```r
#clear global environment
rm(list=ls())
```

```r
#install.packages("doParallel", "foreach")
#install.packages("coda")
```

```r
#define simulated dataset
# Y=a+bx+eps
set.seed(4)
n<-100
x <- rnorm(n, mean = 5, sd = 2)
  alpha_true <- 2                  # True intercept
  beta_true <- 3                   # True slope
  sigma <- 4
  #add white noise to linear relationship
Y <- alpha_true + beta_true * x+rnorm(n, sd = sigma)
X<-matrix(c(rep(1,n),x), nrow=n,ncol=2)
plot(x,Y)
```

```r
bayes.SLR.gibbs <- function(n = 100, alpha_prior_mean = 0, alpha_prior_sd = 1,
                            beta_prior_mean = 0, beta_prior_sd = 1,
                            iter = 8000, seed = 4, burnin = 4000) {

  set.seed(seed)  # Set seed for reproducibility

  # Simulate data
  x <- rnorm(n, mean = 5, sd = 2)  # Predictor variable
  alpha_true <- 2                  # True intercept
  beta_true <- 3                   # True slope
  sigma <- 4                       # Known standard deviation
  y <- alpha_true + beta_true * x + rnorm(n, sd = sigma)  # Response variable

  # Initialize storage for samples - only need space for post-burnin samples
  alpha_samples <- numeric(iter)
  beta_samples <- numeric(iter)

  # Initial values for alpha and beta
  # Starting closer to true values can reduce burnin needed
  a0 <- rnorm(1, mean = mean(y) - mean(x), sd = 1)
  b0 <- rnorm(1, mean = cov(x,y)/var(x), sd = 1)

  # Gibbs sampling - use separate counter for storing samples
  sample_idx <- 0

  # Run the sampler for burnin + iter iterations
  for (i in 1:(burnin + iter)) {
    # Update alpha
    alpha_posterior_var <- 1 / (1 / alpha_prior_sd^2 + n / sigma^2)
    alpha_posterior_mean <- alpha_posterior_var * (
      alpha_prior_mean / alpha_prior_sd^2 + sum(y - b0 * x) / sigma^2
    )
    a0 <- rnorm(1, mean = alpha_posterior_mean, sd = sqrt(alpha_posterior_var))
```

```
    # Update beta
    beta_posterior_var <- 1 / (1 / beta_prior_sd^2 + sum(x^2) / sigma^2)
    beta_posterior_mean <- beta_posterior_var * (
      beta_prior_mean / beta_prior_sd^2 + sum((y - a0) * x) / sigma^2
    )
    b0 <- rnorm(1, mean = beta_posterior_mean, sd = sqrt(beta_posterior_var))

    # Store samples only after burnin
    if (i > burnin) {
      sample_idx <- sample_idx + 1
      alpha_samples[sample_idx] <- a0
      beta_samples[sample_idx] <- b0
    }
  }

  # Summarize posterior samples
  alpha_mean <- mean(alpha_samples)
  beta_mean <- mean(beta_samples)

  # Add true values to the output for comparison
  return(list(alpha_samples = alpha_samples,
              beta_samples = beta_samples,
              alpha_mean = alpha_mean,
              beta_mean = beta_mean,
              alpha_true = alpha_true,
              beta_true = beta_true,
              data = list(x = x, y = y, sigma = sigma)))
}
```

```
eps<-0.01
#relatively un-informative prior
well.spec<-bayes.SLR.gibbs(alpha_prior_mean = 0, alpha_prior_sd = sqrt(1.8),
                           beta_prior_mean = 0, beta_prior_sd = sqrt(1.8), iter=8000)
#miss-specified prior
miss.spec<-bayes.SLR.gibbs(alpha_prior_mean = 0, alpha_prior_sd = sqrt(eps),
                           beta_prior_mean = 0, beta_prior_sd = sqrt(eps), iter=8000)
well.spec$alpha_mean
well.spec$beta_mean
miss.spec$alpha_mean
miss.spec$beta_mean
```

```
#function to calculate parameters for analytical posterior distribution
# inputs are prior mean and prior covariance matrix
posterior.params <- function(pri.mu,pri.sigma, seed=4, n=100){
  set.seed(seed)
x <- rnorm(n, mean = 5, sd = 2)
  alpha_true <- 2                  # True intercept
  beta_true <- 3                   # True slope
  sigma <- 4
