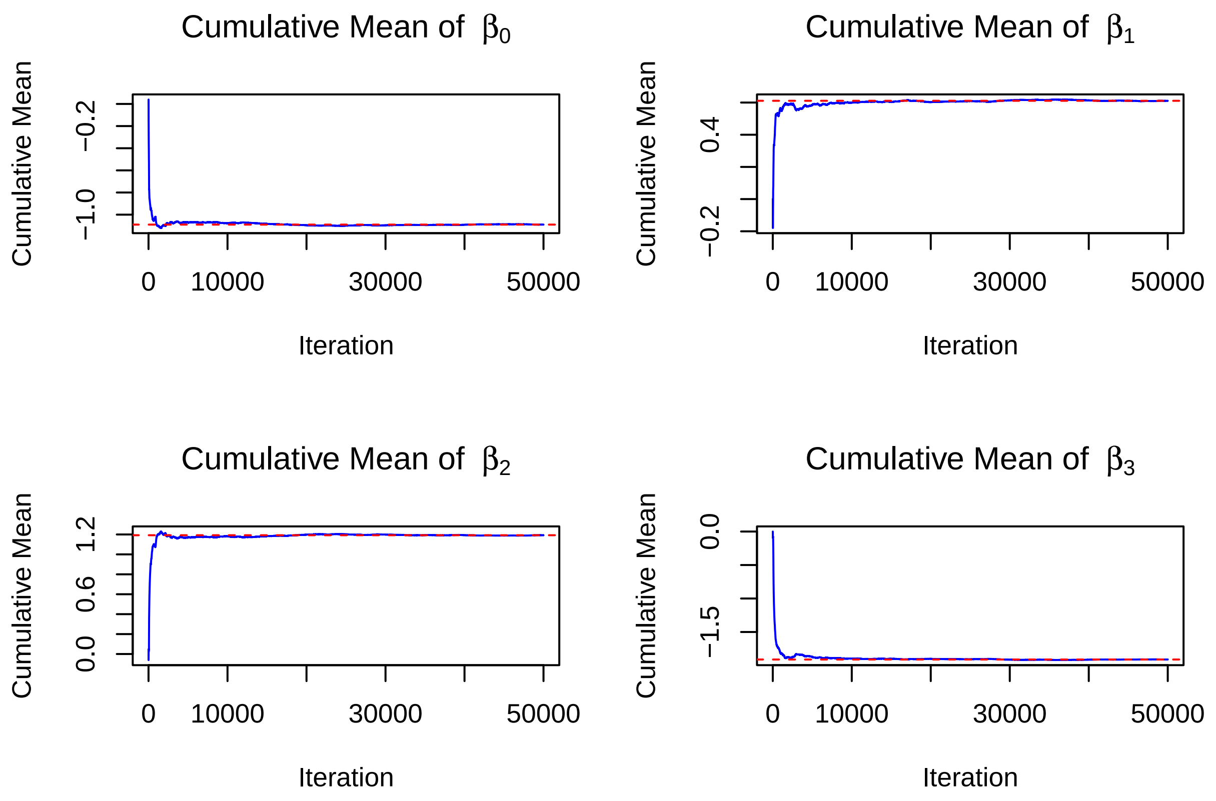
As well as the cumulative means of each parameter:

```r
# Calculate cumulative means for each parameter
# cumsum() computes running totals, divided by iteration count gives running averages
cumulative_means <- apply(beta_samples, 2, function(x) cumsum(x) / seq_along(x))
colnames(cumulative_means) <- c("beta0", "beta1", "beta2", "beta3")

# Set up 2x2 plotting layout
par(mfrow = c(2, 2))

# Plot cumulative means for each parameter
for (j in 1:4) {
  plot(cumulative_means[, j],
       type = "l",
       main = bquote("Cumulative Mean of " ~ beta[.(j-1)]),
       xlab = "Iteration",
       ylab = "Cumulative Mean",
       col = "blue")

  # Add reference line (final posterior mean)
  abline(h = mean(beta_samples[, j]), col = "red", lty = 2)
}
```



