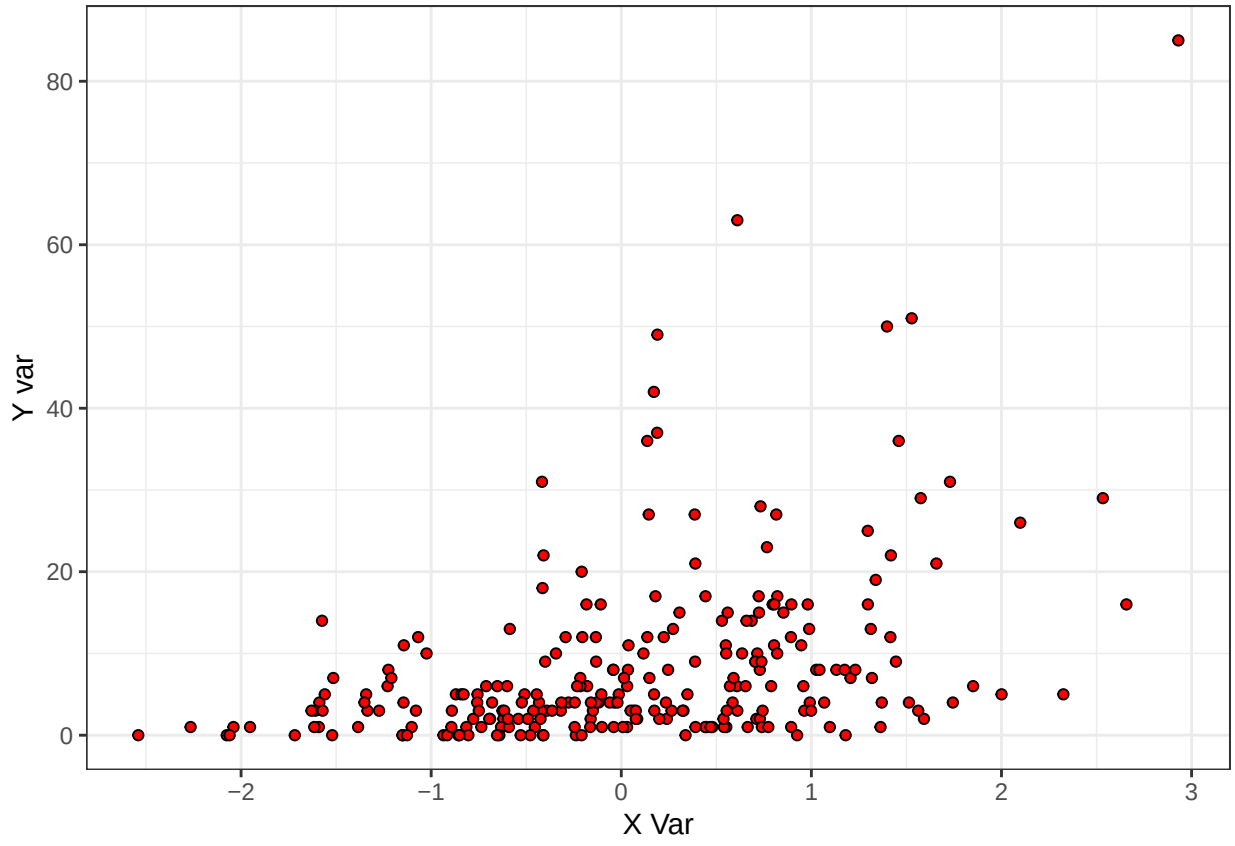
BIS 567 HW0

```
library(here)
```

```
## here() starts at /Users/colebrookson/github/fall-courses-2024
```

```
library(magrittr)
load(here::here("./bis567/homework/hw04/HW4.RData"))
n <- length(y)

ggplot2::ggplot(data = data.frame(x, y)) +
    ggplot2::geom_point(ggplot2::aes(x = x, y = y),
        colour = "black", fill = "red",
        shape = 21
    ) +
    ggplot2::theme_bw() +
    ggplot2::labs(x = "X Var", y = "Y var")
```



The model takes the form of

$$Y_i|\lambda_i \overset{\text{iid}}{\sim} \text{Poisson}(\lambda_i), i = 1, ..., n \tag{1}$$

where

$$\ln(\lambda_i) = \beta_0 + \beta_1 x_i + \phi_i \tag{2}$$

and

$$\phi_i|\sigma^2 \overset{\text{iid}}{\sim} \text{N}(0, \sigma^2). \tag{3}$$

If we have the priors of

$$\beta_0 \sim \text{N}(0, 100^2)\beta_1 \sim \text{N}(0, 100^2)\sigma^2 \sim \text{IG}(0.01, 0.01) \tag{4}$$