Y <- alpha_true + beta_true * x+rnorm(n, sd = sigma)
X<-matrix(c(rep(1,n),x), nrow=n,ncol=2)
z<-(1/sigma^2)*t(X)%*%X+solve(pri.sigma)
theta_mu<-solve(z)%*%(solve(pri.sigma)%*%pri.mu + (1/sigma^2)*t(X)%*%Y)
```

```
list(theta_mu=theta_mu, theta_var_mat=solve(z))
  }

#create very small variance
well.spec.post.mu<-posterior.params(pri.mu = c(0,0), pri.sigma = 1.8*diag(2))$theta_mu
well.spec.post.varmat<-posterior.params(pri.mu = c(0,0), pri.sigma = 1.8*diag(2))$theta_var_mat
well.spec.post.mu
well.spec.post.varmat
miss.spec.post.mu<-posterior.params(pri.mu = c(0,0), pri.sigma = eps*diag(2))$theta_mu
miss.spec.post.varmat<-posterior.params(pri.mu = c(0,0), pri.sigma = eps*diag(2))$theta_var_mat
miss.spec.post.mu
miss.spec.post.varmat

#Well Specified case
# Plot the posterior distributions
hist(well.spec$alpha_samples, breaks=100, main = "Posterior of alpha", xlab = "alpha", freq = F)
# Create a sequence of x values for the normal distribution curve
a_x_values <- seq(min(well.spec$alpha_samples), max(well.spec$alpha_samples), length = 100)

# Calculate the marginal density for those x values
alpha_normal_density <- dnorm(a_x_values, mean = well.spec.post.mu[1], sd = sqrt(well.spec.post.varmat[

# Add the normal distribution density curve to the histogram
lines(a_x_values, alpha_normal_density, col = "red", lwd = 2)
# Add a vertical line for the true alpha value
abline(v = 2, col = "black", lwd = 2, lty = 2)  # True alpha value

# Plot the posterior distributions
hist(well.spec$beta_samples, breaks=100, main = "Posterior of beta", xlab = "beta", freq = F)
# Create a sequence of x values for the normal distribution curve
b_x_values <- seq(min(well.spec$beta_samples), max(well.spec$beta_samples), length = 100)

# Calculate marginal density for those x values
beta_normal_density <- dnorm(b_x_values, mean = well.spec.post.mu[2], sd = sqrt(well.spec.post.varmat[2

# Add the normal distribution density curve to the histogram
lines(b_x_values, beta_normal_density, col = "blue", lwd = 2)
abline(v = 3, col = "black", lwd = 2, lty = 2)  # True beta value

#Miss Specified case
# Plot the posterior distributions
hist(miss.spec$alpha_samples, breaks=100, main = "Posterior of alpha(miss-specified)", xlab = "alpha",
# Create a sequence of x values for the normal distribution curve
a_x_values <- seq(-1, 2.3, length = 1000)

# Calculate the marginal density for those x values
alpha_normal_density <- dnorm(a_x_values, mean = miss.spec.post.mu[1], sd = sqrt(miss.spec.post.varmat[

# Add the normal distribution density curve to the histogram
lines(a_x_values, alpha_normal_density, col = "red", lwd = 2)
# Add a vertical line for the true alpha value
abline(v = 2, col = "black", lwd = 2, lty = 2)  # True alpha value
```

```r
# Plot the posterior distributions
hist(miss.spec$beta_samples, breaks=50, main = "Posterior of beta(miss-specified)", xlab = "beta", freq
# Create a sequence of x values for the normal distribution curve
b_x_values <- seq(1, 3, length = 1000)

# Calculate marginal density for those x values
beta_normal_density <- dnorm(b_x_values, mean = miss.spec.post.mu[2], sd = sqrt(miss.spec.post.varmat[2

# Add the normal distribution density curve to the histogram
lines(b_x_values, beta_normal_density, col = "blue", lwd = 2)
# Add a vertical line for the true beta value
abline(v = 3, col = "black", lwd = 2, lty = 2)  # True beta value
```

```r
#put parameters into single vector
well.theta<-matrix(c(well.spec$alpha_samples,well.spec$beta_samples), nrow=2, byrow = T)
miss.theta<-matrix(c(miss.spec$alpha_samples,miss.spec$beta_samples), nrow=2, byrow = T)
```

```r
