# es3

2024-01-13

## Load libraries

```
shhh = function(lib_name){ # It's a library, so shhh!
  suppressWarnings(suppressMessages(require(lib_name, character.only = TRUE)))
}
shhh("tidyverse")
shhh("ACutils")
shhh("mvtnorm")
shhh("salso")
shhh("FGM")
shhh("gmp")
shhh("mcclust")
shhh("mcclust.ext")
shhh("logr")
shhh("tidygraph")
shhh("ggraph")
shhh("igraph")
shhh("Rcpp")
shhh("RcppArmadillo")
shhh("RcppEigen")

## Load custom functions
source("functions/utility_functions.R");
source("functions/bulky_functions.R");
source("functions/data_generation.R")
sourceCpp("functions/wade.cpp")
Rcpp::sourceCpp('functions/UpdateParamsGSL.cpp')

library('RcppGSL')
library(fda)
library(tidyverse)
library(coda)
library(lattice)
```

## Goal

Stima della partizione e grafo con bulky functions 2024.

## Simulated data

Simuliamo i dati in sim_data.R con p=40 basi e 3 cluster di dimensioni [13,13,14]. Grafo e matrice di precisione sono generati attraverso la funzione Generate_BlockDiagonal.

```
# Import simulated data
BaseMat = as.matrix(read_csv('simulated_data/BaseMat.csv'))         # matrix Phi, dim = 200 x 40
y_hat_true = as.matrix(read_csv('simulated_data/y_hat_true.csv'))   # y true, dim = 300 x 200
beta_true = as.matrix(read_csv('simulated_data/beta_true.csv'))     # beta, dim = 300 x 40
mu_true = as.matrix(read_csv('simulated_data/mu_true.csv'))         # mu true, dim = 40 x 1

# object containing the true graph G, the true precision matrix K
simKG <- readRDS("simulated_data/simKG.rds")

n <- dim(y_hat_true)[1]   # n=300
r <- dim(y_hat_true)[2]   # r=200
p <- dim(BaseMat)[2]      # p=40
```

## Initialization

```
# Define the starting Beta matrix
Beta = t(beta_true)

# Define the starting value of mu
mu = mu_true

# Fix tau_eps (squared)
tau_eps = 100

# Define the starting precision matrix K and graph G
K = matrix(0,p,p)
G = matrix(0,p,p)

# Define the starting partition
rho = p
z = rep(1,p)

# Compute quantities for function UpdateParamGSL
tbase_base = t(BaseMat)%*%BaseMat              # p x p (phi_t * phi)
tbase_data = t(BaseMat)%*%t(y_hat_true)        # p x n (phi_t * Y_t)
Sdata = sum(diag(y_hat_true%*%t(y_hat_true))) # initialize phi*beta = 0

# Set True binary flag used to update values
Update_Beta <- FALSE
Update_Mu <- FALSE
Update_Tau <- FALSE

# Define hyperparameters values
a_tau_eps <- 2000
b_tau_eps <- 2
sigma_mu <- 100
```

```r
# Define variance of the Beta
beta_sig2 = 0.2

# Compute graph density
graph_density = sum(simKG$Graph) / (p*(p-1))

# Set the number of iterations and burn-in
niter <- 10000
burn_in <- 1000


# Create a list for chains to save the values of each iteration
chains <- list(
  Beta = vector("list", length = niter),
  mu = vector("list", length = niter),
  tau_eps = vector("list", length = niter),
  K = vector("list", length = niter),
  G = vector("list", length = niter),
  z = vector("list", length = niter),
  rho = vector("list", length = niter),
  time = vector("list", length = niter),
  sigma_prior = vector("list", length = niter),
  theta_prior = vector("list", length = niter)
)


# Initialization of the chains
chains$Beta[[1]] <- Beta
chains$mu[[1]] <- mu
chains$tau_eps[[1]] <- tau_eps
chains$K[[1]] <- K
chains$G[[1]] <- G
chains$z[[1]] <- z
chains$rho[[1]] <- rho
chains$time <- 0          # execution time
chains$sigma <- 0.5       # parameter of the prior of the partition
chains$theta <- 1         # parameter of the prior of the partition

# initialization of parameters for set_options
weights_a <- rep(1,p-1)  # starting weights
weights_d <- rep(1,p-1)  # starting weights
total_weights <- 0       # sum of weights
total_K <- K
total_graphs <- G
graph_start <- NULL
```

