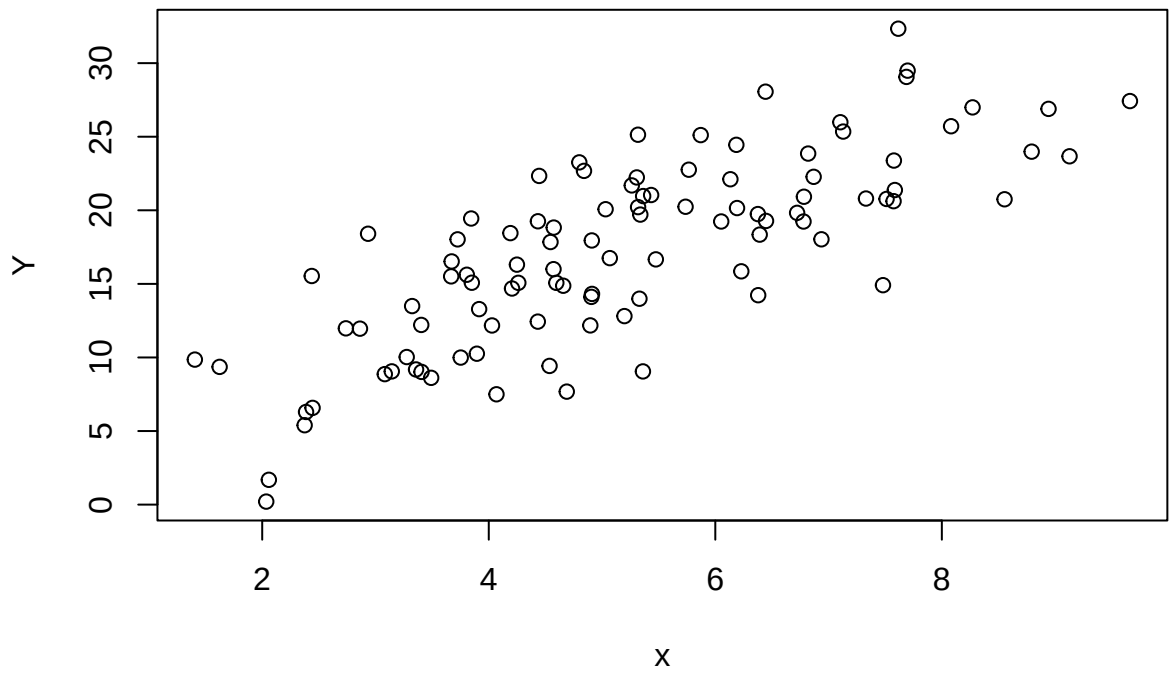
```r
#clear global environment
rm(list=ls())
```

```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
```

```r
# 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
```

```
## [1] 2.028763
```

```
well.spec$beta_mean
```

```
## [1] 2.900992
```

```
miss.spec$alpha_mean
```

```
## [1] 0.377963
```

```
miss.spec$beta_mean
```

```
## [1] 2.088636
```

```
#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
```

```
##          [,1]
## [1,] 2.008330
## [2,] 2.905397
```

```
well.spec.post.varmat
```

```
##            [,1]        [,2]
## [1,]  0.7970637 -0.13631948
## [2,] -0.1363195  0.02858374
```