As suggested in textbook, we discard the first 10000 values, and obtain the mean values:

```r
post_samples <- beta_samples[10001:50000, ]
posterior_means <- colMeans(post_samples)
posterior_means
```

```
## [1] -1.092744  0.613635  1.194473 -1.914503
```

As well as the 95% CIs of the parameters:

```r
beta_CI_95 <- apply(post_samples, 2, quantile, probs = c(0.025, 0.975))
beta_CI_95
```

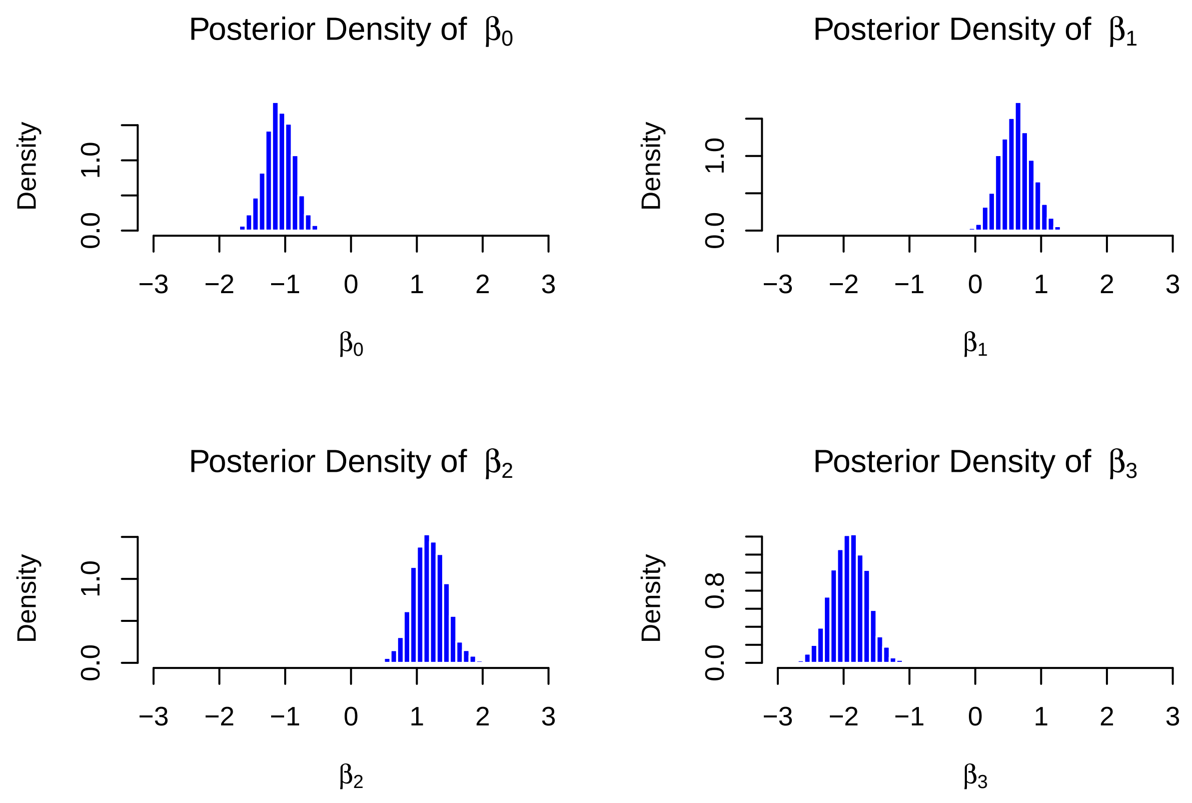
```
##             [,1]      [,2]      [,3]      [,4]
## 2.5%  -1.534629 0.1486033 0.7234641 -2.436888
## 97.5% -0.673628 1.1020927 1.7238048 -1.382434
```

Finally, we can plot the posterior densities of the parameters:

```r
# Set up 2x2 plotting layout
par(mfrow = c(2, 2))

# Define consistent breaks for all parameters
bin_breaks <- seq(floor(min(post_samples)),
                  ceiling(max(post_samples)),
                  by = 0.1)

# Create density histograms for each parameter
for (j in 1:4) {
  hist(post_samples[, j],
       breaks = bin_breaks,
       main = bquote("Posterior Density of " ~ beta[.(j-1)]),
       xlab = bquote(beta[.(j-1)]),
       ylab = "Density",
       col = "blue",
       border = "white",
       xlim = range(bin_breaks),
       freq = FALSE)
}
```



The results we obtained are similar to those in the textbook.

