Estimating ERP latencies using the hierarchical model SLAM and Monte Carlo EM algorithm

In this vignette, we provide worked examples on simulated data to demonstrates how to use the package `slam` to obtain full posterior distributions of ERP latencies, or in general the location of stationary points of a function, which is discussed in the paper "Semiparametric Latent ANOVA Model for Event-Related Potentials". Please note that with the current version of code, compiling the entire document takes over one hour. Faster implementation is under development.

Required Packages

The code and analysis use some packages that help develop the algorithm and data generation, and we load them before doing study.

```
## ===========================================================================
## load packages
## ===========================================================================
packages <- c("Matrix", "matrixcalc", "emulator", "doParallel", "Rsolnp",
              "mvnfast", "extraDistr", "truncnorm", "invgamma", "ggridges",
              "ggplot2", "tidyverse")
invisible(lapply(packages, library, character.only = TRUE))
```

```
library(slam)
```

Simulation

Data

The data presented in this vignette are generated and plotted using the script `./data-raw/GenMultiSimData.R` and saved in `./data/multi_sim_data.RData`. The replicates of data used for simulation analysis in the paper can be obtained from `./data-raw/data_replicates.R` and `./data-raw/data_from_model.R`.

```
## ===========================================================================
## Load the data
## ===========================================================================
load("../data/multi_sim_data.RData", verbose = TRUE)
```

```
## Loading objects:
##   YYcos
##   YYsin
##   sig
##   x_a
##   x_b
##   cri_pts_cos_lst
##   cri_pts_sin_lst
##   no_data
```

The object `YYsin` and `YYcos` save the data of 10 subjects in the `sin` and `cos` groups. `x_a` and `x_b` are the end points of the domain of `x`. `cri_pts_sin_lst` and `cri_pts_sin_lst` store the true stationary points for the subjects in each group, and `no_data` is the number of subjects which is 10 in both groups.

```
## ================================================================================
## Generate the simulated data
## ================================================================================
source("../data-raw/seed.R")
source("../data-raw/GenMultiSimData.R")
```

Simulation study

Here we show how to use the functions in the `slam` package to implement the method proposed in the paper. To ease computational burden, in this demo we consider one replicate of data only.

```
## ================================================================================
## Set parameter names
## ================================================================================
name_par <- c(paste0("t_sin_1_", 1:no_data), paste0("t_sin_2_", 1:no_data),
              paste0("t_cos_1_", 1:no_data), paste0("t_cos_2_", 1:no_data),
              "beta0_1", "beta0_2", "beta1_1", "beta1_2",
              "r1_sin", "r2_sin", "r1_cos", "r2_cos",
              "eta1_sin", "eta2_sin", "eta1_cos", "eta2_cos", "sigma")
```

```
## ================================================================================
## Set initial values
## ================================================================================
start_lst <- list(t_g1 = matrix(runif(no_data * 2,
                                         min = c(0, 0.5),
                                         max = c(0.5, 1)), 2, no_data),
                  t_g2 = matrix(runif(no_data * 2,
                                       min = c(0, 0.5),
                                       max = c(0.5, 1)), 2, no_data),
                  beta0_1 = 0, beta0_2 = 0, beta1_1 = 0, beta1_2 = 0,
                  eta1_g1 = 1, eta2_g1 = 1, eta1_g2 = 1, eta2_g2 = 1,
                  sigma = 1)
```

```
## ================================================================================
## Multi subject data
## ================================================================================
multi_y_cos <- sapply(YYcos, function(x) x$y)
multi_y_sin <- sapply(YYsin, function(x) x$y)
x <- YYcos[[1]]$x
x_test <- seq(0, 1, length.out = 150)
## The distance matrix in the powered exponential kernel function
H0_diff <- outer(x, x, FUN = function(x1, x2) (x1 - x2))
a_1 <- 0; b_1 <- 0.5
a_2 <- 0.5; b_2 <- 1
```

The MCEM algorithm is wrapped up as the function `mcem_slam()`. The required arguments include

- `multi_y_g1`: A $n \times S$ matrix saving multisuject response values of Group 1. n is the number of data points, and S is the number of subjects.
- `multi_y_g2`: A $n \times S$ matrix saving multisuject response values of Group 2.
- `x`: Input values.
- `H0`: Distance matrix in the powered exponential kernel function.
- `a_1, b_1`: Time search window for the 1st stationary point (latency).
- `a_2, b_2`: Time search window for the 2nd stationary point (latency).

- `start.lst`: A list saving initial values of parameters.
- `name.par`: A vector of parameter names.

Other arguments are optional. Please check the function to learn more about the function arguments.

The following code for running the algorithm takes about 13 to 15 minutes.

```
## ============================================================================
## MCEM algorithm implementation
## ============================================================================
system.time(slam_fit <- mcem_slam(multi_y_g1 = multi_y_sin,
                                  multi_y_g2 = multi_y_cos, x = x, H0 = H0_diff,
                                  a_1 = a_1, b_1 = b_1, a_2 = a_2,  b_2 = b_2,
                                  start.lst = start_lst, name.par = name_par,
                                  n_mcmc_e = 2100, burn_e = 100, thin_e = 1,
                                  n_mcmc_final = 11000, burn_final = 1000,
                                  thin_final = 2))
```

```
## [1] "MCEM SLAM begins"
## [1] "E-step sampling t, beta, eta and sigma"
## [1] "M-step updating kernel parameters"
## [1] "theta = 1.70089388917274"  "theta = 0.199822504864577"
## [1] "marg post = -2127.88190065049"
## [1] "eps = 1.1315  count 2"
## [1] "E-step sampling t, beta, eta and sigma"
## [1] "M-step updating kernel parameters"
## [1] "theta = 11.7818812260999"  "theta = 0.383958931189525"
## [1] "marg post = -414.050150223075"
## [1] "eps = 101.6602  count 3"
## [1] "E-step sampling t, beta, eta and sigma"
## [1] "M-step updating kernel parameters"
## [1] "theta = 15.7868084822235"  "theta = 0.399555351074611"
## [1] "marg post = -293.642522625093"
## [1] "eps = 16.0397  count 4"
## [1] "E-step sampling t, beta, eta and sigma"
## [1] "M-step updating kernel parameters"
## [1] "theta = 16.1438429931234"  "theta = 0.403353082827092"
## [1] "marg post = -291.43846522056"
## [1] "eps = 0.1275  count 5"
## [1] "E-step sampling t, beta, eta and sigma"
## [1] "M-step updating kernel parameters"
## [1] "theta = 15.475133040562"   "theta = 0.395198468140336"
## [1] "marg post = -292.819561043484"
## [1] "eps = 0.4472  count 6"
## [1] "E-step sampling t, beta, eta and sigma"
## [1] "M-step updating kernel parameters"
## [1] "theta = 16.8907355641574"  "theta = 0.410283024961729"
## [1] "marg post = -292.647176970062"
## [1] "eps = 2.0042  count 7"
## [1] "E-step sampling t, beta, eta and sigma"
## [1] "M-step updating kernel parameters"
## [1] "theta = 16.5874201579062"  "theta = 0.405964645824723"
## [1] "marg post = -291.638722561624"
## [1] "eps = 0.092  count 8"
## [1] "E-step sampling t, beta, eta and sigma"
## [1] "M-step updating kernel parameters"
```

```
## [1] "theta = 16.5873940115942"  "theta = 0.406737238441316"
## [1] "marg post = -291.465670030284"
## [1] "eps = 0  count 9"
## [1] "Final sampling for t, beta, eta, sigma"
##
## [1] "Done!"
```

```
##     user   system  elapsed
## 1358.768  141.318 1501.580
```