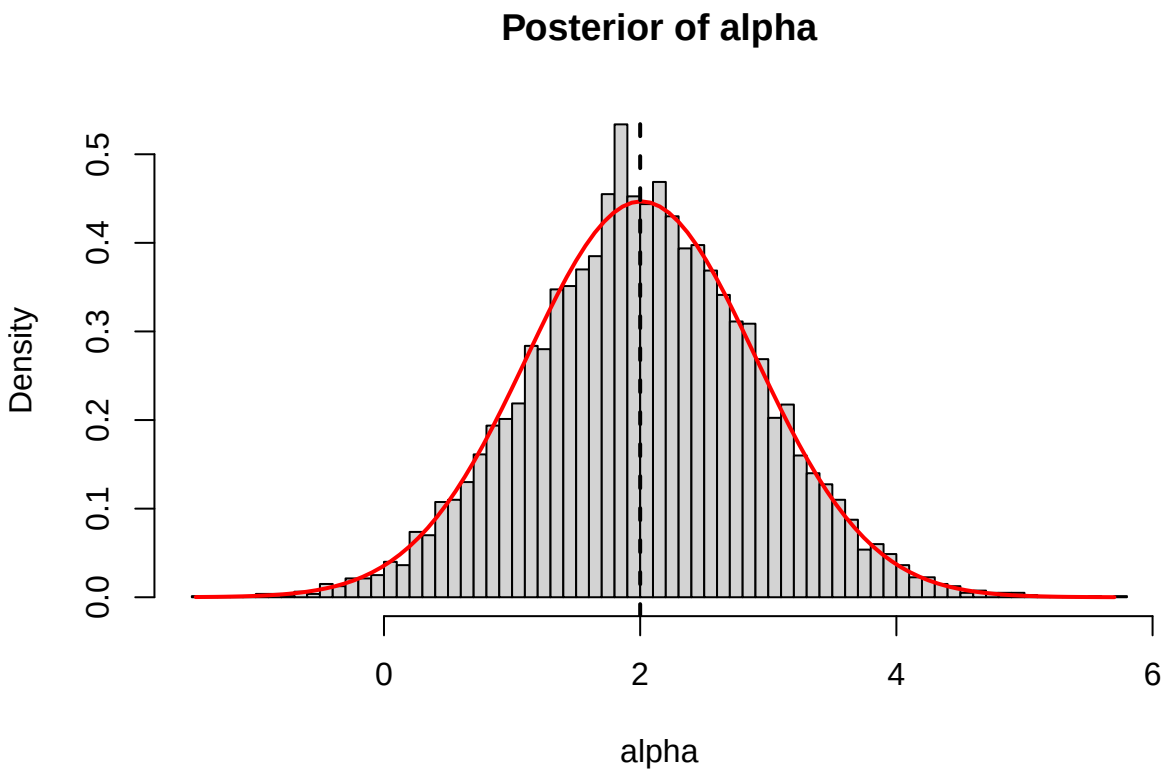
```
miss.spec.post.mu<-posterior.params(pri.mu = c(0,0), pri.sigma = eps*diag(2))$theta_mu
miss.spec.post.varmat<-posterior.params(pri.mu = c(0,0), pri.sigma = eps*diag(2))$theta_var_mat
miss.spec.post.mu