#computes posterior predictive via MCMC, AOI importance sampling
posterior_predictive.v2 <- function(Y, X, y, x_new, theta, sigma) {
  # Compute weights(log scale)
  log_w <- apply(theta, 2, function(th) dnorm(y, mean = x_new %*% th, sd = sigma, log=TRUE))
  max_log_w <- max(log_w)
  log_sum_exp <- max_log_w + log(sum(exp(log_w - max_log_w)))
  normalized_weights <- exp(log_w - log_sum_exp)

  # Get dimensions
  n_eval <- length(Y)
  n_samples <- ncol(theta)

  # Calculate all means for all evaluation points and samples
  all_means <- X %*% theta

  # Create an array that repeats Y for each sample
  # Create a n_eval × n_samples matrix where each row is filled with one Y value
  Y_mat <- matrix(Y, nrow=n_eval, ncol=n_samples)

  # Calculate all densities at once
  all_densities <- dnorm(Y_mat, mean=all_means, sd=sigma)

  # Apply weights and sum rows
  result <- rowSums(all_densities * matrix(normalized_weights,
                                           nrow=n_eval,
                                           ncol=n_samples,
                                           byrow=TRUE))

  return(result)
}
```

```r
library(foreach)
library(doParallel)
#Function to compute upper and lower bounds of conformal interval
#uses parallel computation to loop over new x values
conformal_bounds_single_parallel <- function(Y, X, conf_x_new_mat, y_grid, theta, sigma, alpha) {
```

```
  num_cores <- detectCores() - 1  # Use all but one core
  cl <- makeCluster(num_cores)
  registerDoParallel(cl)

  # Export required functions and variables
  clusterExport(cl, varlist = c("posterior_predictive.v2", "Y", "X", "y_grid", "theta", "sigma", "alpha

  conf_range <- foreach(i = 1:nrow(conf_x_new_mat), .combine = rbind, .packages = c("foreach")) %dopar%
    x_new_i <- conf_x_new_mat[i, , drop = FALSE]  # Extract the current `x_new`

    # Compute conformal prediction set directly within the same loop
    accepted_y <- foreach(l = 1:length(y_grid), .combine = c) %do% {
      # Compute conformity scores on dataset
      sig_1_to_n <- posterior_predictive.v2(Y = Y, X = X, y = y_grid[l], x_new = x_new_i, theta = theta

      # Conformity score on test point
      sig_n_plus_one <- posterior_predictive.v2(Y = y_grid[l], X = x_new_i, y = y_grid[l], x_new = x_ne

      # Adjusted quantile calculation
      n <- length(Y)
      pi <- (length(which(sig_1_to_n <= sig_n_plus_one)) + 1) / (n + 1)

      # Only return y_grid[l] if it meets the condition
      if (pi > alpha) {
        y_grid[l]
      } else {
        NULL
      }
    }

    # Compute lower and upper bounds
    bounds <- range(accepted_y)
    return(bounds)
  }

  stopCluster(cl)

  return(conf_range)  # Matrix where each row is [lower, upper] for an `x_new`
}
```

```
#miss-spec
library("doParallel")
library("foreach")
#create a y grid to loop over
#ny is grid fineness for y
ny <-150
nx<- 100
y_grid <- seq(-10,44, length.out = ny)
x_grid <- seq(min(x), max(x), length.out = nx )
#turn x-grid into design matrix
x_new_mat<-matrix(c(rep(1,nx),x_grid ), nrow=nx)
#this involves (nx)x8000x(100+ny)=200,000,000 likelihood evals
miss_conf_range <- conformal_bounds_single_parallel(Y, X, conf_x_new_mat=x_new_mat, y_grid, theta=miss.
```

```r
#well-spec
library("doParallel")
library("foreach")
#create a y grid to loop over
#ny is grid fineness for y
ny <-150
nx<- 100
y_grid <- seq(0,40, length.out = ny)
x_grid <- seq(min(x), max(x), length.out = nx )
#turn x-grid into design matrix
x_new_mat<-matrix(c(rep(1,nx),x_grid ), nrow=nx)
#this involves (nx)x8000x(100+ny)=200,000,000 likelihood evals
well_conf_range <- conformal_bounds_single_parallel(Y, X, conf_x_new_mat=x_new_mat, y_grid, theta=well.