Furthermore, we want to evaluate the performance of the algorithm. We know that since our method is MCMC-based, the samples we generate are inevitably correlated. However, since i.i.d samples are the most ideal samples, we want the correlation of the samples we generate to be as small as possible. Therefore, we estimate the value of average autocorrelation $\rho(\beta_j^{(t-1)}, \beta_j^{(t)}), t = 0, \cdots, 3$ by computing the Pearson correlation of $(\beta_j^1, \cdots, \beta_j^{49999})$ and $(\beta_j^2, \cdots, \beta_j^{50000})$

```r
# Calculate Lag-1 autocorrelations for each parameter column
beta_cor <- numeric(4)  # Initialize vector to store results

for (j in 1:4) {
  # Extract the chain for parameter j
  chain <- beta_samples[, j]

  # Calculate correlation between (X_t-1, X_t) pairs
  beta_cor[j] <- cor(chain[-length(chain)], chain[-1])
}

# Name and print results
names(beta_cor) <- c("beta0", "beta1", "beta2", "beta3")
print("Lag-1 Autocorrelation Coefficients:")
```

```
## [1] "Lag-1 Autocorrelation Coefficients:"
```

```r
print(beta_cor)
```

```
##     beta0     beta1     beta2     beta3
## 0.9470038 0.9467540 0.9547270 0.9543514
```

Now that we have a metric measuring the correlation of the chain, we seek to find the optimal α (assuming that $\Sigma = \alpha I$) by doing a grid search between $\alpha = 0.02$ to $\alpha = 0.12$, with a step size of 0.02

```r
Sigma <- diag(0.06, nrow = 4)
beta_samples_0.06 <- run_mcmc(n_iter, Z, y, n, lambda, Sigma)

beta_cor_0.06 <- numeric(4)

for (j in 1:4) {
  chain <- beta_samples_0.06[, j]

  beta_cor_0.06[j] <- cor(chain[-length(chain)], chain[-1])
}

names(beta_cor_0.06) <- c("beta0", "beta1", "beta2", "beta3")
print("Lag-1 Autocorrelation Coefficients when alpha = 0.06:")
```

```
## [1] "Lag-1 Autocorrelation Coefficients when alpha = 0.06:"
```

```r
print(beta_cor_0.06)
```

```
##     beta0     beta1     beta2     beta3
## 0.9432157 0.9483198 0.9516744 0.9514953
```

```r
Sigma <- diag(0.04, nrow = 4)
beta_samples_0.04 <- run_mcmc(n_iter, Z, y, n, lambda, Sigma)

beta_cor_0.04 <- numeric(4)

for (j in 1:4) {
  chain <- beta_samples_0.04[, j]

  beta_cor_0.04[j] <- cor(chain[-length(chain)], chain[-1])
}
```

```r
names(beta_cor_0.04) <- c("beta0", "beta1", "beta2", "beta3")
print("Lag-1 Autocorrelation Coefficients when alpha = 0.04:")
```

```
## [1] "Lag-1 Autocorrelation Coefficients when alpha = 0.04:"
```

```r
print(beta_cor_0.04)
```

```
##     beta0     beta1     beta2     beta3
## 0.9441400 0.9504723 0.9532999 0.9556235
```

```r
Sigma <- diag(0.02, nrow = 4)
beta_samples_0.02 <- run_mcmc(n_iter, Z, y, n, lambda, Sigma)

beta_cor_0.02 <- numeric(4)

for (j in 1:4) {
  chain <- beta_samples_0.02[, j]

  beta_cor_0.02[j] <- cor(chain[-length(chain)], chain[-1])
}

names(beta_cor_0.02) <- c("beta0", "beta1", "beta2", "beta3")
print("Lag-1 Autocorrelation Coefficients when alpha = 0.02")
```

```
## [1] "Lag-1 Autocorrelation Coefficients when alpha = 0.02"
```

```r
print(beta_cor_0.02)
```

```
##     beta0     beta1     beta2     beta3
## 0.9490456 0.9554541 0.9596411 0.9585049
```

```r
Sigma <- diag(0.1, nrow = 4)
beta_samples_0.1 <- run_mcmc(n_iter, Z, y, n, lambda, Sigma)

beta_cor_0.1 <- numeric(4)

for (j in 1:4) {
  chain <- beta_samples_0.1[, j]

  beta_cor_0.1[j] <- cor(chain[-length(chain)], chain[-1])
}

names(beta_cor_0.1) <- c("beta0", "beta1", "beta2", "beta3")
print("Lag-1 Autocorrelation Coefficients when alpha = 0.1:")
```

```
## [1] "Lag-1 Autocorrelation Coefficients when alpha = 0.1:"
```

```r
print(beta_cor_0.1)
```

```
##     beta0     beta1     beta2     beta3
## 0.9511579 0.9499198 0.9561460 0.9555180
```

```r
Sigma <- diag(0.12, nrow = 4)
beta_samples_0.12 <- run_mcmc(n_iter, Z, y, n, lambda, Sigma)

beta_cor_0.12 <- numeric(4)
```

```r
for (j in 1:4) {
  chain <- beta_samples_0.12[, j]

  beta_cor_0.12[j] <- cor(chain[-length(chain)], chain[-1])
}

names(beta_cor_0.12) <- c("beta0", "beta1", "beta2", "beta3")
print("Lag-1 Autocorrelation Coefficients when alpha = 0.12:")
```

```
## [1] "Lag-1 Autocorrelation Coefficients when alpha = 0.12:"
```

```r
print(beta_cor_0.12)
```

```
##     beta0     beta1     beta2     beta3
## 0.9539326 0.9533637 0.9569136 0.9566065
```