The function `mcem_slam()` returns the MCMC information as a list, including posterior samples of parameters, and adaptive tuning information such as acceptance rate of the Metropolis step. The function also returns the estimated hyperparameters τ and lengthscale h in the squared exponential kernel obtained from every M-step.

The hyperparameter estimates are shown below.

```r
slam_fit$theta_mat
```

```
##             tau         h
##  [1,]  1.000000 1.0000000
##  [2,]  1.700894 0.1998225
##  [3,] 11.781881 0.3839589
##  [4,] 15.786808 0.3995554
##  [5,] 16.143843 0.4033531
##  [6,] 15.475133 0.3951985
##  [7,] 16.890736 0.4102830
##  [8,] 16.587420 0.4059646
##  [9,] 16.587394 0.4067372
```

```r
theta_est <- slam_fit$theta_mat[nrow(slam_fit$theta_mat), ]
```

Population level latency estimates

We first grab the posterior samples of population level latency estimates. The function `change_to_r_latency()` turn the r variable in $(0, 1)$ into a value in the input space $\mathcal{X}$. Here we show the inference about the sine group. The inference about the cosine group follows the same idea.

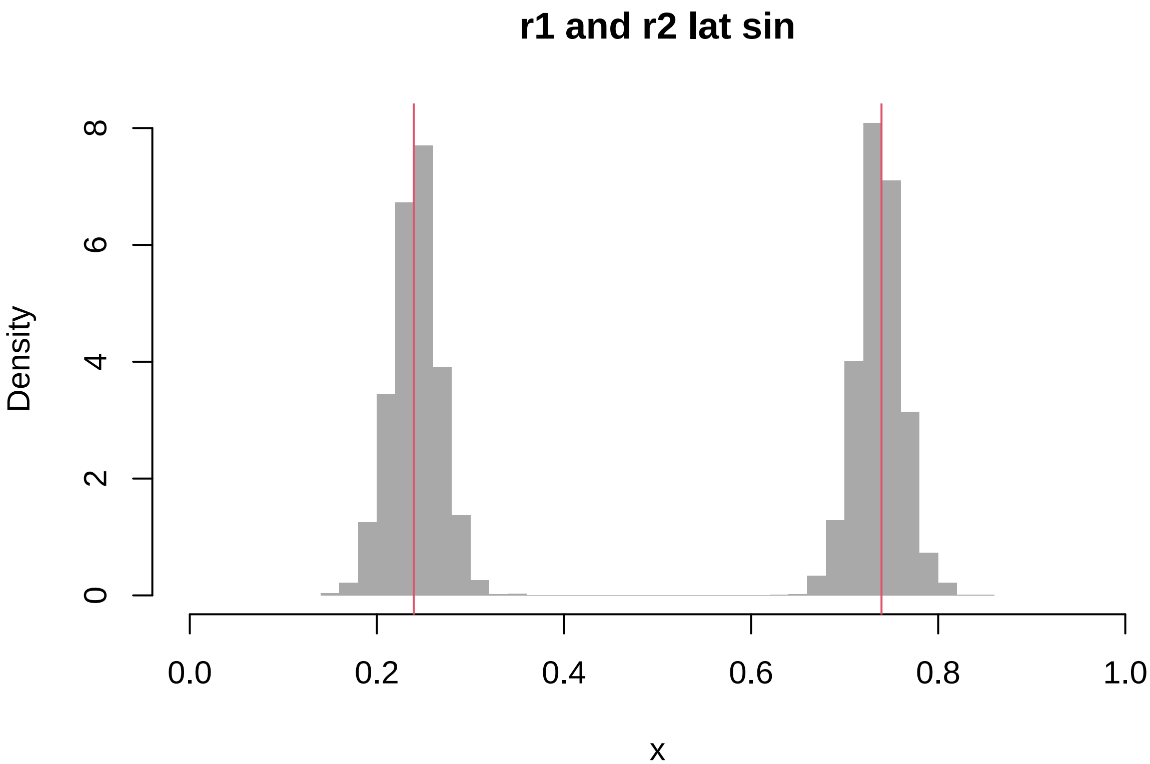
```r
##==================================
# Distribution of r-latency
##==================================
r1_sam_sin <- slam_fit$mcmc_output$draws[, "r1_sin"]
r2_sam_sin <- slam_fit$mcmc_output$draws[, "r2_sin"]
r1_sam_cos <- slam_fit$mcmc_output$draws[, "r1_cos"]
r2_sam_cos <- slam_fit$mcmc_output$draws[, "r2_cos"]
r1_sam_sin_lat <- slam::change_to_r_latency(r1_sam_sin, a_1, b_1)
r2_sam_sin_lat <- slam::change_to_r_latency(r2_sam_sin, a_2, b_2)
r1_sam_cos_lat <- slam::change_to_r_latency(r1_sam_cos, a_1, b_1)
r2_sam_cos_lat <- slam::change_to_r_latency(r2_sam_cos, a_2, b_2)
```

```r
cri_pts_sin <- matrix(unlist(cri_pts_sin_lst), 2, length(cri_pts_sin_lst))
cri_pts_sin_r <- apply(cri_pts_sin, 1, mean)
```

The true population level latency estimates are 0.2393896, 0.7393895. We can check the full latency distribution as follows. The vertical bars show the true r_1 and r_2.

```r
hist_t(c(r1_sam_sin_lat, r2_sam_sin_lat), a = 0, b = 1, xlab = "x",
       main = "r1 and r2 lat sin")
```

```r
abline(v = cri_pts_sin_r, col = 2)
```