```

```
##          [,1]
## [1,] 0.377910
## [2,] 2.089379
```

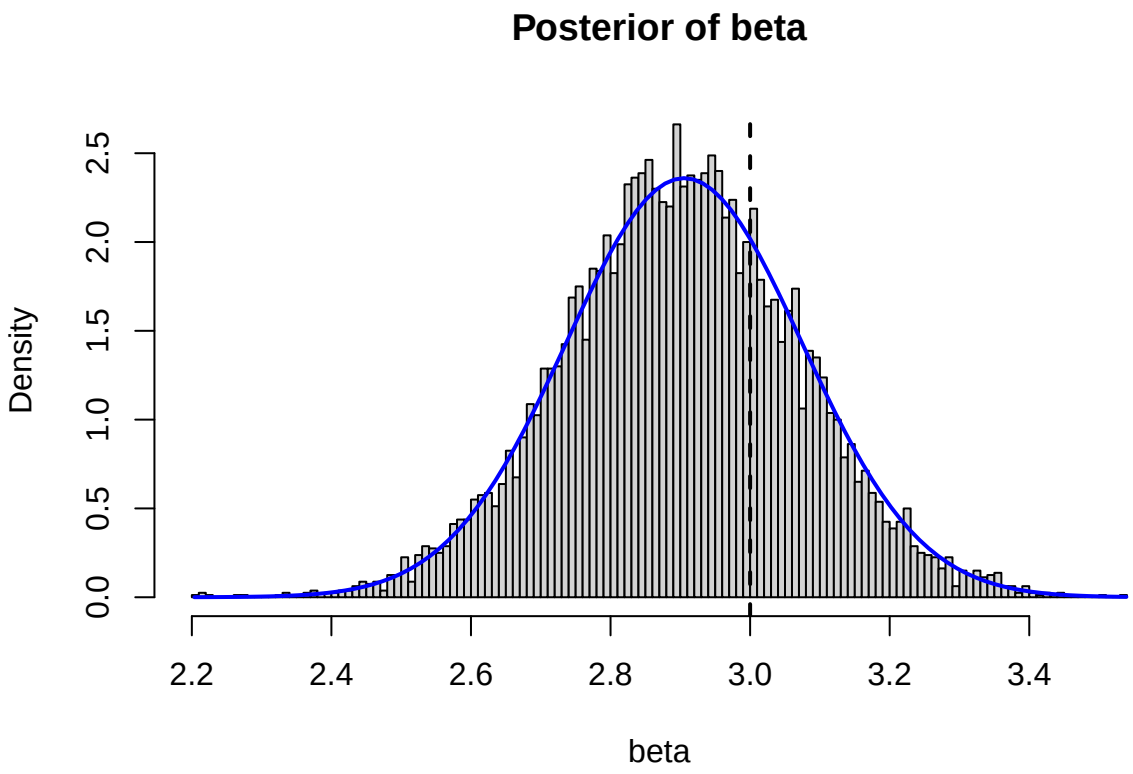
```
miss.spec.post.varmat
```

```
##              [,1]         [,2]
## [1,]  0.009745859 -0.001093695
## [2,] -0.001093695  0.003580325
```