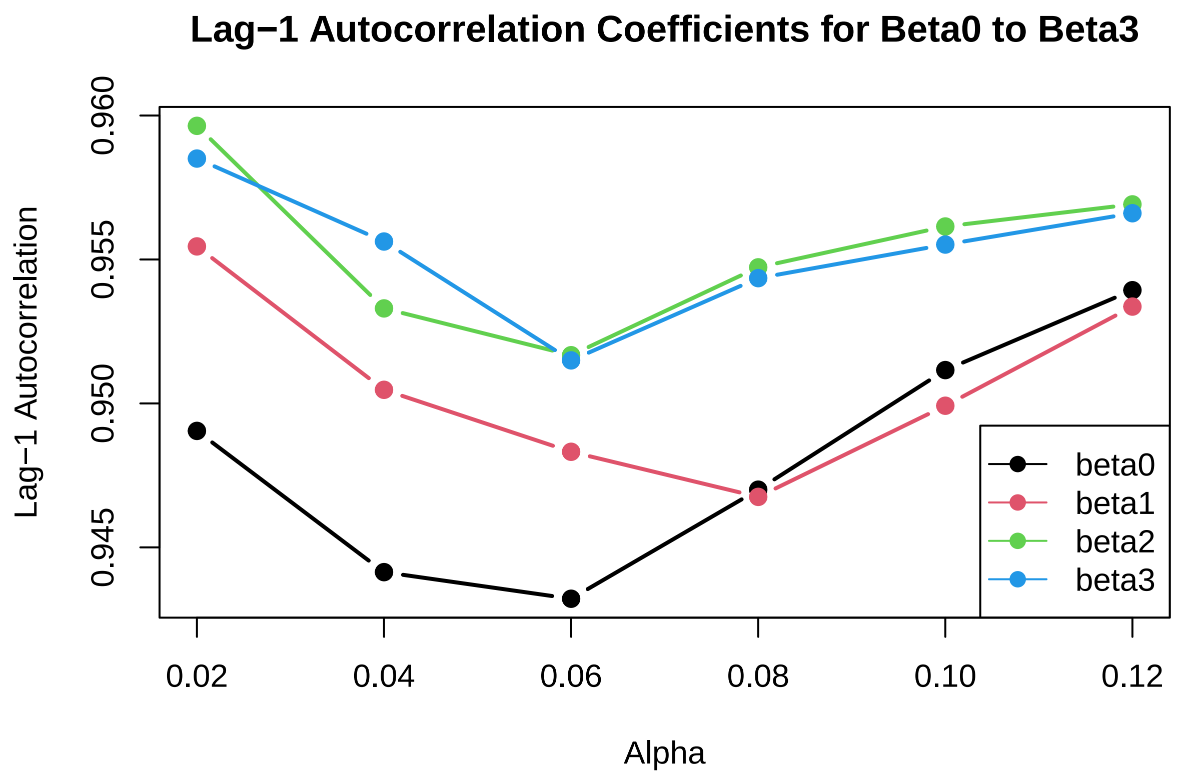
Now we plot the line chart of the autocorrelation of $(\beta^{(t-1)}, \beta^{(t)})$ as a function of (α)

```r
# Organize the data for different alpha values
beta_cor_matrix <- rbind(beta_cor_0.02, beta_cor_0.04, beta_cor_0.06,
                         beta_cor, beta_cor_0.1, beta_cor_0.12)

# Set the column names and row names for the matrix
colnames(beta_cor_matrix) <- c("beta0", "beta1", "beta2", "beta3")
rownames(beta_cor_matrix) <- paste("alpha =", seq(0.02, 0.12, by = 0.02))

# Use matplot to plot the lag-1 autocorrelation coefficients as a function of alpha
matplot(seq(0.02, 0.12, by = 0.02), beta_cor_matrix, type = "b", pch = 19, col = 1:4,
        xlab = "Alpha", ylab = "Lag-1 Autocorrelation",
        main = "Lag-1 Autocorrelation Coefficients for Beta0 to Beta3",
        lty = 1, lwd = 2)

# Add a legend to the top-right corner of the plot
legend("bottomright", legend = c("beta0", "beta1", "beta2", "beta3"),
       col = 1:4, pch = 19, lty = 1)
```