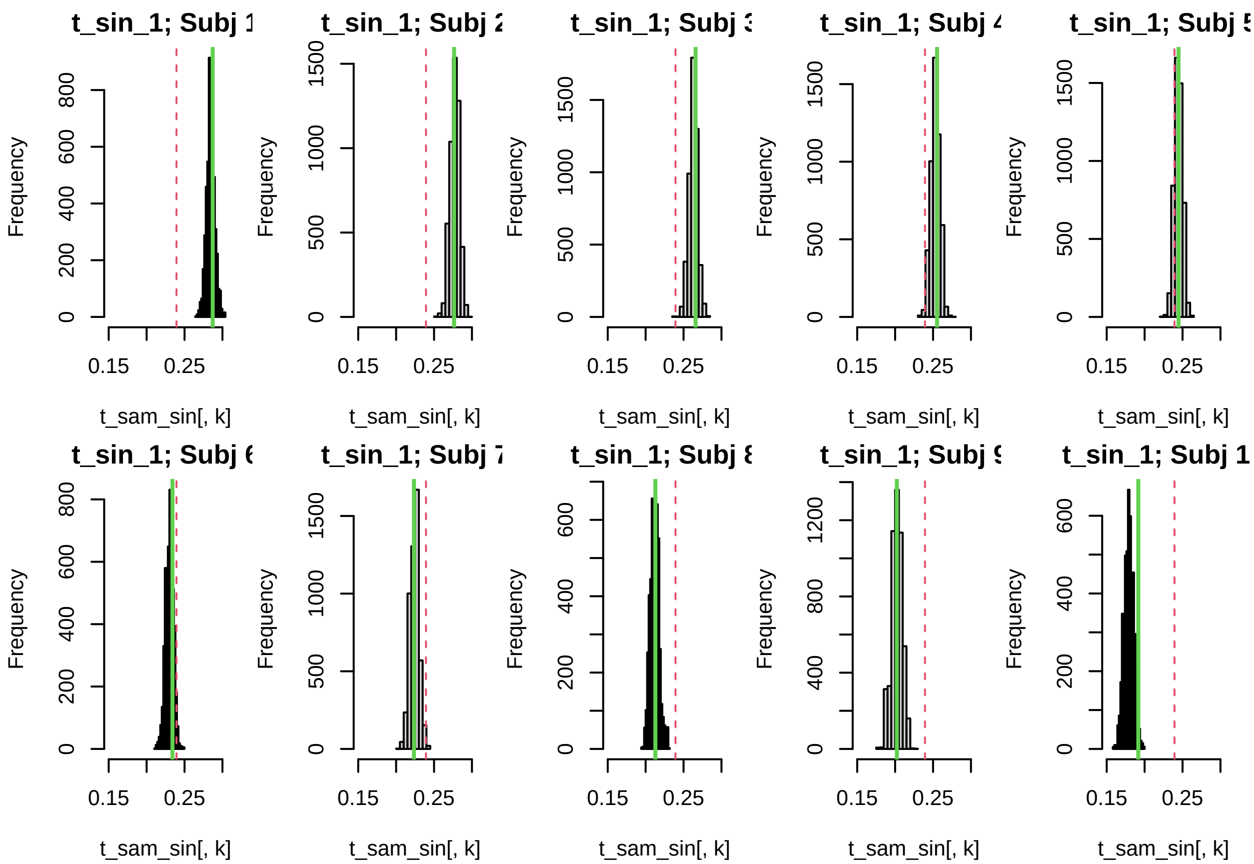



```r
# abline(v = cri_pts_sin, col = 4, lwd = 1)
```