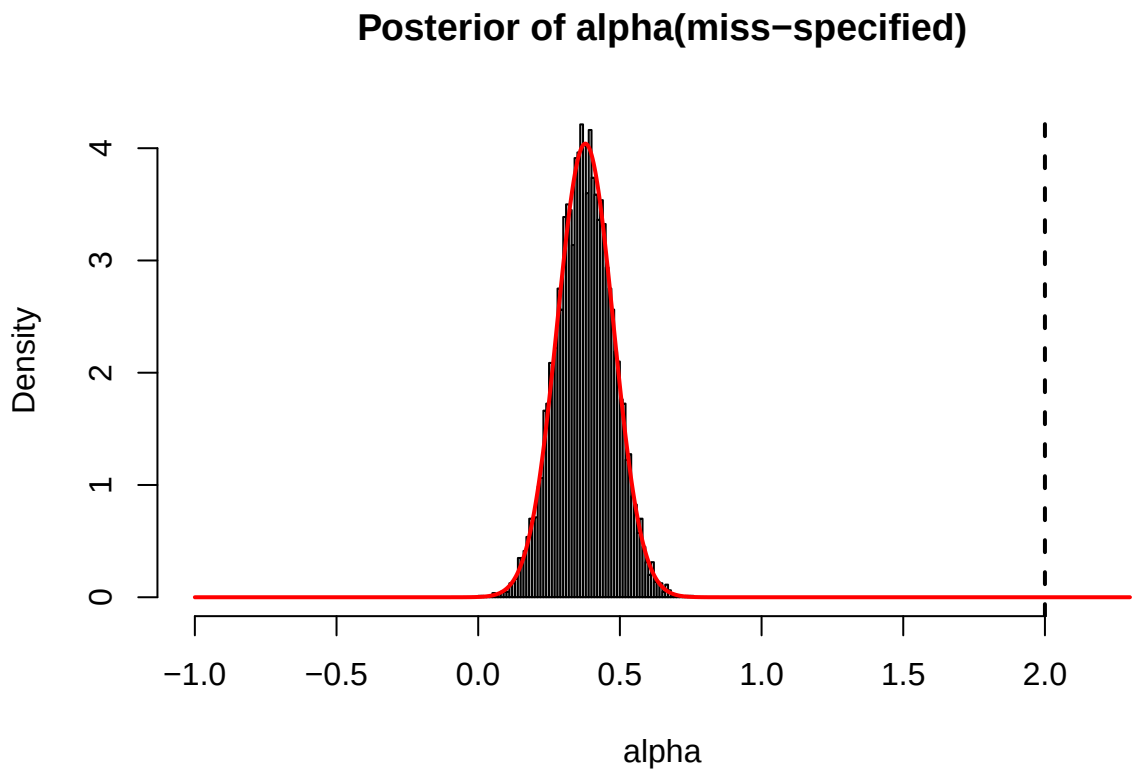
```
#Well Specified case
# Plot the posterior distributions
hist(well.spec$alpha_samples, breaks=100, main = "Posterior of alpha", xlab = "alpha", freq = F)
# Create a sequence of x values for the normal distribution curve
a_x_values <- seq(min(well.spec$alpha_samples), max(well.spec$alpha_samples),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



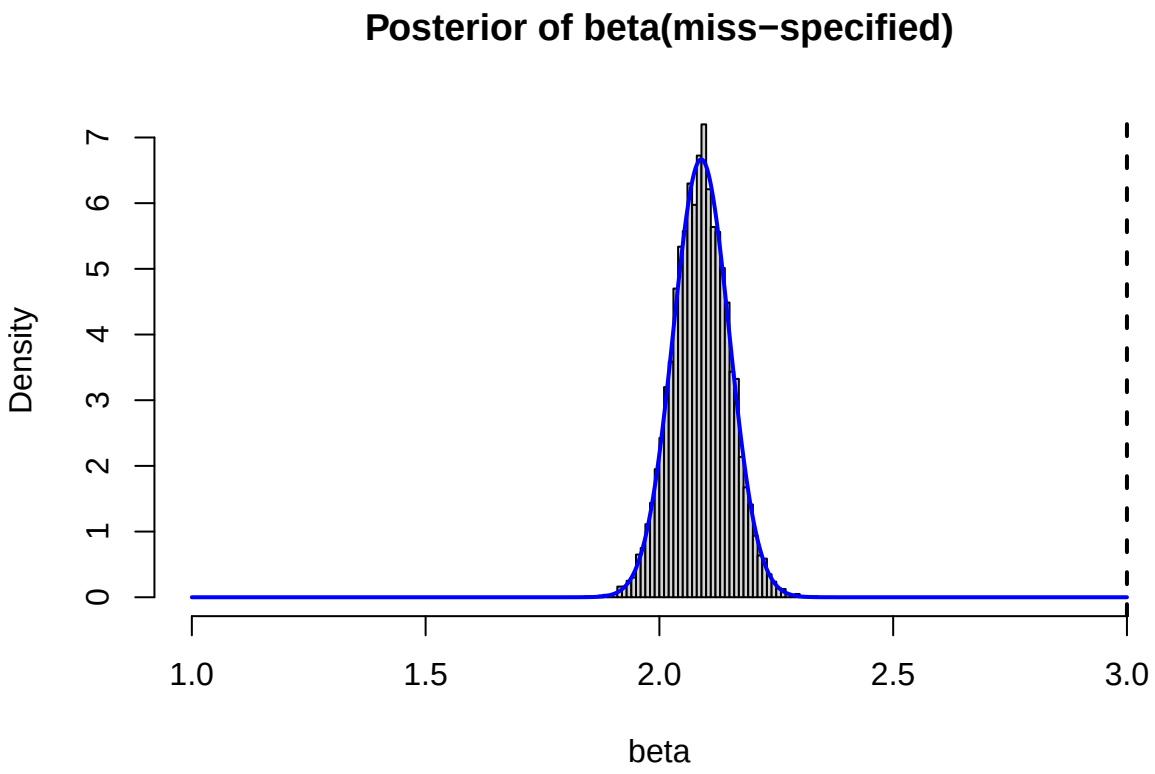