Question 1A:

```
# since we have to work on the log scale, just a little helper to get the log-
# liklihood for the poisson with log-link
log_likelihood <- function(y, x, beta0, beta1, phi) {
    lambda_i <- exp(beta0 + beta1 * x + phi)
    return(sum(y * log(lambda_i) - lambda_i))
}

# since beta0 and beta1 are the same, we can just use a function to do our
# metropolis hasting steps
metropolis_beta <- function(
    y, x, beta0_current, beta1_current, phi_current,
    beta0_prior_sd, beta1_prior_sd, delta_beta0, delta_beta1) {
    # first make new proposals
    beta0_proposal <- beta0_current + stats::rnorm(1,
        mean = 0,
        sd = delta_beta0
    )
    beta1_proposal <- beta1_current + stats::rnorm(1,
        mean = 0,
        sd = delta_beta1
    )

    # calculate the log-liklihood for each one given
    # the current and the proposed
    ll_current <- log_likelihood(
        y, x, beta0_current,
        beta1_current, phi_current
    )
    ll_proposal <- log_likelihood(
        y, x, beta0_proposal,
        beta1_proposal, phi_current
    )

    # Log-priors for current and proposed values
    # ok need to calculate the priors and make sure they're on the log scale
    log_prior_current <-
        stats::dnorm(beta0_current, mean = 0, sd = beta0_prior_sd, log = TRUE) +
        stats::dnorm(beta1_current, mean = 0, sd = beta1_prior_sd, log = TRUE)
    log_prior_proposal <-
        stats::dnorm(beta0_proposal, mean = 0, sd = beta0_prior_sd, log = TRUE) +
        stats::dnorm(beta1_proposal, mean = 0, sd = beta1_prior_sd, log = TRUE)
```

```
    # the acceptance ratio is just the log-lik. (in this case the way I've
    # chosen to do it) of the proposal - the current value
    log_acceptance_ratio <- (ll_proposal + log_prior_proposal) - (ll_current + log_prior_current)

    # we want to pull a value here as the proposal (using a uniform dist)
    if (log(stats::runif(1)) < log_acceptance_ratio) { # accept
        # want to keep track of all three things here
        return(c(beta0_proposal, beta1_proposal, 1))
    } else {
        return(c(beta0_current, beta1_current, 0))
    }
}

# the gibbs is much easier, we can just pull from the actual distribution
sample_sigma2 <- function(phi_current, alpha_sigma2, beta_sigma2) {
    alpha_post <- n / 2 + alpha_sigma2
    beta_post <- sum(phi_current^2) / 2 + beta_sigma2
    return(1 / stats::rgamma(1, shape = alpha_post, rate = beta_post))
}

# we want the mean, sd, and 95% ci once we've actually DONE all of the sampling
# stuff, and then we want it in a table
summary_statistics <- function(samples) {
    mean_value <- mean(samples)
    sd_value <- sd(samples)
    ci_95 <- quantile(samples, c(0.025, 0.975))
    return(c(
        mean = mean_value, sd = sd_value,
        ci_95_lower = ci_95[[1]], ci_95_upper = ci_95[[2]]
    ))
}

set.seed(123) # reproduce
n <- length(y) # number of observations
n_iter <- 100000 # need to crank this up to get good inference

# these are empty objects to store the things we're interested in inside,
# note phi needs to have the same number of columns as observations since
# it's phi_j
beta0_samples <- numeric(n_iter)
beta1_samples <- numeric(n_iter)
sigma2_samples <- numeric(n_iter)
phi_samples <- matrix(0, nrow = n_iter, ncol = n)

# choosing really naive value not going to bother using a regression to get
# starting values since it's an easy problem
beta0_current <- 0
beta1_current <- 0
sigma2_current <- 1
phi_current <- rep(0, n)

# the prior parameter values here
beta0_prior_mean <- 0
```

```r
beta0_prior_sd <- 100
beta1_prior_mean <- 0
beta1_prior_sd <- 100
alpha_sigma2 <- 0.01
beta_sigma2 <- 0.01

# i want to keep track of what the acceptance rates are like, so just going to
# count for each thing
beta_accept_counter <- 0
phi_accept_counter <- rep(0, n)

# I want to be able to pick how big the step sizes are (delta) so I'm setting
# them each individuallsy (technically the deltas could be the same thign)
delta_beta0 <- 0.01 # step size for beta0
delta_beta1 <- 0.01 # step size for beta1
delta_phi <- 0.5 # step size for each phi_i

# sample in one big loop
start_a <- Sys.time()
for (t in 1:n_iter) {
    ## STEP 1 ## sample beta0 and beta1 via the sampling function from above
    betas <- metropolis_beta(
        y, x, beta0_current, beta1_current, phi_current, beta0_prior_sd,
        beta1_prior_sd, delta_beta0, delta_beta1
    )
    beta0_current <- betas[1]
    beta1_current <- betas[2]
    beta_accept_counter <- beta_accept_counter + betas[3] # track acceptance

    ## STEP 2 ## again we need MH here but for each value of phi (n) values
    for (i in 1:n) {
        # make proposal and get the lambda values
        phi_proposal <- phi_current[i] +
            stats::rnorm(1, mean = 0, sd = delta_phi)
        lambda_current <- exp(beta0_current + beta1_current *
            x[i] + phi_current[i])
        lambda_proposal <- exp(beta0_current + beta1_current *
            x[i] + phi_proposal)
        # calculate the acceptance ratio - don't need to sum here cos it's in
        # the loop
        accept_ratio <- (y[i] * log(lambda_proposal) - lambda_proposal) -
            (y[i] * log(lambda_current) - lambda_current)
        accept_ratio <- accept_ratio +
            stats::dnorm(phi_proposal,
                mean = 0,
                sd = sqrt(sigma2_current), log = TRUE
            ) -
            stats::dnorm(phi_current[i],
                mean = 0,
                sd = sqrt(sigma2_current), log = TRUE
            )
        # this statement decides if we're accepting or rejecting and then
        # count the acceptances if we can
```

```
        if (log(stats::runif(1)) < accept_ratio) {
            phi_current[i] <- phi_proposal
            phi_accept_counter[i] <- phi_accept_counter[i] + 1 # track acceptance
        }
    }

    ## STEP 3 ## sample sigma2 from its full conditional with gibbs
    sigma2_current <- sample_sigma2(phi_current, alpha_sigma2, beta_sigma2)

    # store the samples
    beta0_samples[t] <- beta0_current
    beta1_samples[t] <- beta1_current
    sigma2_samples[t] <- sigma2_current
    phi_samples[t, ] <- phi_current

    # leaving this here even though I don't really need it for the submission,
    # but so you can see that I tuned the deltas w/ this
    if (i == 100000) {
        cat("Iteration:", i, "\n")
        cat("Acceptance rate for beta0:", beta_accept_counter / i, "\n")
        cat("Acceptance rate for beta1:", beta_accept_counter / i, "\n")
        cat("Average acceptance rate for phi:", mean(phi_accept_counter / i), "\n")
    }
}
end_a <- Sys.time()

# pre-defined function for the summary stats
beta0_summary <- summary_statistics(beta0_samples)
beta1_summary <- summary_statistics(beta1_samples)
sigma2_summary <- summary_statistics(sigma2_samples)

# summary table! (excluding ln(lambda_i))
knitr::kable(
    data.frame(
        Parameter = c("beta0", "beta1", "sigma2"),
        `Posterior Mean` = c(
            beta0_summary["mean"],
            beta1_summary["mean"], sigma2_summary["mean"]
        ),
        `Posterior SD` = c(
            beta0_summary["sd"],
            beta1_summary["sd"], sigma2_summary["sd"]
        ),
        `95% Credible Interval Lower` = c(
            beta0_summary[["ci_95_lower"]],
            beta1_summary[["ci_95_lower"]], sigma2_summary[["ci_95_lower"]]
        ),
        `95%_Credible_Interval_Upper` = c(
            beta0_summary[["ci_95_upper"]],
            beta1_summary[["ci_95_upper"]], sigma2_summary[["ci_95_upper"]]
        )
    ),
    caption = "Parameter estimates from method 1 (Q1A)",
```

```
    col.names = c(
        "Parameter", "Post. Mean", "Post SD", "Post 95% Lower",
        "Post 95% Upper"
    )
) %>%
    kableExtra::kable_classic()
```