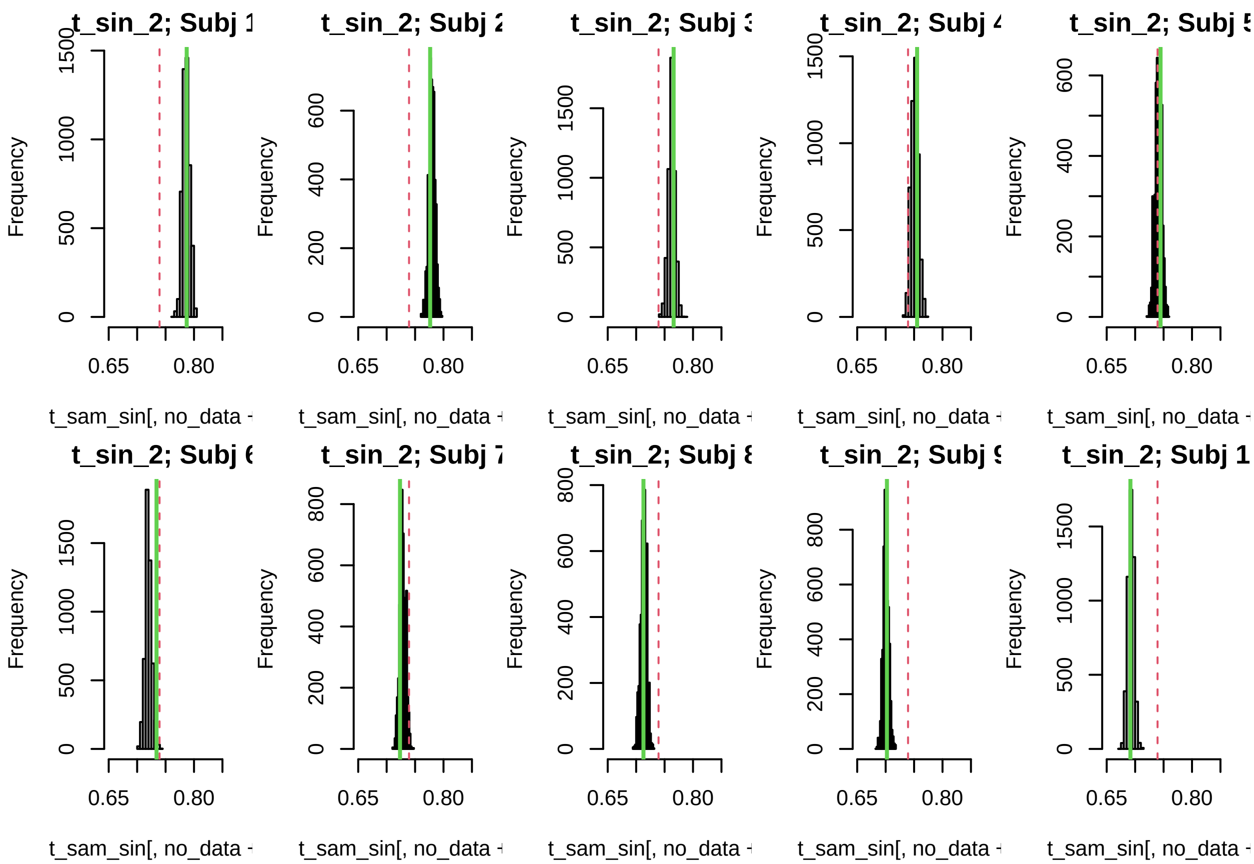
Subject level latency estimates

The proposed hierarchical model SLAM can also estimate latencies at the individual subject level. Using the sine group as illustration, we have the following distribution of subject latency. The red dashed vertical bar denotes the population latency level, and the green bars are the true subject-level latency.

```r
##==================================
# Distribution of ts
##==================================
t_sam_sin <- slam_fit$mcmc_output$draws[, 1:(2*no_data)]
par(mfrow = c(2, 5), mar = c(4, 4, 2, 1))
for (k in 1:no_data) {
    hist(t_sam_sin[, k], xlim = c(0.15, .3), main = paste("t_sin_1; Subj", k))
    abline(v = cri_pts_sin_r, col = 2, lty = 2)
    abline(v = cri_pts_sin[1, k], col = 3, lwd = 2)
}
```




```
for (k in 1:no_data) {
    hist(t_sam_sin[, no_data + k], xlim = c(0.65, 0.85),
         main = paste("t_sin_2; Subj", k))
    abline(v = cri_pts_sin_r, col = 2, lty = 2)
    abline(v = cri_pts_sin[2, k], col = 3, lwd = 2)
}
```



Curve fitting

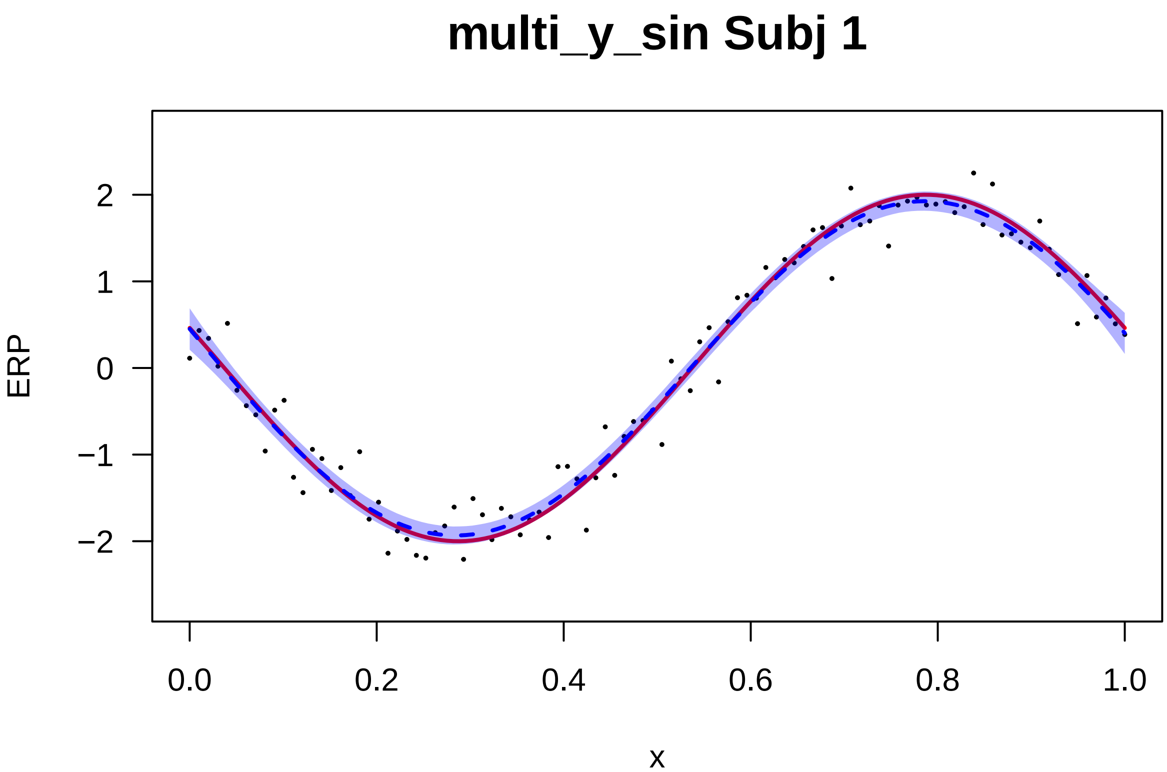
Although curve fitting is not the main objective of the proposed method, it can be done by posterior simulation. Given each posterior sample of t and σ, and hyperparameter estimates τ and h, a derivative Gaussian process is simulated. The function `slam::get_pi_t_sig` does that. The following does the curve fitting for the first subject in the sine group.

```
##===================================
# Predictive f
##===================================
pred_sin <- slam::get_pi_t_sig(yJ = c(multi_y_sin[, 1], rep(0, 2)),
                                x = x,
                                x_test = x_test,
                                idx_der = matrix(t_sam_sin[, c(1, no_data+1)], 5000, 2),
                                sample_sig = slam_fit$mcmc_output$draws[, "sigma"],
                                tau = theta_est[1],
                                h = theta_est[2])
```

The function `slam::plot_pred_gp_f_y()` plots the posterior mean fitted curve with 95% uncertainty interval. The red curve is the true regression curve, and the blue dashed line is the estimated one. The blue shaded area shows the 95% uncertainty band for the true function.

```
par(mfrow = c(1, 1))
plot_pred_gp_f_y(x = x, y = multi_y_sin[, 1],
                 x_test = x_test,
                 idx = x_test,
                 mu_test = pred_sin$mu_test,
                 CI_Low_f = pred_sin$ci_low,
```