```r
# Plot the posterior distributions
hist(well.spec$beta_samples, breaks=100, main = "Posterior of beta", xlab = "beta", freq = F)
# Create a sequence of x values for the normal distribution curve
b_x_values <- seq(min(well.spec$beta_samples), max(well.spec$beta_samples), length = 100)

# 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
```




```r
#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
```




```
# 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)
}
```

```
library(foreach)
library(doParallel)
```

```
## Loading required package: iterators
```

```
## Loading required package: parallel
```

```
#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.
```

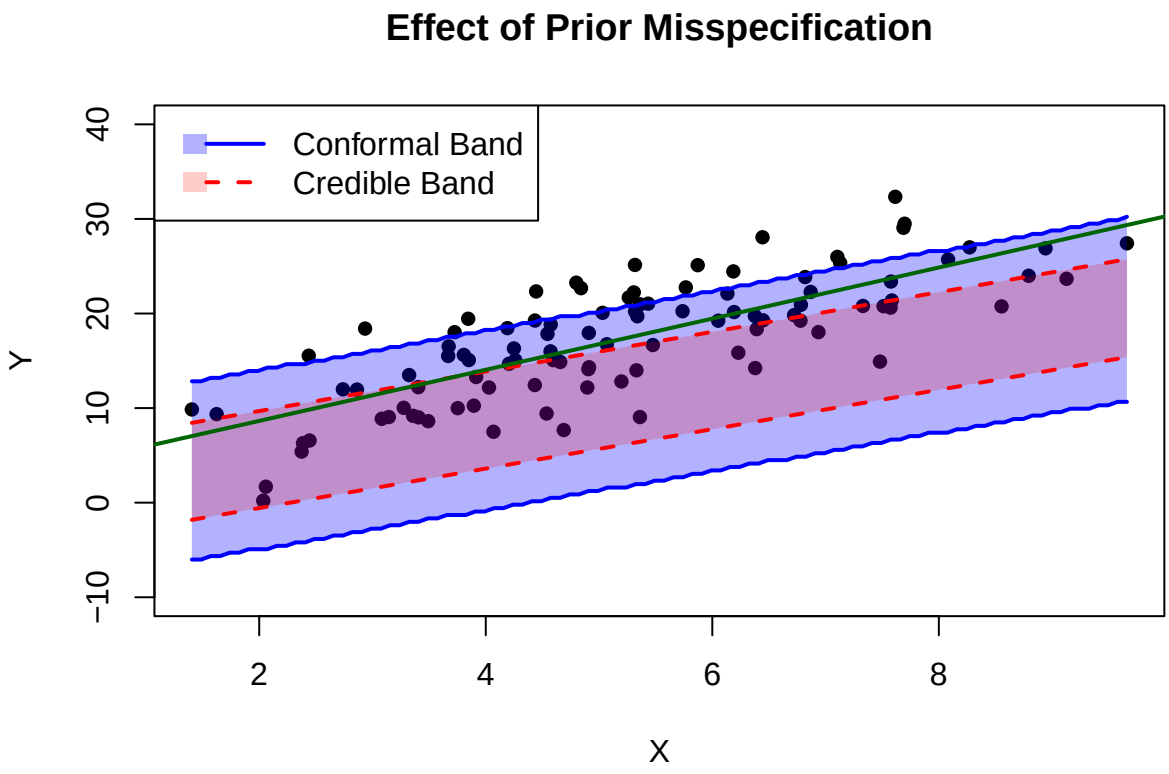
```
#well-spec
library("doParallel")
library("foreach")
#create a y grid to loop over
#ny is grid fineness for y
ny <-150
nx<- 100
y_grid <- seq(0,40, length.out = ny)
x_grid <- seq(min(x), max(x), length.out = nx )
#turn x-grid into design matrix
x_new_mat<-matrix(c(rep(1,nx),x_grid ), nrow=nx)
#this 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.
```

```
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)))
}
```

```
# 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)
```

```
# 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)