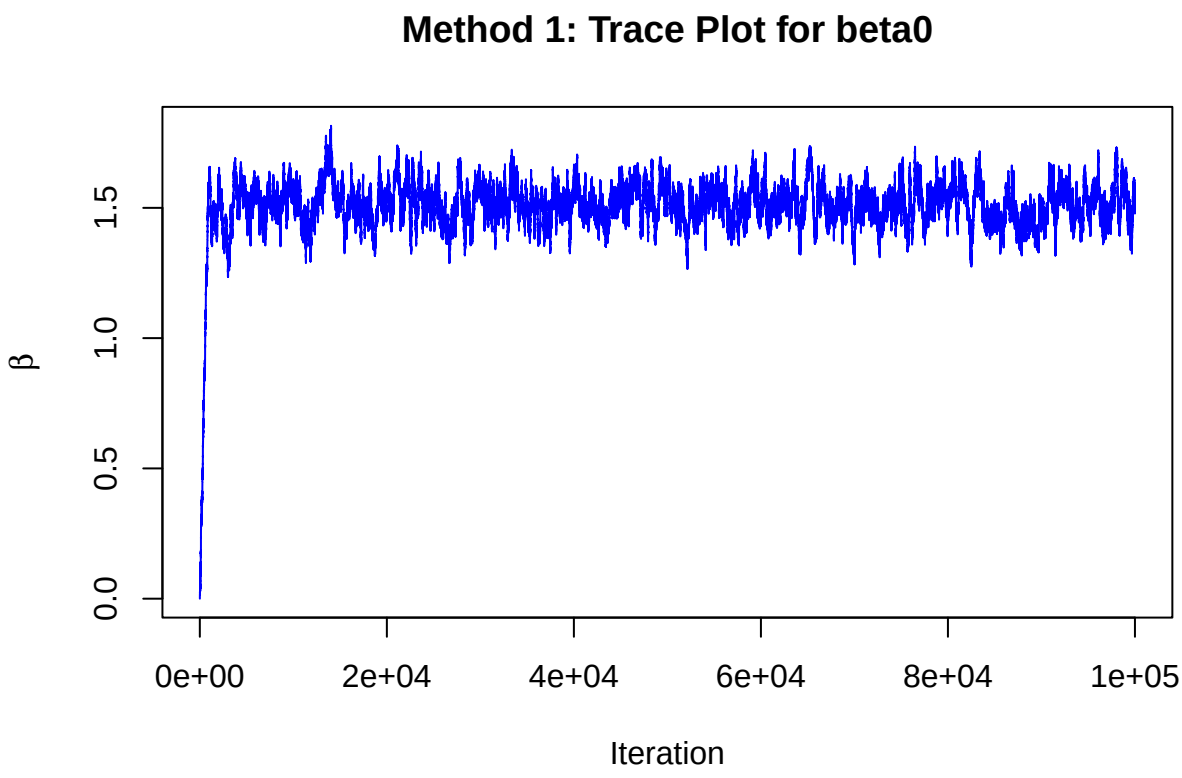
Table 1: Parameter estimates from method 1 (Q1A)

Parameter	Post. Mean	Post SD	Post 95% Lower	Post 95% Upper
beta0	1.5082206	0.1109601	1.3597697	1.6626770
beta1	0.5839797	0.0754553	0.4374636	0.7338309
sigma2	0.9200891	0.1684692	0.7109871	1.1740931