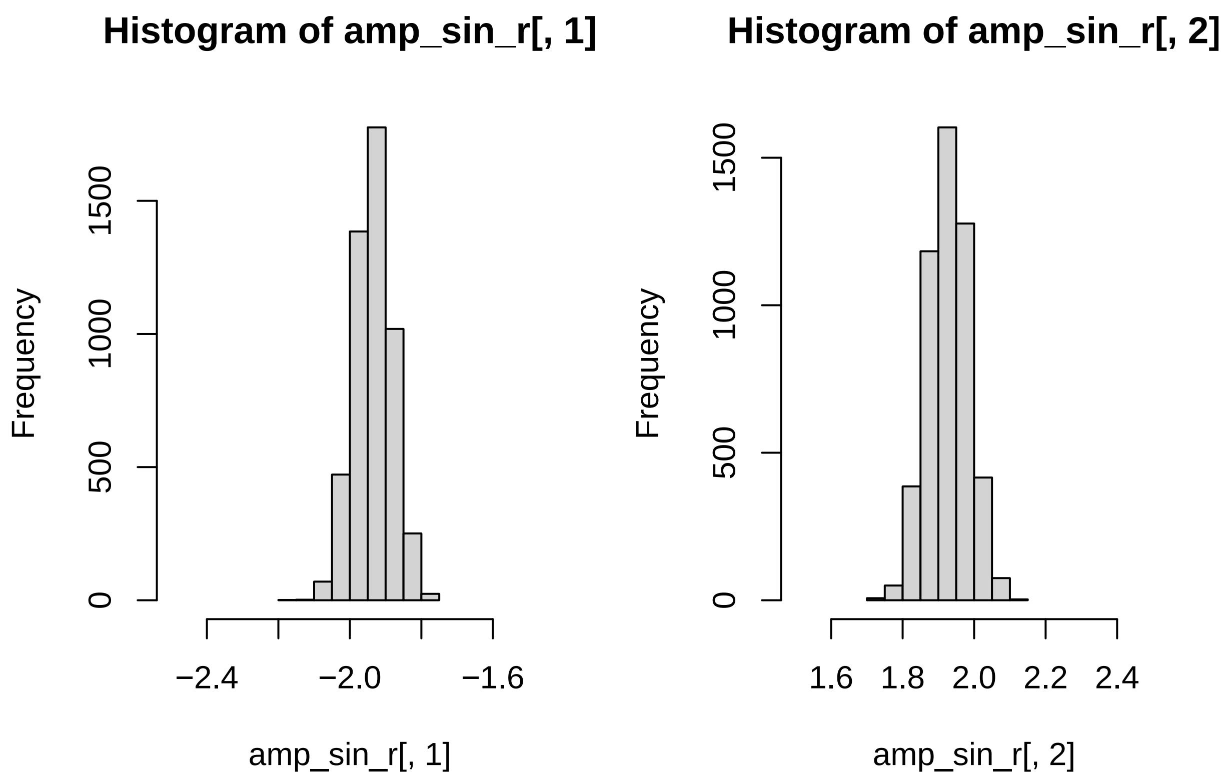
```
                  CI_High_f = pred_sin$ci_high,
                  is.der.line = FALSE,
                  xlim = c(0, 1),
                  ylim = c(min(multi_y_sin[, 1])-0.5, max(multi_y_sin[, 1])+0.5),
                  col.poly = rgb(0, 0, 1, 0.3), cex = 0.2,
                  pred_lwd = 2, title = "multi_y_sin Subj 1",
                  xlab = "x", ylab = "ERP", regfcn = regfcn_sin, k = 1)
```



Amplitude estimates

ERP amplitude can be estimated using various methods and algorithms. The package provides the `get_amp*` family of functions to estimate amplitude of ERP components. Here we use the Max Peak algorithm described in the paper to obtain the amplitude distribution. The function is `slam::get_amp_max()`. For the sine group, the true amplitude is -2 and 2. The amplitude distributions capture the amplitude size fairly well.

```
par(mfrow = c(1, 2))
amp_sin_r <- get_amp_max(pred_f = pred_sin$pred_f, x_test = x_test,
                         lat_sample_1 = r1_sam_sin_lat,
                         lat_sample_2 = r2_sam_sin_lat)
hist(amp_sin_r[, 1], xlim = c(-2.5, -1.5))
hist(amp_sin_r[, 2], xlim = c(1.5, 2.5))
```



ERP application

Data

The ERP data for the case study in the paper are saved in `./data/subj_y.RData`, `./data/subj_y_old.RData`, `./data/subj_erp_lst_group_mean.RData`, and `./data/subj_data_lst_old_11_bias_only.RData`.

The following code reproduce Figure 5 that shows the ERP data set.

```
load("../data/subj_erp_lst_group_mean.RData", verbose = TRUE)
```

```
## Loading objects:
##   subj_erp_lst_group_mean
```

```
idx_11_good <- c(1, 5, 7, 9, 11, 12, 13, 14, 15, 17, 20)
subj_erp_lst_group_mean_ts <- lapply(subj_erp_lst_group_mean,
                                     function(x) apply(x, 1, mean)[11:360])
subj_erp_mat_group_mean_ts <- sapply(subj_erp_lst_group_mean,
                                     function(x) apply(x, 1, mean)[11:360])
subj_erp_group_mean_all_11 <- apply(subj_erp_mat_group_mean_ts[, idx_11_good],
                                    1, mean)
load("../data/subj_data_lst_old_11_bias_only.RData", verbose = TRUE)
```