It can be seen that $\alpha = 0.06$ seems like a better choice than the original parameter $\alpha = 0.08$. However, the improvement is marginal, and there isn't a need to engage in a finer grid search. Perhaps choosing an ϵ with correlation (i.e stop assuming that Σ is diagonal) may produce a better result.

Optimal choice of Σ

We noticed that changing the value of α doesn't make such a big difference, and that the autocorrelation of $(\beta^{(t-1)}, \beta^{(t)})$ remains higher than 0.94 no matter how we adjust α. This might be due to the strong assumption we posed on Σ (assuming that the covariates in Σ are independent). Now in this section, we remove this constraint and obtain Σ by estimating the asymptotic covariance of the MLE of β using frequentist theory. We assume that this could improve the performance of the random walk algorithm.

The textbook suggests that we use GLM theory to estimate the aforementioned asymptotic covariance by

$$\hat{Var}(\hat{\beta}) = c(z^T D z)^{-1} \tag{3}$$

where c being a constant and D being a suitable diagonal matrix. However, here we try to explore a different approach using the idea of Fisher information matrix, and we can show that both methods obtain similar results.

First, the likelihood of β based on the observed data can be written as:

$$L(\beta) = \prod_{i=1}^{7} \pi_i^{y_i} (1 - \pi_i)^{n_i - y_i} \tag{4}$$

so that the log-likelihood is:

$$\ell(\beta) = \sum_{i=1}^{7} \left[y_i \log \Phi(z_i^T \beta) + (n_i - y_i) \log(1 - \Phi(z_i^T \beta)) \right] \tag{5}$$

The score function can be obtained by taking the derivative of the log-likelihood w.r.t β:

$$S(\beta) = \frac{\partial \ell(\beta)}{\partial \beta} = \sum_{i=1}^{7} \left[\frac{y_i - n_i \Phi(z_i^T \beta)}{\Phi(z_i^T \beta)(1 - \Phi(z_i^T \beta))} \cdot \phi(z_i^T \beta) \cdot z_i \right] \tag{6}$$

The Fisher information matrix is the variance of the score function, i.e $I(\beta) = Var(S(\beta)) = E\left[S(\beta)S(\beta)^T\right] - E\left[S(\beta)\right] E\left[S(\beta)\right]^T$. We notice that for $\beta = \beta_{MLE}$, $S(\beta) = 0$, hence $I(\beta) = E\left[S(\beta)S(\beta)^T\right]$.