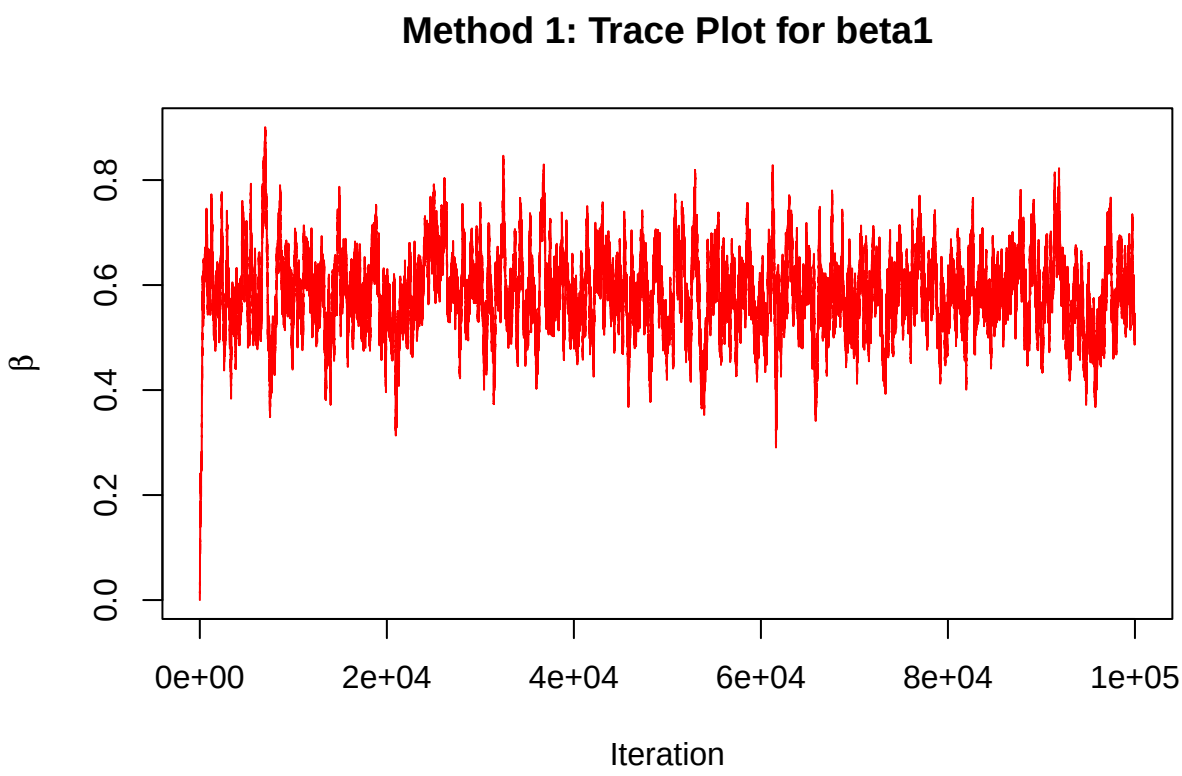
Question 1C-1

For 1A, here are the trace plots denoting convergence:

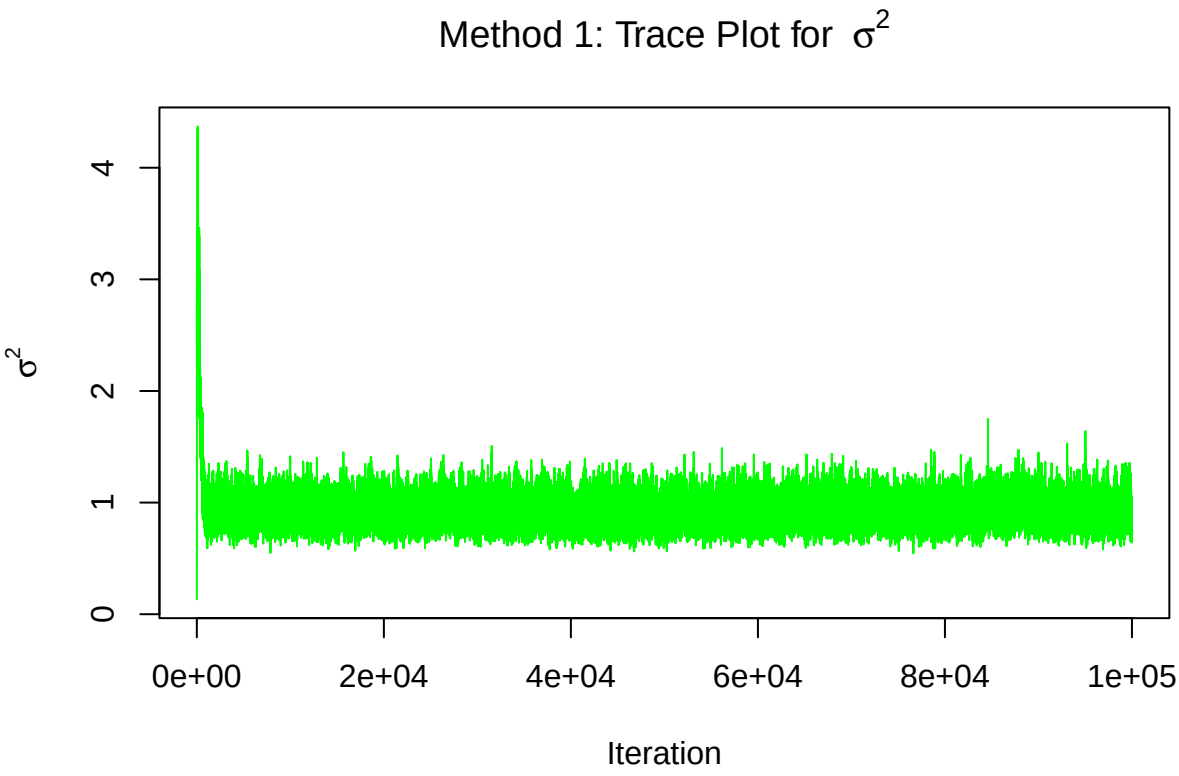
```
# Trace plot for beta0
plot(beta0_samples,
    type = "l", col = "blue", main = "Method 1: Trace Plot for beta0",
    xlab = "Iteration", ylab = expression(beta)
)
```



```r
# Trace plot for beta1
plot(beta1_samples,
    type = "l", col = "red", main = "Method 1: Trace Plot for beta1",
    xlab = "Iteration", ylab = expression(beta)
)
```




```r
# Trace plot for sigma2
plot(sigma2_samples,
    type = "l", col = "green", main = expression("Method 1: Trace Plot for " ~ sigma^2),
    xlab = "Iteration", ylab = expression(sigma^2)
)
```