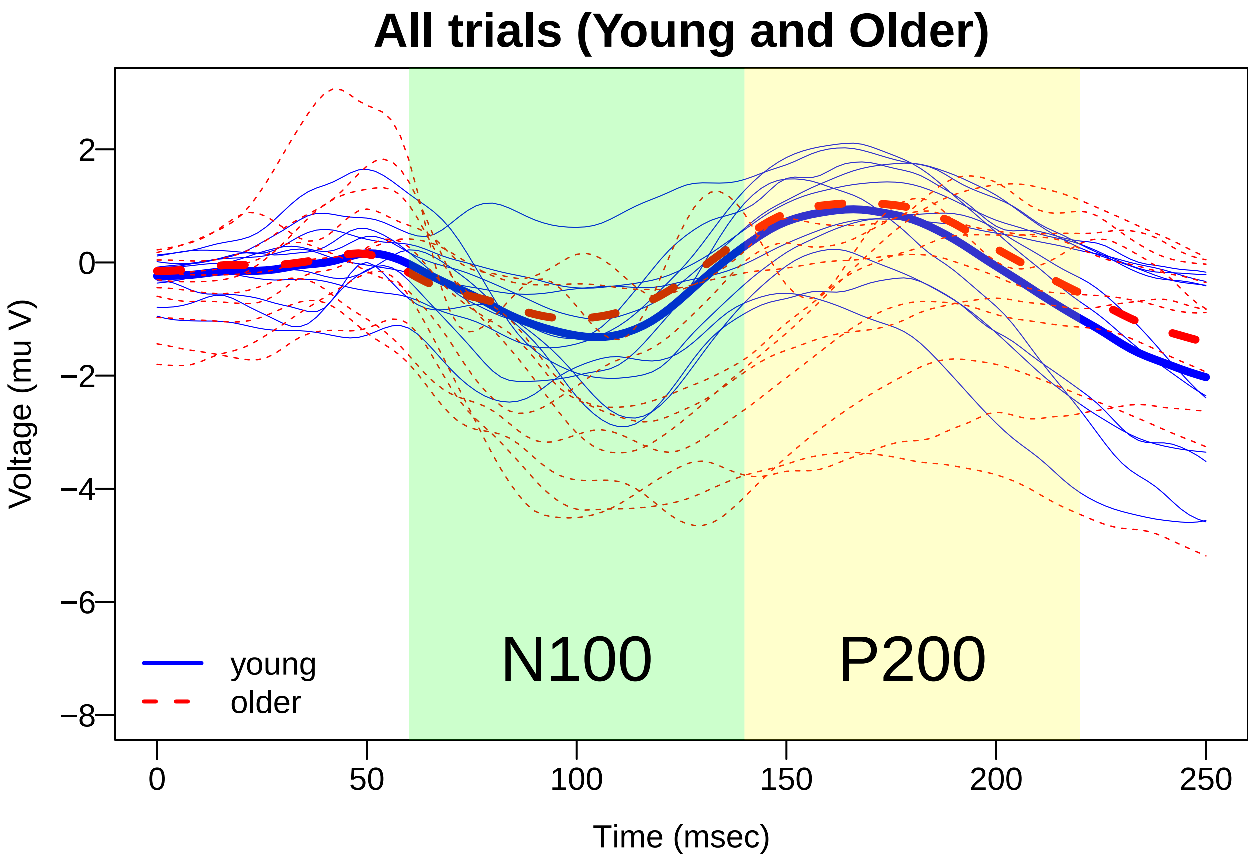
```
## Loading objects:
##   subj_data_lst_old
```

```
subj_erp_mat_group_mean_old <- sapply(subj_data_lst_old,
                                      function(x) apply(x, c(2, 3), mean))

subj_erp_lst_group_mean_old <- lapply(subj_data_lst_old,
                                      function(x) apply(x, c(2, 3), mean))
```

```r
subj_erp_lst_group_mean_old_ts <- lapply(subj_erp_lst_group_mean_old,
                                         function(x) apply(x, 1, mean)[11:360])
subj_erp_mat_group_mean_old_ts <- sapply(subj_erp_lst_group_mean,
                                     function(x) apply(x, 1, mean)[11:360])
subj_erp_group_mean_all_old <- apply(subj_erp_mat_group_mean_old_ts, 1, mean)
x_all <- seq(0.002, 0.7, by = 0.002)

par(mfrow = c(1, 1), mar = c(3, 3, 2, 0), mgp = c(2, 0.5, 0))
plot(c(x_all * 1000 - 200)[100:225], subj_erp_lst_group_mean_ts[[idx_11_good[1]]][100:225],
     type = "l", lty = 1, lwd = 0.5, ylim = c(-8, 3), ylab = "Voltage (mu V)",
     xlab = "Time (msec)", col = "blue", las = 1)
title(main = "All trials (Young and Older)", cex.main = 1.5)
for (i in 2:length(idx_11_good)) {
    lines(c(x_all * 1000 - 200)[100:225], subj_erp_lst_group_mean_ts[[idx_11_good[i]]][100:225],
          col = "blue", lwd = 0.5)
}
lines(c(x_all * 1000 - 200)[100:225], subj_erp_group_mean_all_11[100:225],
      lwd = 4, col = "blue")
for (i in 1:11) {
    lines(c(x_all * 1000 - 200)[100:225], subj_erp_lst_group_mean_old_ts[[i]][100:225],
          col = "red", lwd = 0.7, lty = 2)
}
lines(c(x_all * 1000 - 200)[100:225], subj_erp_group_mean_all_old[100:225], lwd = 4,
      col = "red", lty = 2)
polygon(c(seq(60, 140, length = 100), rev(seq(60, 140, length = 100))),
        c(rep(-10, 100), rep(5, 100)), col = rgb(0, 1, 0, 0.2), border = NA)
polygon(c(seq(140, 220, length = 100), rev(seq(140, 220, length = 100))),
        c(rep(-10, 100), rep(5, 100)), col = rgb(1, 1, 0, 0.2), border = NA)
text(x = 100, y = -7, "N100", cex = 2)
text(x = 180, y = -7, "P200", cex = 2)
legend("bottomleft", c("young", "older"), col = c("blue", "red"), lwd = c(2, 2),
       lty = c(1, 2), bty = "n")
```



The following code prepare the time input values, search window, and other objects needed for running the algorithm.

```
x_all <- seq(0.002, 0.7, by = 0.002)
sub_data_idx <- 125:225
x <- x_all[sub_data_idx]
x_s_all <- (x_all - min(x_all)) / (max(x_all) - min(x_all))   ## standardize to (0, 1)
x_s <- x_s_all[sub_data_idx]
x_test <- sort(seq(min(x), max(x), length.out = 150))
x_test_s <- sort(seq(min(x_s), max(x_s), length.out = 150))
H0_diff <- outer(as.vector(x_s), as.vector(x_s), FUN = function(x1, x2) (x1 - x2))
a_1 <- x_s[which(sec_to_msec(x, 200) == 60)]
b_1 <- x_s[which(sec_to_msec(x, 200) == 140)]
a_2 <- x_s[which(sec_to_msec(x, 200) == 140)]
b_2 <- x_s[which(sec_to_msec(x, 200) == 220)]
```

```
no.young <- 18; no.old <- 11
t_young <- t(matrix(c(rep(mean(c(a_1, b_1)), no.young), rep(mean(c(a_2, b_2)), no.young)),
                    ncol = 2, nrow = no.young))
t_old <- t(matrix(c(rep(mean(c(a_1, b_1)), no.old), rep(mean(c(a_2, b_2)), no.old)),
                  ncol = 2, nrow = no.old))
start_lst <- list(t_g1 = t_young, t_g2 = t_old, beta0_1 = 0, beta0_2 = 0,
                  beta1_1 = 0, beta1_2 = 0, eta1_g1 = 1, eta2_g1 = 1,
                  eta1_g2 = 1, eta2_g2 = 1, sigma = 1)
name_par <- c(paste0("t_young_1.", 1:18), paste0("t_young_2.", 1:18),
                   paste0("t_old_1.", 1:11),paste0("t_old_2.", 1:11),
                   "beta0_1", "beta0_2", "beta1_1", "beta1_2",
```

```
                   "r1_young", "r2_young", "r1_old", "r2_old",
                   "eta1_young", "eta2_young", "eta1_old", "eta2_old", "sigma")
```