Since we assume that $Y_i \sim Bin(n_i, \pi_i)$, we know that $E(Y_i) = n_i \Phi(z_i^T \beta)$ and $Var(Y_i) = n_i \Phi(z_i^T \beta)(1 - \Phi(z_i^T \beta))$, so that

$$E[S(\beta)S(\beta)^T] = \sum_{i=1}^{n} \left[\frac{\phi(z_i^T \beta)^2}{\Phi(z_i^T \beta)^2 (1 - \Phi(z_i^T \beta))^2} \cdot \mathrm{Var}(Y_i) \cdot z_i z_i^T \right] = \sum_{i=1}^{n} \left[\frac{n_i \phi(z_i^T \beta)^2}{\Phi(z_i^T \beta)(1 - \Phi(z_i^T \beta))} \cdot z_i z_i^T \right] \tag{7}$$

Finally, we can obtain the asymptotic variance of the MLE of β:

$$AsymptoticVar(\hat{\beta}) = I(\hat{\beta})^{-1} \tag{8}$$

We can use some numerical methods to solve for the MLE of β, such as Newton-Raphson or Fisher Scoring. Here, we choose BFGS Quasi-Newton Method to solve for $\hat{\beta}$. Below is the R-implementation:

```r
# Negative log-likelihood function for Probit model
neg_log_lik <- function(beta) {
  eta <- Z %*% beta
  pi <- pnorm(eta)
  -sum(y * log(pi) + (n - y) * log(1 - pi))
}

# Initial values (all zeros)
beta_init <- rep(0, ncol(Z))

# Optimize to find MLE using BFGS method
fit <- optim(beta_init, neg_log_lik, method = "BFGS", hessian = TRUE)
beta_hat <- fit$par

# Calculate observed information matrix
phi <- dnorm(Z %*% beta_hat)  # standard normal PDF
Phi <- pnorm(Z %*% beta_hat)  # standard normal CDF
W <- n * (phi^2) / (Phi * (1 - Phi))  # weights
I_beta <- t(Z) %*% diag(as.vector(W)) %*% Z  # Fisher information matrix

# Asymptotic variance (inverse of information matrix)
asymptotic_var <- solve(I_beta)
asymptotic_var
```

```
##             [,1]         [,2]         [,3]        [,4]
## [1,]  0.04983393 -0.014308281 -0.045909499  0.00917574
## [2,] -0.01430828  0.058925356 -0.001742937 -0.03803933
## [3,] -0.04590950 -0.001742937  0.066143756 -0.01858566
## [4,]  0.00917574 -0.038039331 -0.018585659  0.06943386
```

Now, we use the estimated asymptotic variance of $\hat{\beta}$ as Σ, and we run the random walk algorithm and calculate the lag-1 autocorrelation of the coefficients:

```r
beta_samples <- run_mcmc(n_iter, Z, y, n, lambda, asymptotic_var)

beta_cor <- numeric(4)

for (j in 1:4) {
  chain <- beta_samples[, j]

  beta_cor[j] <- cor(chain[-length(chain)], chain[-1])
}

names(beta_cor) <- c("beta0", "beta1", "beta2", "beta3")
print("Lag-1 Autocorrelation Coefficients:")
```

```
## [1] "Lag-1 Autocorrelation Coefficients:"
```

```r
print(beta_cor)
```

```
##     beta0     beta1     beta2     beta3
## 0.8648962 0.8723194 0.8683877 0.8731234
```

The results we obtain is similar to the results in the textbook, suggesting that both methods could enhance the performance of the MCMC algorithm.

Conclusion

In this assignment, we explored the optimal choice of the proposal distribution in the MCMC algorithm in the following aspects (assuming that $\epsilon \sim N(0, \Sigma)$:

(1) Assuming that $\Sigma = \alpha I$ and adjust for α;

(2) Removing assumption (1) and estimating Σ using the idea of Fisher information matrix.

We found out that adjusting α doesn't make such a big difference, and the lag-1 autocorrelation of the coefficients remain higher than 0.94. However, if we are able to estimate Σ properly without assumption (1), we can reduce the lag-1 autocorrelation of the coefficients to approximately 0.87.