```

```r
alpha<-0.2
# Create a fine grid of x values to approximate the credible bands
cred_x_grid <- seq(min(x), max(x), length.out = 100)
#create design matrix for x_new
cred_x_new_mat<-matrix(c(rep(1,100),cred_x_grid), nrow=100)
#Use quantiles to get credible intervals, using analytically derived density of the posterior predictiv
credible_lower <- function(x_new,mu,varmat) {
qnorm(alpha / 2, mean = x_new%*%mu, sd = sqrt(sigma^2+x_new%*%varmat%*%t(x_new)))
}

credible_upper <- function(x_new,mu,varmat) {
qnorm(1 - alpha / 2, mean = x_new%*%mu, sd = sqrt(sigma^2+x_new%*%varmat%*%t(x_new)))
}
```

```r
# Calculate the credible interval bounds over the grid(drop preserves dimension of extracted vector)
m.cred.lower <- sapply(1:nrow(cred_x_new_mat), function(i) credible_lower(cred_x_new_mat[i, , drop=F], m
miss.spec.post.varmat))
m.cred.upper <- sapply(1:nrow(cred_x_new_mat), function(i) credible_upper(cred_x_new_mat[i, , drop=F], m
miss.spec.post.varmat))
w.cred.lower <- sapply(1:nrow(cred_x_new_mat), function(i) credible_lower(cred_x_new_mat[i, , drop=F], w
well.spec.post.varmat))
w.cred.upper <- sapply(1:nrow(cred_x_new_mat), function(i) credible_upper(cred_x_new_mat[i, , drop=F], w
well.spec.post.varmat))

#debugging
#credible_lower(cred_x_new_mat[100, , drop=F], miss.spec.post.mu,
#miss.spec.post.varmat)
```

```r
# Extract lower and upper bounds
m.lower <- miss_conf_range[, 1]
m.upper <- miss_conf_range[, 2]

# Plot observed data
plot(X[, 2], Y, pch = 16, col = "black", xlab = "X", ylab = "Y",
     main = "Effect of Prior Misspecification", ylim = c(-10,40))

# Fill the confidence region (blue)
polygon(c(x_grid, rev(x_grid)), c(m.lower, rev(m.upper)), col = rgb(0, 0, 1, 0.3), border = NA)
```

```r
# Fill the region between red dotted lines
polygon(c(cred_x_grid, rev(cred_x_grid)), c(m.cred.lower, rev(m.cred.upper)),
        col = rgb(1, 0, 0, 0.2), border = NA)

# Plot smooth upper and lower bounds
lines(x_grid, m.lower, col = "blue", lwd = 2)
lines(x_grid, m.upper, col = "blue", lwd = 2)
lines(cred_x_grid, m.cred.lower, col = "red", lwd = 2, lty = 2)  # Lower bound line
lines(cred_x_grid, m.cred.upper, col = "red", lwd = 2, lty = 2)  # Upper bound line

# Add the least squares line
lm_fit <- lm(Y ~ X[, 2])
abline(lm_fit, col = "darkgreen", lwd = 2)

# Simplified legend with just the two bands
legend("topleft",
       legend = c("Conformal Band", "Credible Band"),
       lty = c(1, 2),
       lwd = c(2, 2),
       col = c("blue", "red"),
       fill = c(rgb(0, 0, 1, 0.3), rgb(1, 0, 0, 0.2)),
       border = NA,
       bg = "white")
```

```r
well_conf_range
```

```r
# Extract lower and upper bounds
w.lower <- well_conf_range[, 1]
w.upper <- well_conf_range[, 2]

# Plot observed data
plot(X[, 2], Y, pch = 16, col = "black", xlab = "X", ylab = "Y",
     main = "Well Specified Case", ylim = c(-3,40))

# Fill the confidence region (blue)
polygon(c(x_grid, rev(x_grid)), c(w.lower, rev(w.upper)), col = rgb(0, 0, 1, 0.3), border = NA)

# Fill the region between red dotted lines
polygon(c(cred_x_grid, rev(cred_x_grid)), c(w.cred.lower, rev(w.cred.upper)),
        col = rgb(1, 0, 0, 0.2), border = NA)

# Plot smooth upper and lower bounds
lines(x_grid, w.lower, col = "blue", lwd = 2)
lines(x_grid, w.upper, col = "blue", lwd = 2)
lines(cred_x_grid, w.cred.lower, col = "red", lwd = 2, lty = 2)  # Lower bound line
lines(cred_x_grid, w.cred.upper, col = "red", lwd = 2, lty = 2)  # Upper bound line

# Add the least squares line
lm_fit <- lm(Y ~ X[, 2])
abline(lm_fit, col = "darkgreen", lwd = 2)

# Simplified legend with just the two bands
legend("topleft",
```

```
       legend = c("Conformal Band", "Credible Band"),
       lty = c(1, 2),
       lwd = c(2, 2),
       col = c("blue", "red"),
       fill = c(rgb(0, 0, 1, 0.3), rgb(1, 0, 0, 0.2)),
       border = NA,
       bg = "white")