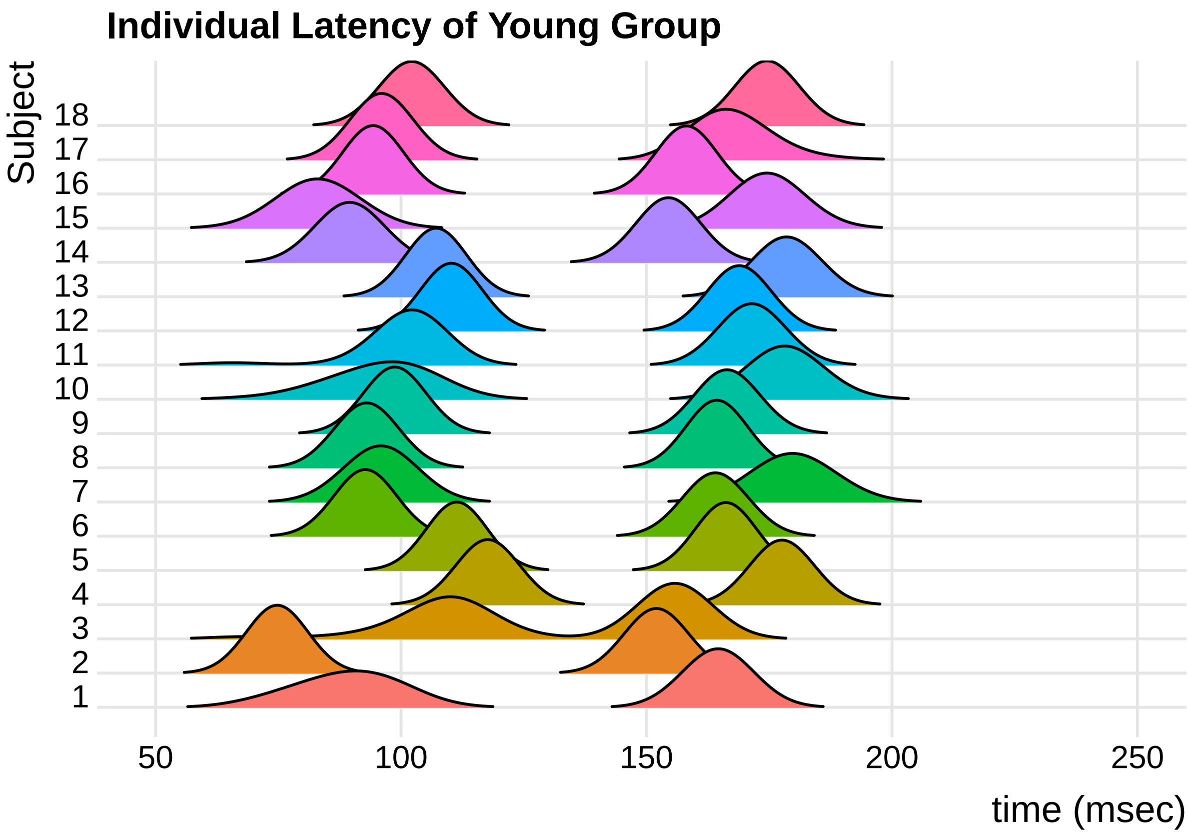
With the ERP data, the `mcem_slam()` takes about 3000 seconds to finish the algorithm.

```
load("../data/subj_y.RData", verbose = TRUE)
load("../data/subj_y_old.RData", verbose = TRUE)
idx_18 <- seq(1, 20)[-c(6, 18)]
subj_y_18 <- subj_y[, idx_18]
system.time(erp_slam <- mcem_slam(multi_y_g1 = subj_y_18,
    multi_y_g2 = subj_y_old, x = x_s, H0 = H0_diff, ga_shape = 200, ga_rate = 380,
    n_mcmc_e = 2100, burn_e = 100, n_mcmc_final = 21000, burn_final = 1000,
    thin_final = 10, start.lst = start_lst, name.par = name_par,
    a_1 = a_1, b_1 = b_1, a_2 = a_2, b_2 = b_2))
```

Once the algorithm is done, we have the posterior samples of latencies at the both subject and group level. The following code reproduce Figure 5 for the posterior distribution of t of the young group.

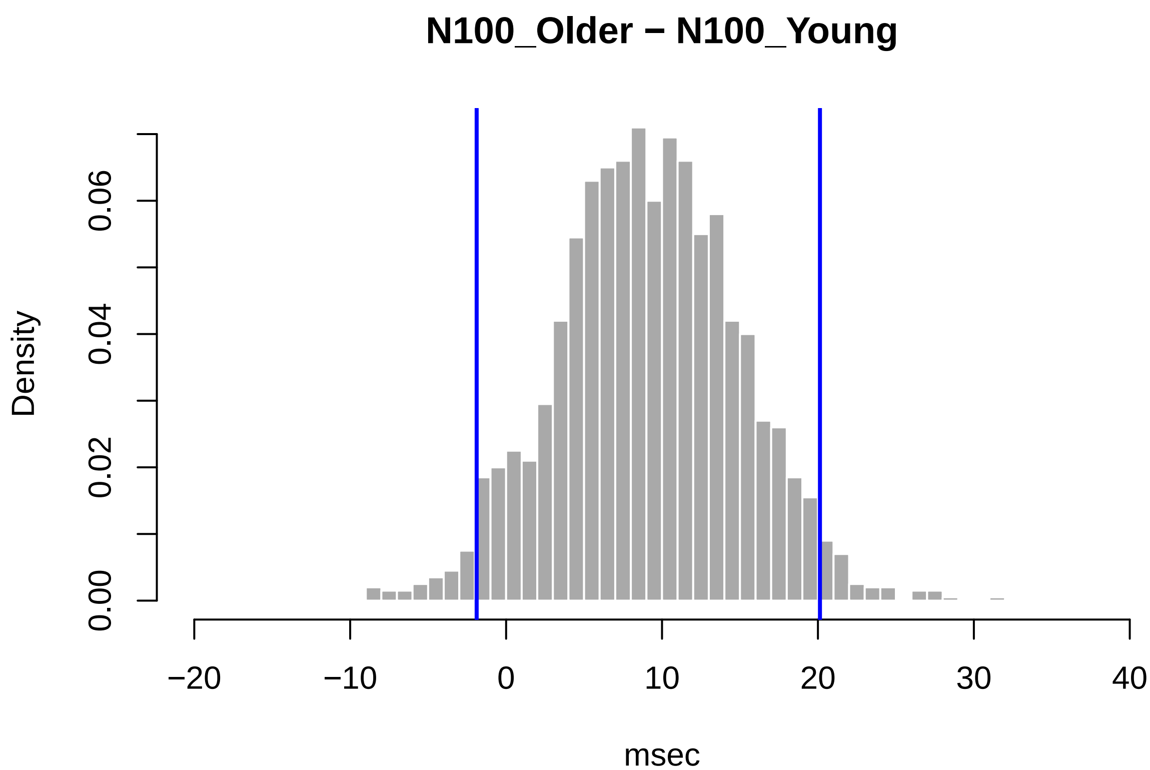
```
no.young <- 18
t_sam_young <- erp_slam$mcmc_output$draws[, 1:(2*no.young)]
t_sam_young <- t_sam_young * (max(x_all) - min(x_all)) + min(x_all)
t_tibble <- as_tibble(t_sam_young) |>
    pivot_longer(cols = t_young_1.1:t_young_2.18, names_to = "t",
                 values_to = "value")
t_tibble$t <- as.factor(rep(1:18, 4000))
ggplot(t_tibble, aes(x = sec_to_msec(value, 200), y = t, fill = t)) +
    geom_density_ridges_gradient(scale = 2, rel_min_height = 0.01) +
    theme_ridges() +
    theme(legend.position = "none") +
    xlab("time (msec)") +
    ylab("Subject") +
    xlim(50, 250) +
    ggtitle("Individual Latency of Young Group")
```

```
## Picking joint bandwidth of 6.05
```