Note that beyond values of ~2000, all three parameters have stabilized nicely. We can see that they are circling the values we take to be the right values, and in addition, appear to have satisfactorily converged. Note that we would need to discared the first number of iterations (e.g. 2000-5000) before calculating any further diagnostic values of convergence.

Question 1B

```
## NOTE !!!
## much of this will be very similar to the above one, I wrote this one first
## then wrote the above one to try it with the functions (performance test)
library(MASS) # For using multivariate normal functions if needed in extensions
n <- length(y)

# mcmc configuration
mcmc_samples <- 100000
# proposal variances are just the same ones from above, not going to re-state

# define the prior distribution parameters - for convenience
prior_beta_sd <- 100
prior_sigma_shape <- 0.01
prior_sigma_rate <- 0.01

# empty objects to store the values in
beta0 <- numeric(mcmc_samples)
beta1 <- numeric(mcmc_samples)
```

```r
sigma2 <- numeric(mcmc_samples)
# this needs a column for each ob
phi <- matrix(0, nrow = mcmc_samples, ncol = n)

# initial values - just used naive values here
beta0[1] <- 0
beta1[1] <- 0
sigma2[1] <- 1
# this one I pull from N(0, sigma^2)
phi[1, ] <- rnorm(n, 0, sigma2[1]^2)

# want to know the acceptance rates for the three things, these will track
accept_beta0 <- 0
accept_beta1 <- 0
accept_phi <- rep(0, n)

# sample with a big loop -- NOTE THE ORDER IS DIFFERENT FOR THIS ONE (I just
# wrote this one first)
start_b <- Sys.time()
for (i in 2:mcmc_samples) { # first values are already set

    ## STEP 1 ## update phi for each observation j using MH
    for (j in 1:n) {
        # new proposal for phi
        phi_proposal <- rnorm(1, phi[i - 1, j], sqrt(delta_phi))
        acceptance_ratio <- y[j] * phi_proposal -
            exp(beta0[i - 1] + beta1[i - 1] * x[j] + phi_proposal) -
            (phi_proposal^2) /
                (2 * sigma2[i - 1]) -
            (y[j] * phi[i - 1, j] -
                exp(beta0[i - 1] + beta1[i - 1] * x[j] + phi[i - 1, j]) -
                (phi[i - 1, j]^2) / (2 * sigma2[i - 1]))

        # Accept or reject based on acceptance ratio
        if (log(runif(1)) < acceptance_ratio) { # If proposal is accepted
            phi[i, j] <- phi_proposal
            accept_phi[j] <- accept_phi[j] + 1
        } else { # If proposal is rejected
            phi[i, j] <- phi[i - 1, j]
        }
    }

    ## STEP 2 ## the second metropolis
    beta0_proposal <- rnorm(1, beta0[i - 1], sqrt(delta_beta0))
    acceptance_ratio <- sum(
        dpois(y,
            lambda = exp(beta0_proposal + beta1[i - 1] * x + phi[i, ]),
            log = TRUE
        )
    ) +
        dnorm(beta0_proposal, 0, prior_beta_sd, log = TRUE) -
        sum(dpois(y,
            lambda = exp(beta0[i - 1] + beta1[i - 1] * x + phi[i, ]),
```

```
            log = TRUE
        )) -
        dnorm(beta0[i - 1], 0, prior_beta_sd, log = TRUE)

    if (log(runif(1)) < acceptance_ratio) {
        beta0[i] <- beta0_proposal
        accept_beta0 <- accept_beta0 + 1
    } else {
        beta0[i] <- beta0[i - 1]
    }

    ## STEP 3 ## Update beta1 using Metropolis-Hastings
    beta1_proposal <- rnorm(1, beta1[i - 1], sqrt(delta_beta1)) # this is normal
    acceptance_ratio <- sum(
        dpois(y,
            lambda = exp(beta0[i] + beta1_proposal * x + phi[i, ]),
            log = TRUE
        )
    ) +
        dnorm(beta1_proposal, 0, prior_beta_sd, log = TRUE) -
        sum(dpois(y,
            lambda = exp(beta0[i] + beta1[i - 1] * x + phi[i, ]),
            log = TRUE
        )) -
        dnorm(beta1[i - 1], 0, prior_beta_sd, log = TRUE)

    if (log(runif(1)) < acceptance_ratio) {
        beta1[i] <- beta1_proposal
        accept_beta1 <- accept_beta1 + 1
    } else {
        beta1[i] <- beta1[i - 1]
    }

    ## STEP 4 ## direct sample -- not using the function from A
    shape_post <- prior_sigma_shape + n / 2
    rate_post <- prior_sigma_rate + sum(phi[i, ]^2) / 2
    sigma2[i] <- 1 / rgamma(1, shape = shape_post, rate = rate_post)

    if (i == 100000) {
        cat("Iteration:", i, "\n")
        cat("Acceptance rate for beta0:", accept_beta0 / i, "\n")
        cat("Acceptance rate for beta1:", accept_beta1 / i, "\n")
        cat("Average acceptance rate for phi:", mean(accept_phi / i), "\n")
    }
}
```

```
## Iteration: 100000
## Acceptance rate for beta0: 0.26174
## Acceptance rate for beta1: 0.23423
## Average acceptance rate for phi: 0.53258
```

```
end_b <- Sys.time()

# pre-defined function for the summary stats -- used this function cos not
```

```r
# included in performance test
beta0_summary <- summary_statistics(beta0)
beta1_summary <- summary_statistics(beta1)
sigma2_summary <- summary_statistics(sigma2)

# summary table! (excluding ln(lambda_i))
knitr::kable(
    data.frame(
        Parameter = c("beta0", "beta1", "sigma2"),
        `Posterior Mean` = c(
            beta0_summary["mean"],
            beta1_summary["mean"], sigma2_summary["mean"]
        ),
        `Posterior SD` = c(
            beta0_summary["sd"],
            beta1_summary["sd"], sigma2_summary["sd"]
        ),
        `95% Credible Interval Lower` = c(
            beta0_summary[["ci_95_lower"]],
            beta1_summary[["ci_95_lower"]], sigma2_summary[["ci_95_lower"]]
        ),
        `95% Credible Interval Upper` = c(
            beta0_summary[["ci_95_upper"]],
            beta1_summary[["ci_95_upper"]], sigma2_summary[["ci_95_upper"]]
        )
    ),
    caption = "Parameter estimates from method 1 (Q1B)",
    col.names = c(
        "Parameter", "Post. Mean", "Post SD", "Post 95% Lower",
        "Post 95% Upper"
    )
) %>%
    kableExtra::kable_classic()
```