# 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")
```




```
well_conf_range
```

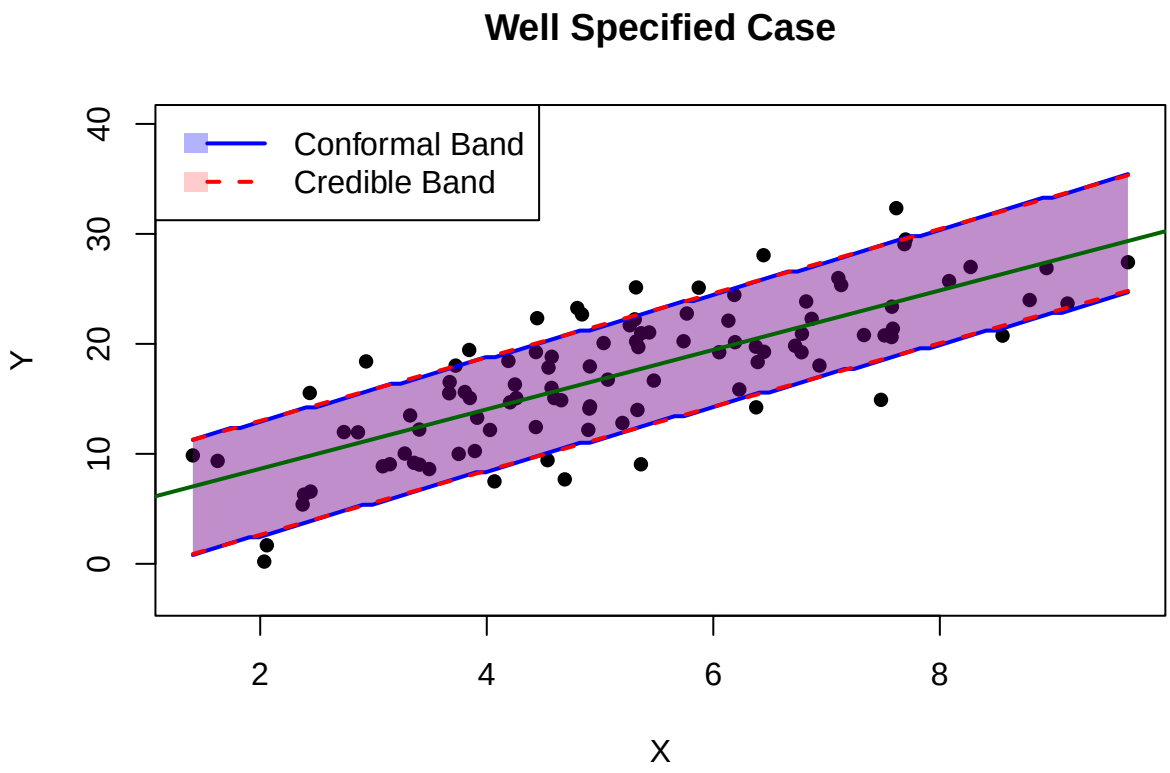
```
##                 [,1]     [,2]
## result.1    0.8053691 11.27517
## result.2    1.0738255 11.54362
## result.3    1.3422819 11.81208
## result.4    1.6107383 12.08054
## result.5    1.8791946 12.34899
## result.6    2.1476510 12.34899
## result.7    2.4161074 12.61745
## result.8    2.4161074 12.88591
## result.9    2.6845638 13.15436
## result.10   2.9530201 13.42282
## result.11   3.2214765 13.69128
## result.12   3.4899329 13.95973
## result.13   3.7583893 14.22819
## result.14   4.0268456 14.22819
## result.15   4.2953020 14.49664
## result.16   4.5637584 14.76510
## result.17   4.8322148 15.03356
## result.18   5.1006711 15.30201
## result.19   5.3691275 15.57047
## result.20   5.3691275 15.83893
## result.21   5.6375839 16.10738
## result.22   5.9060403 16.37584
## result.23   6.1744966 16.37584
```

```
## result.24   6.4429530 16.64430
## result.25   6.7114094 16.91275
## result.26   6.9798658 17.18121
## result.27   7.2483221 17.44966
## result.28   7.5167785 17.71812
## result.29   7.7852349 17.98658
## result.30   8.0536913 18.25503
## result.31   8.3221477 18.52349
## result.32   8.3221477 18.79195
## result.33   8.5906040 18.79195
## result.34   8.8590604 19.06040
## result.35   9.1275168 19.32886
## result.36   9.3959732 19.59732
## result.37   9.6644295 19.86577
## result.38   9.9328859 20.13423
## result.39  10.2013423 20.40268
## result.40  10.4697987 20.67114
## result.41  10.7382550 20.93960
## result.42  11.0067114 21.20805
## result.43  11.0067114 21.20805
## result.44  11.2751678 21.47651
## result.45  11.5436242 21.74497
## result.46  11.8120805 22.01342
## result.47  12.0805369 22.28188
## result.48  12.3489933 22.55034
## result.49  12.6174497 22.81879
## result.50  12.8859060 23.08725
## result.51  13.1543624 23.35570
## result.52  13.4228188 23.62416
## result.53  13.4228188 23.89262
## result.54  13.6912752 23.89262
## result.55  13.9597315 24.16107
## result.56  14.2281879 24.42953
## result.57  14.4966443 24.69799
## result.58  14.7651007 24.96644
## result.59  15.0335570 25.23490
## result.60  15.3020134 25.50336
## result.61  15.5704698 25.77181
## result.62  15.5704698 26.04027
## result.63  15.8389262 26.30872
## result.64  16.1073826 26.57718
## result.65  16.3758389 26.57718
## result.66  16.6442953 26.84564
## result.67  16.9127517 27.11409
## result.68  17.1812081 27.38255
## result.69  17.4496644 27.65101
## result.70  17.7181208 27.91946
## result.71  17.7181208 28.18792
## result.72  17.9865772 28.45638
## result.73  18.2550336 28.72483
## result.74  18.5234899 28.99329
## result.75  18.7919463 29.26174
## result.76  19.0604027 29.53020
## result.77  19.3288591 29.79866
```