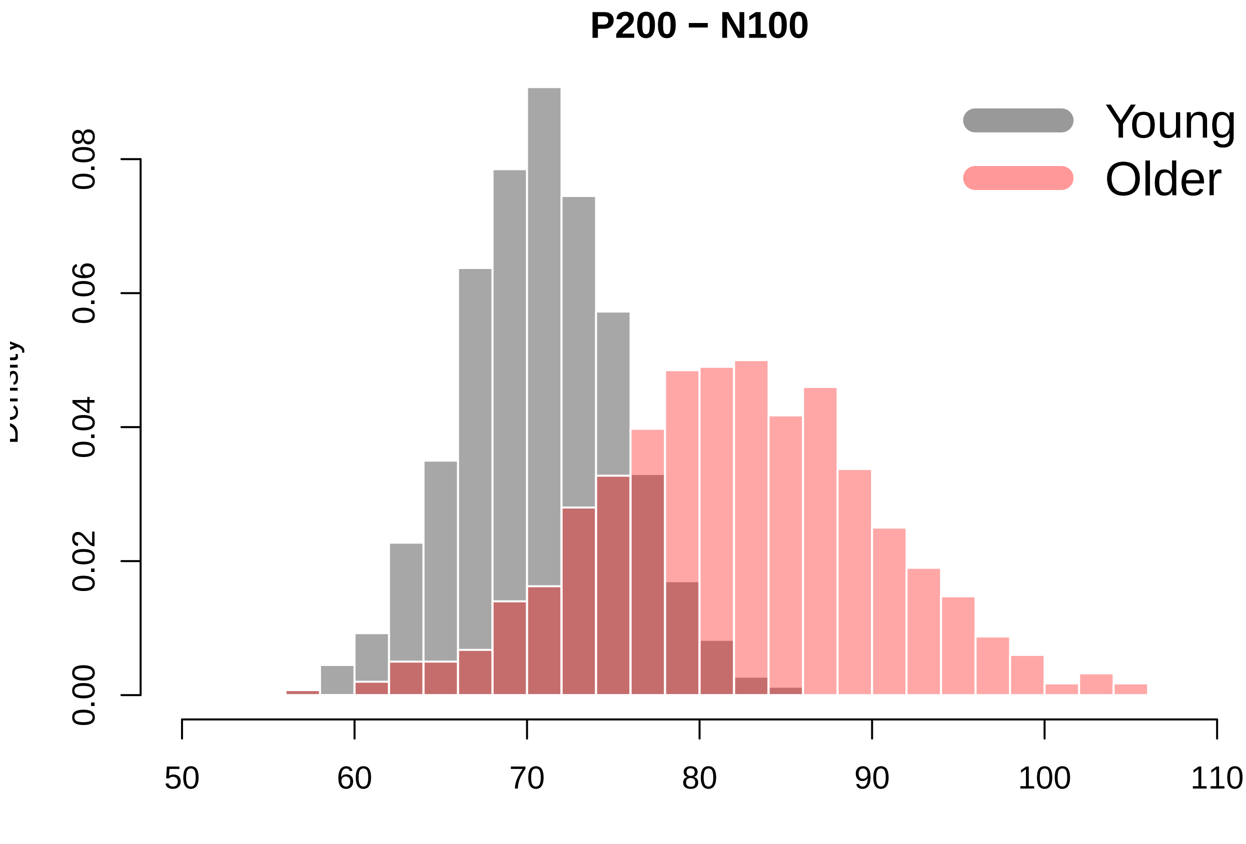
To reproduce Figure 6, we first obtain the samples of r, then scale them to unit msec for plotting.

```
r1_sam_young <- erp_slam$mcmc_output$draws[, which(name_par == "r1_young")]
r1_sam_old <- erp_slam$mcmc_output$draws[, which(name_par == "r1_old")]
r1_sam_young_lat <- change_to_r_latency(r1_sam_young, a_1, b_1)
r1_sam_old_lat <- change_to_r_latency(r1_sam_old, a_1, b_1)
r1_sam_young_lat <- r1_sam_young_lat * (max(x_all) - min(x_all)) + min(x_all)
r1_sam_old_lat <- r1_sam_old_lat * (max(x_all) - min(x_all)) + min(x_all)
hist(sec_to_msec(r1_sam_old_lat, 200) - sec_to_msec(r1_sam_young_lat, 200),
     col = "darkgrey", xlab = "msec", breaks = 30, border = "white",
     main = "N100_Older - N100_Young", freq = FALSE, xlim = c(-20, 40))
abline(v = quantile(sec_to_msec(r1_sam_old_lat, 200) -
                      sec_to_msec(r1_sam_young_lat, 200),
                    probs = c(0.025, 0.975)), col = "blue", lwd = 2)
```




```
par(mfrow = c(1, 1))
par(mar = c(3.4, 3.4, 2, 0))
r2_sam_young <- erp_slam$mcmc_output$draws[, which(name_par == "r2_young")]
r2_sam_old <- erp_slam$mcmc_output$draws[, which(name_par == "r2_old")]
r2_sam_young_lat <- change_to_r_latency(r2_sam_young, a_2, b_2)
r2_sam_old_lat <- change_to_r_latency(r2_sam_old, a_2, b_2)
r2_sam_young_lat <- r2_sam_young_lat * (max(x_all) - min(x_all)) + min(x_all)
r2_sam_old_lat <- r2_sam_old_lat * (max(x_all) - min(x_all)) + min(x_all)
r1_sam_young_lat_msec <- sec_to_msec(r1_sam_young_lat, 200)
r2_sam_young_lat_msec <- sec_to_msec(r2_sam_young_lat, 200)
r1_sam_old_lat_msec <- sec_to_msec(r1_sam_old_lat, 200)
r2_sam_old_lat_msec <- sec_to_msec(r2_sam_old_lat, 200)

hist(r2_sam_young_lat_msec - r1_sam_young_lat_msec,
     breaks = 20, col = rgb(0,0,0,0.35), border = "white", xlim = c(50, 110),
     xlab = "msec", main = paste("P200 - N100"), probability = TRUE)
hist(r2_sam_old_lat_msec - r1_sam_old_lat_msec, breaks = 30, col = rgb(1,0,0,0.35),
     border = "white", probability = TRUE, add = TRUE)
legend("topright", c("Young", "Older"), lwd = 12, bty = "n",
       col = c(rgb(0,0,0,0.4), rgb(1,0,0,0.4)), cex = 1.5)
```


P200 − N100
Young
Older
0.00
0.02
0.04
0.06
0.08
50
60
70
80
90
100
110