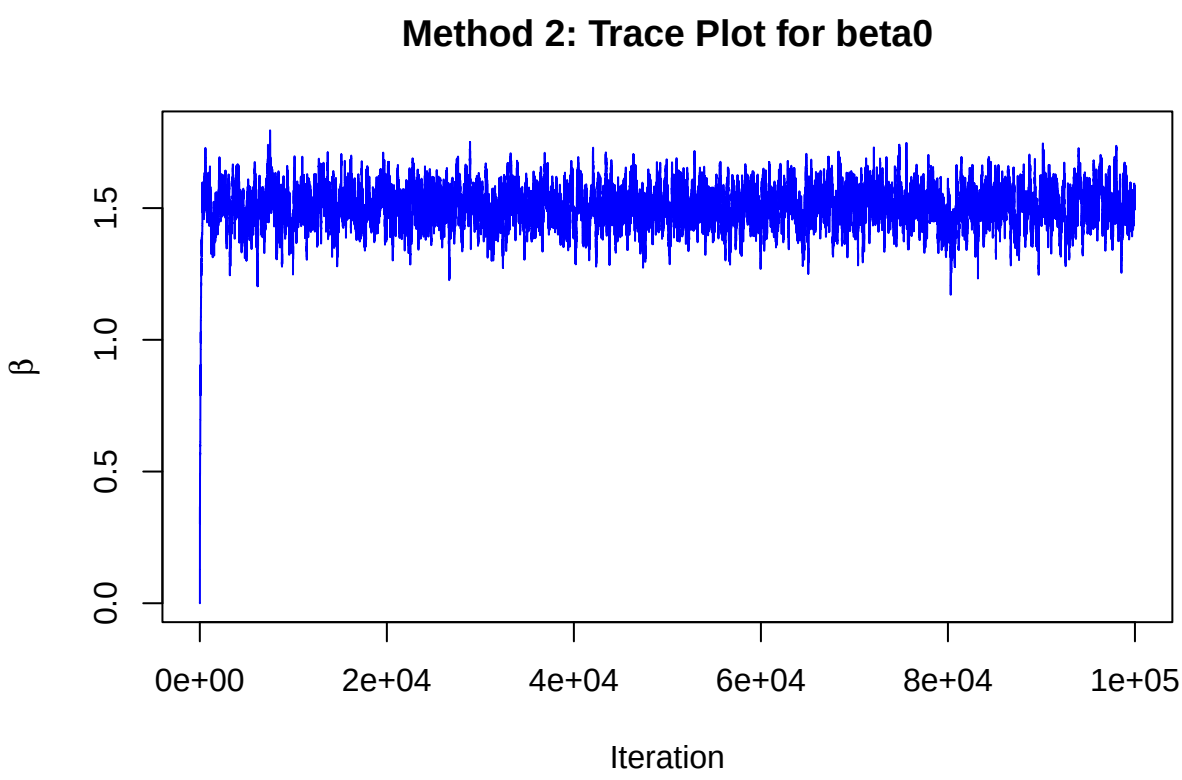
Table 2: Parameter estimates from method 1 (Q1B)

Parameter	Post. Mean	Post SD	Post 95% Lower	Post 95% Upper
beta0	1.5046769	0.0788104	1.3549039	1.6443918
beta1	0.5956956	0.0749656	0.4481277	0.7426872
sigma2	0.9149807	0.1196182	0.7110126	1.1677439