```
## result.78  19.5973154 29.79866
## result.79  19.5973154 30.06711
## result.80  19.8657718 30.33557
## result.81  20.1342282 30.60403
## result.82  20.4026846 30.87248
## result.83  20.6711409 31.14094
## result.84  20.9395973 31.40940
## result.85  21.2080537 31.67785
## result.86  21.2080537 31.94631
## result.87  21.4765101 32.21477
## result.88  21.7449664 32.48322
## result.89  22.0134228 32.75168
## result.90  22.2818792 33.02013
## result.91  22.5503356 33.28859
## result.92  22.8187919 33.28859
## result.93  22.8187919 33.55705
## result.94  23.0872483 33.82550
## result.95  23.3557047 34.09396
## result.96  23.6241611 34.36242
## result.97  23.8926174 34.63087
## result.98  24.1610738 34.89933
## result.99  24.4295302 35.16779
## result.100 24.6979866 35.43624
```

```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)
```

```
## [1] 7520.873
```

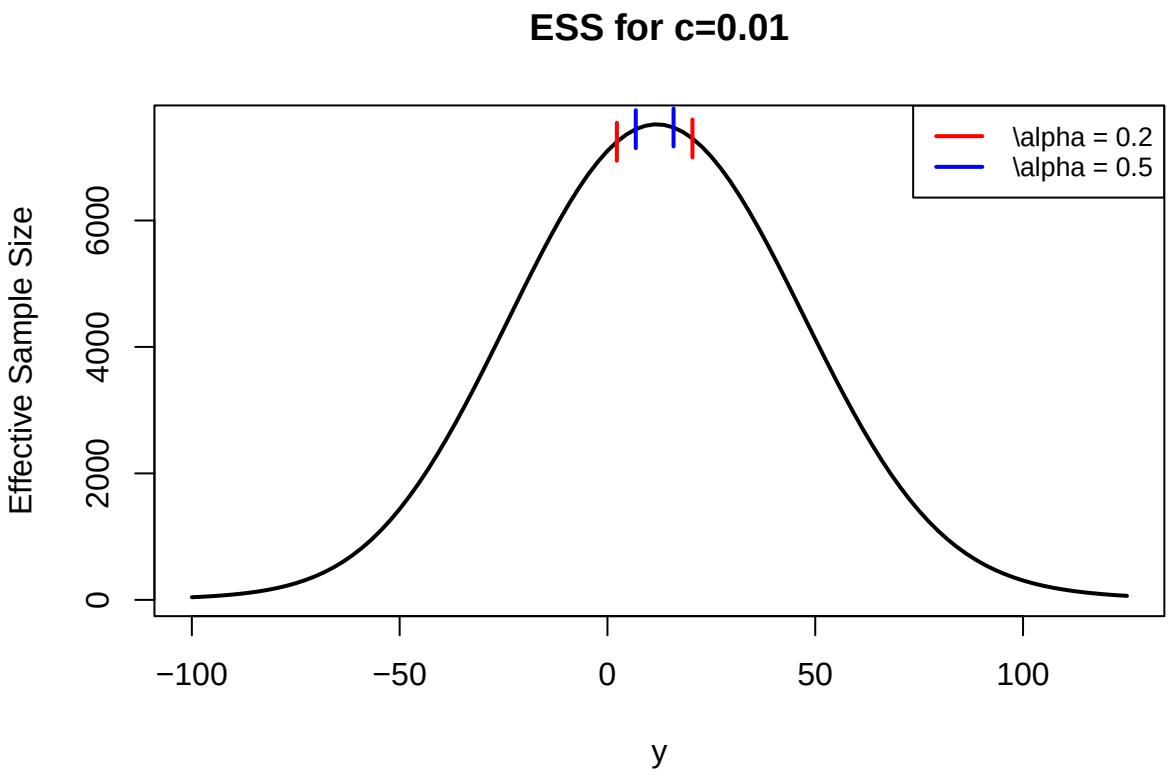
```
compute_min_ess(well.theta)
```