```

```
# Function to calculate ESS (Effective Sample Size) from MCMC chains
compute_min_ess <- function(theta) {
  # theta should be a matrix with parameters in rows and samples in columns

  if (requireNamespace("coda", quietly = TRUE)) {
    library(coda)
    # Convert to mcmc object (coda expects samples in rows, parameters in columns)
    theta_mcmc <- as.mcmc(t(theta))
    # Calculate ESS for each parameter
    ess_values <- effectiveSize(theta_mcmc)
    # Return the minimum ESS
    return(min(ess_values))
  } }
compute_min_ess(miss.theta)
compute_min_ess(well.theta)
```

```
# Function to compute and plot ESS with conformal bounds
#usual inputs, select up to 4 alpha values
#mcmc_ess is for scaling plot by min(ESS)/T
#linelen is a plotting parameter to select length to wich conformla bound will be extended vertically
plot_ess_with_bounds <- function(Y, X, y_grid, x_new, theta, sigma,
                                 mcmc_ess = NULL,
                                 alpha_values = c(0.1, 0.2),
                                 title, linelen) {

  require(doParallel)
  require(foreach)

  n_grid <- length(y_grid)
  ess_values <- numeric(n_grid)
  n_samples <- ncol(theta)

  # Compute ESS for each value of y in the grid
  for (i in 1:n_grid) {
    y <- y_grid[i]

    # Compute weights (still using log scale for numerical stability)
    log_w <- apply(theta, 2, function(th) dnorm(y, mean = x_new %*% th, sd = sigma, log=TRUE))
    max_log_w <- max(log_w)
    log_sum_exp <- max_log_w + log(sum(exp(log_w - max_log_w)))
    normalized_weights <- exp(log_w - log_sum_exp)

    # Calculate ESS using formula 1/sum(w_i^2) for normalized weights
    ess_values[i] <- 1 / sum(normalized_weights^2)
  }
```

```r
 # Scale ESS values
 if (!is.null(mcmc_ess)) {
   ess_values <- ess_values * (mcmc_ess / n_samples)
 }

 # Create plot
 plot(y_grid, ess_values, type="l", lwd=2,
      xlab="y", ylab="Effective Sample Size",
      main=title)

   # Format x_new for conformal_bounds_single_parallel
   conf_x_new_mat <- x_new

   # Define colors for different alpha values
   alpha_colors <- c("red", "blue", "green", "purple", "orange")[1:length(alpha_values)]

   # Compute and plot bounds for each alpha
   for (i in 1:length(alpha_values)) {
     alpha <- alpha_values[i]

     # Compute conformal bounds
     conf_bounds <- conformal_bounds_single_parallel(
       Y = Y, X = X,
       conf_x_new_mat = conf_x_new_mat,
       y_grid = y_grid,
       theta = theta,
       sigma = sigma,
       alpha = alpha
     )

     # Extract lower and upper bounds
     lower_bound <- conf_bounds[1]
     upper_bound <- conf_bounds[2]
# Find y-positions for segments near the ESS curve
   # Find the closest y_grid points to our bounds
   lower_idx <- which.min(abs(y_grid - lower_bound))
   upper_idx <- which.min(abs(y_grid - upper_bound))

   # Get the ESS values at these points
   lower_ess <- ess_values[lower_idx]
   upper_ess <- ess_values[upper_idx]

   # Add localized vertical line segments near the ESS curve
   segments(
     x0 = lower_bound, y0 = lower_ess-linelen,
     x1 = lower_bound, y1 = lower_ess+linelen,  # Extend slightly above the curve
     col = alpha_colors[i], lwd = 2
   )
   segments(
     x0 = upper_bound, y0 = upper_ess-linelen,
     x1 = upper_bound, y1 = upper_ess+linelen,  # Extend slightly above the curve
     col = alpha_colors[i], lwd = 2
   )
```

```
  }

  # Add legend
  legend("topright",
         legend = paste(" \\alpha =", alpha_values),
         col = alpha_colors,
         lty = 1,
         lwd = 2,
         cex = 0.8)

}
```

```
#y_grid <- seq(-6, 6, length.out=100)
 plot_ess_with_bounds(Y, X,y_grid=seq(-100, 125, length.out=100), x_new=x_new_mat[50, , drop=F], theta=
```

```
 plot_ess_with_bounds(Y,X, y_grid=seq(-100, 125, length.out=100), x_new=x_new_mat[50, , drop=F], theta=
```