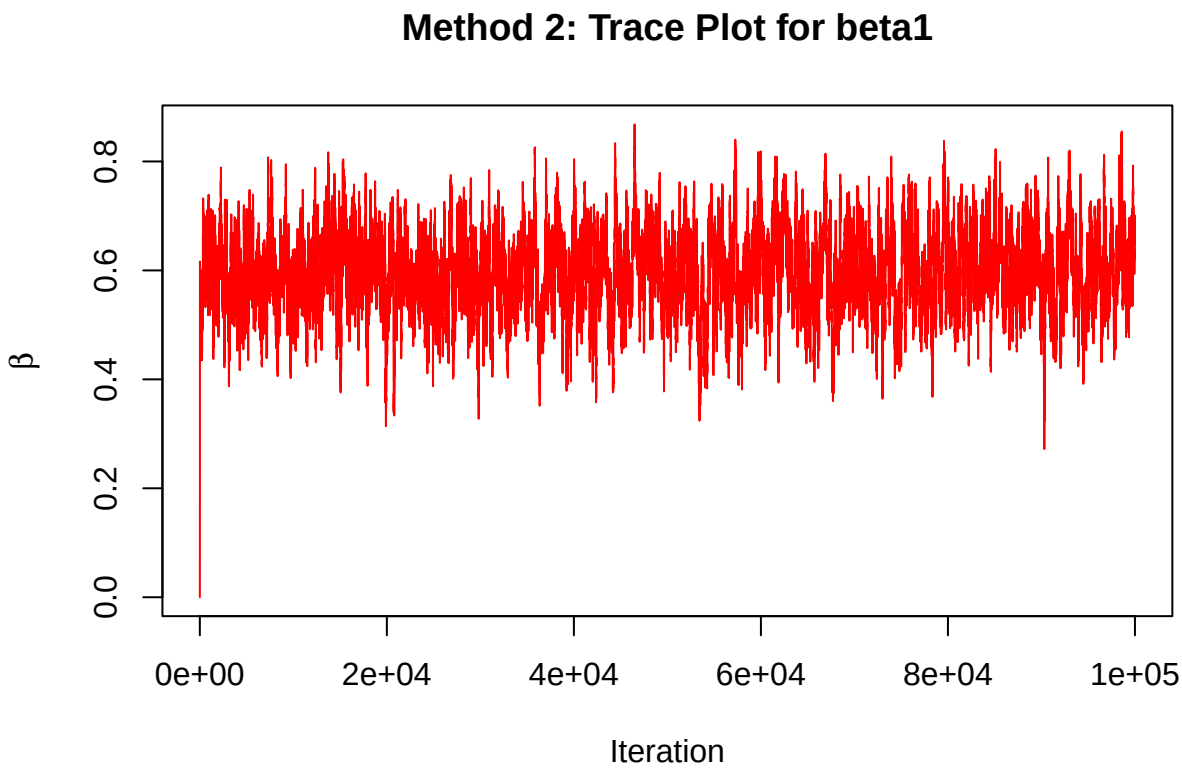
Question 1C-2

For 1B, here are the trace plots denoting convergence:

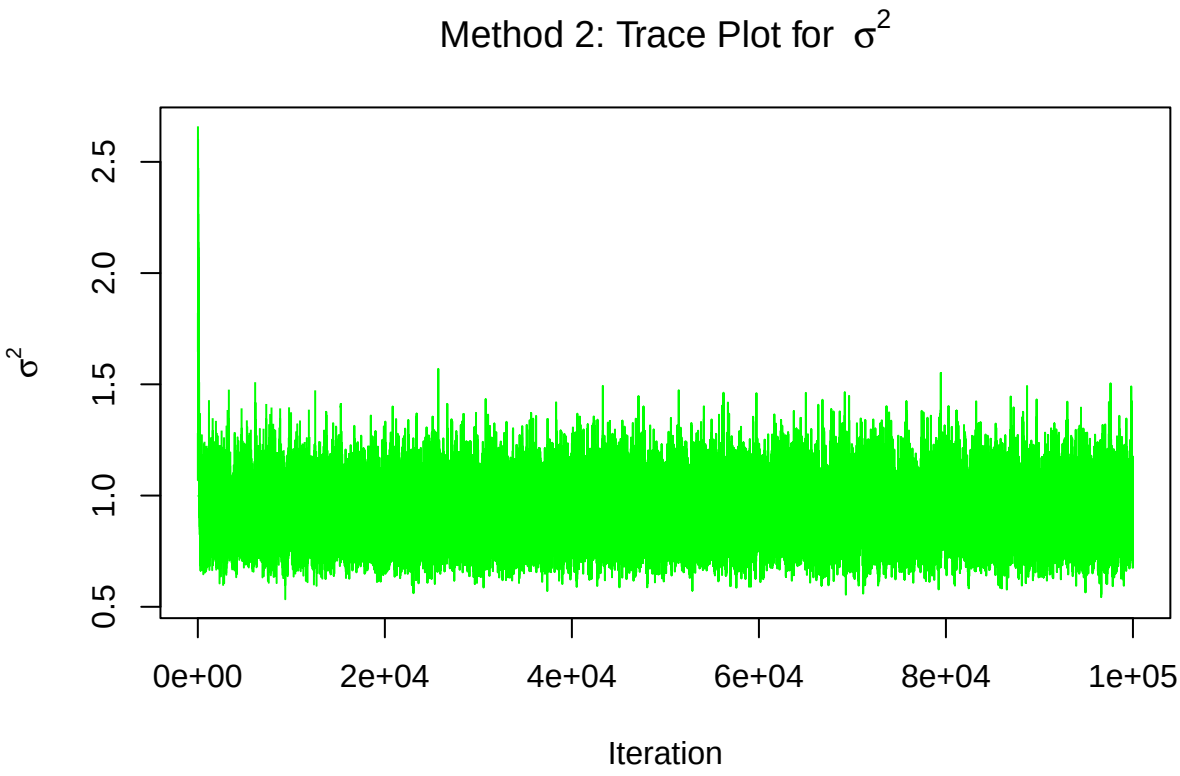
```r
# Trace plot for beta0
plot(beta0,
    type = "l", col = "blue", main = "Method 2: Trace Plot for beta0",
    xlab = "Iteration", ylab = expression(beta)
)
```




```r
# Trace plot for beta1
plot(beta1,
    type = "l", col = "red", main = "Method 2: Trace Plot for beta1",
    xlab = "Iteration", ylab = expression(beta)
)
```




```
# Trace plot for sigma2
plot(sigma2,
    type = "l", col = "green", main = expression("Method 2: Trace Plot for " ~ sigma^2),
    xlab = "Iteration", ylab = expression(sigma^2)
)
```



Similarly to above. Note that beyond values of ~2000, all three parameters have stabilized nicely. We can see that they are circling the values we take to be the right values, and in addition, appear to have satisfactorily converged. Note that we would need to discared the first number of iterations (e.g. 2000-5000) before calculating any further diagnostic values of convergence.