```
## [1] 822.414
```

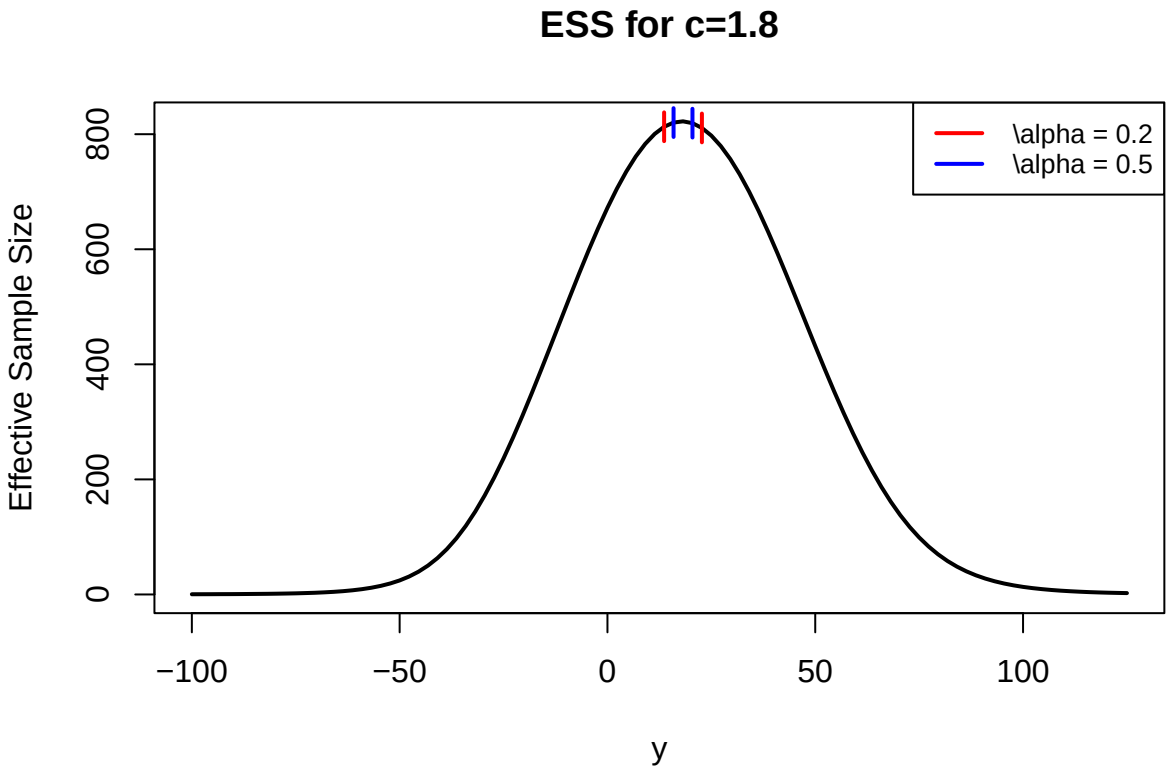
```r
# Function to compute and plot ESS with conformal bounds
#usual inputs, select up to 4 alpha values
#mcmc_ess is for scaling plot by min(ESS)/T
#linelen is a plotting parameter to select length to wich conformla bound will be extended vertically
plot_ess_with_bounds <- function(Y, X, y_grid, x_new,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

  # 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=
```


ESS for c=1.8
Effective Sample Size
0
200
400
600
800
−100
−50
0
50
100
y
\alpha = 0.2
\alpha = 0.5