Question 1D

The values between the two fitting methods are quite comparable. They converge to the same parameter combinations, at least for the ones we're interested in. We can see how "well" the methods I've implemented above do in comparison to a software that should in theory generate as-good-as-possible estimates of the values. I show that here with RStan:

```
# Load required packages
library(rstan)
```

```
## Loading required package: StanHeaders
```

```
##
## rstan version 2.32.6 (Stan version 2.32.2)
```

```
## For execution on a local, multicore CPU with excess RAM we recommend calling
## options(mc.cores = parallel::detectCores()).
## To avoid recompilation of unchanged Stan programs, we recommend calling
## rstan_options(auto_write = TRUE)
## For within-chain threading using `reduce_sum()` or `map_rect()` Stan functions,
## change `threads_per_chain` option:
## rstan_options(threads_per_chain = 1)
```

```
##
## Attaching package: 'rstan'
## The following object is masked from 'package:magrittr':
##
##     extract
```

```r
# Prepare data list for Stan
stan_data <- list(
    n = n,
    x = x,
    y = y
)

# Define the Stan model as a string
stan_model_code <- "
data {
  int<lower=0> n;            // number of observations
  vector[n] x;               // predictor variable
  int<lower=0> y[n];         // response variable (counts)
}

parameters {
  real beta0;                // intercept
  real beta1;                // slope
  real<lower=0> sigma2;      // variance of phi
  vector[n] phi;             // random effects
}

model {
  // Priors
  beta0 ~ normal(0, 100);
  beta1 ~ normal(0, 100);
  sigma2 ~ inv_gamma(0.01, 0.01);

  // Likelihood
  for (i in 1:n) {
    // The Poisson model with random effects
    y[i] ~ poisson(exp(beta0 + beta1 * x[i] + phi[i]));
    phi[i] ~ normal(0, sqrt(sigma2));  // Random effects prior
  }
}
"

# Compile the Stan model
stan_model <- stan_model(model_code = stan_model_code)

# Fit the model using Stan
fit <- sampling(stan_model,
    data = stan_data,
    iter = 10000, chains = 4,
    show_messages = FALSE,
    refresh = 0
)
```

```r
# Extract posterior samples
posterior_samples <- extract(fit)

# Summary of the results
beta0_samples_stan <- posterior_samples$beta0
beta1_samples_stan <- posterior_samples$beta1
sigma2_samples_stan <- posterior_samples$sigma2

beta0_summary_stan <- summary_statistics(beta0_samples_stan)
beta1_summary_stan <- summary_statistics(beta1_samples_stan)
sigma2_summary_stan <- summary_statistics(sigma2_samples_stan)

# Create summary table
knitr::kable(
    data.frame(
        Parameter = c("beta0", "beta1", "sigma2"),
        Posterior_Mean = c(
            beta0_summary_stan["mean"],
            beta1_summary_stan["mean"],
            sigma2_summary_stan["mean"]
        ),
        Posterior_SD = c(
            beta0_summary_stan["sd"],
            beta1_summary_stan["sd"],
            sigma2_summary_stan["sd"]
        ),
        `95%_Credible_Interval_Lower` = c(
            beta0_summary_stan["ci_95_lower"],
            beta1_summary_stan["ci_95_lower"],
            sigma2_summary_stan["ci_95_lower"]
        ),
        `95%_Credible_Interval_Upper` = c(
            beta0_summary_stan["ci_95_upper"],
            beta1_summary_stan["ci_95_upper"],
            sigma2_summary_stan["ci_95_upper"]
        )
    ),
    caption = "Parameter estimates from Stan for comparison to our own sampler",
    col.names = c(
        "Parameter", "Post. Mean", "Post SD", "Post 95% Lower",
        "Post 95% Upper"
    )
) %>% kableExtra::kable_classic()
```

Table 3: Parameter estimates from Stan for comparison to our own sampler

Parameter	Post. Mean	Post SD	Post 95% Lower	Post 95% Upper
beta0	1.5088861	0.0717560	1.3667981	1.6477604
beta1	0.5938131	0.0733070	0.4508042	0.7377056
sigma2	0.9131501	0.1164576	0.7092126	1.1601654

We can see that both of the method 1 and method 2 samplers above did a comparable job at reproducing the

results from Stan. We can assume that there will always be small differences (both due to just the stochastic nature of the samplers as well as due to the algorithmic differences). We can also look at the performance:

For method a, the runtime on 100,000 iterations was:

```
print(end_a - start_a)
```

```
## Time difference of 1.415015 mins
```

And on method b, the runtime on the same number of iterations was:

```
print(end_b - start_b)
```

```
## Time difference of 50.03883 secs
```

Method 2 was slightly quicker, and I think was made slightly easier by not having to worry about the log-scale on λ, making it more appealing.

Overall, the esimates given by the two methods however did differ on the order of 2 decimal places, which, depending on the parameters, may not be insignificant. Further thought on which approach better captures the true value would be needed to decide if the speed of option 2 is worth using it.