## Gibbs sampler

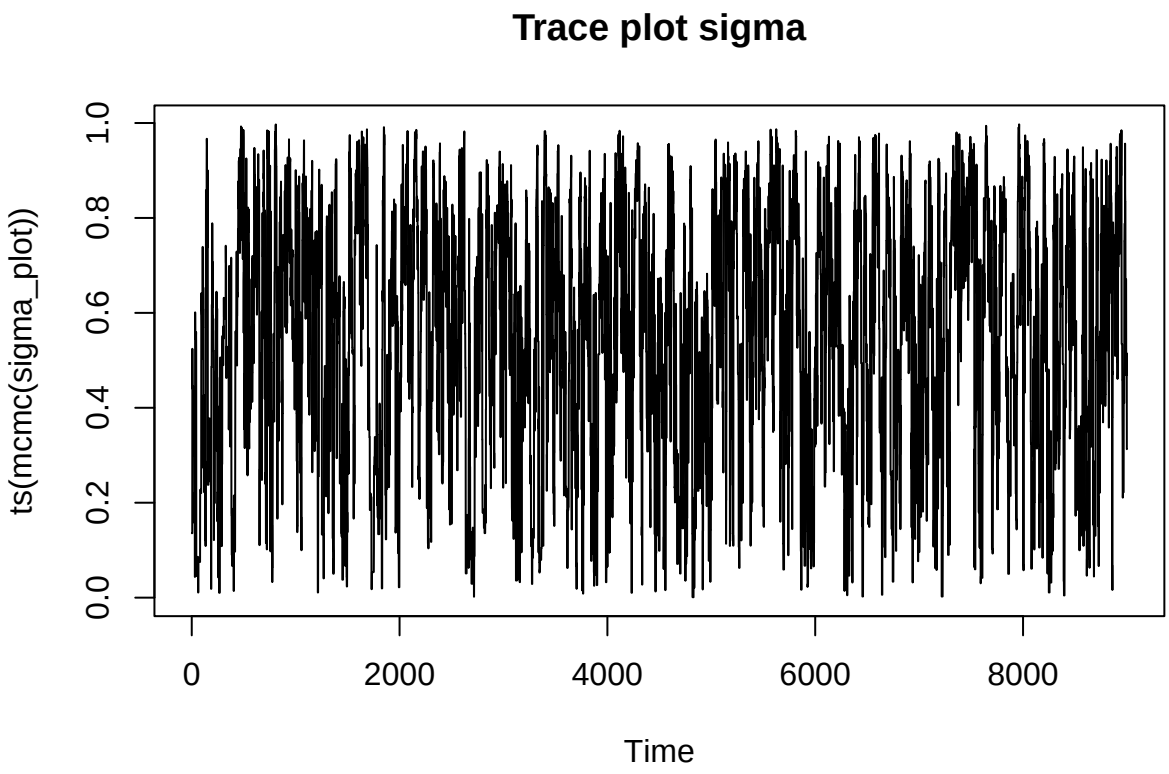
```r
for(s in 2:niter) {
```

```
fit = UpdateParamsGSL(
  chains$Beta[[s-1]],
  chains$mu[[s-1]],
  chains$tau_eps[[s-1]],
  chains$K[[s-1]],
  tbase_base,
  tbase_data,
  Sdata,
  a_tau_eps,
  b_tau_eps,
  sigma_mu,
  r,
  Update_Beta,
  Update_Mu,
  Update_Tau
  )

# Save Beta
chains$Beta[[s]] <- fit$Beta

# Save mu
chains$mu[[s]] <- fit$mu

# Save tau
chains$tau_eps[[s]] <- fit$tau_eps

# Set options for a single iteration of the Gibbs_sampler
options = set_options(
  sigma_prior_0=chains$sigma[[s-1]],
  sigma_prior_parameters=list("a"=1,"b"=1,"c"=1,"d"=1),
  theta_prior_0=chains$theta[[s-1]],
  theta_prior_parameters=list("c"=1,"d"=1),
  rho0=chains$rho[[s-1]],
  weights_a0=weights_a,
  weights_d0=weights_d,
  total_weights0=total_weights,
  total_K0 = total_K,
  total_graphs0 = total_graphs,
  graph = graph_start,
  alpha_target=0.234,
  beta_mu=graph_density,    # expected value beta distr of the graph
  beta_sig2=beta_sig2,      # var beta distr del grafo, fra 0 e 0.25
  d=3,                      # param della G wishart (default 3)
  alpha_add=0.5,
  adaptation_step=1/(p*1000),
  update_sigma_prior=TRUE,
  update_theta_prior=TRUE,
  update_weights=TRUE,
  update_partition=TRUE,
  update_graph=TRUE,
  perform_shuffle=TRUE
  )
```

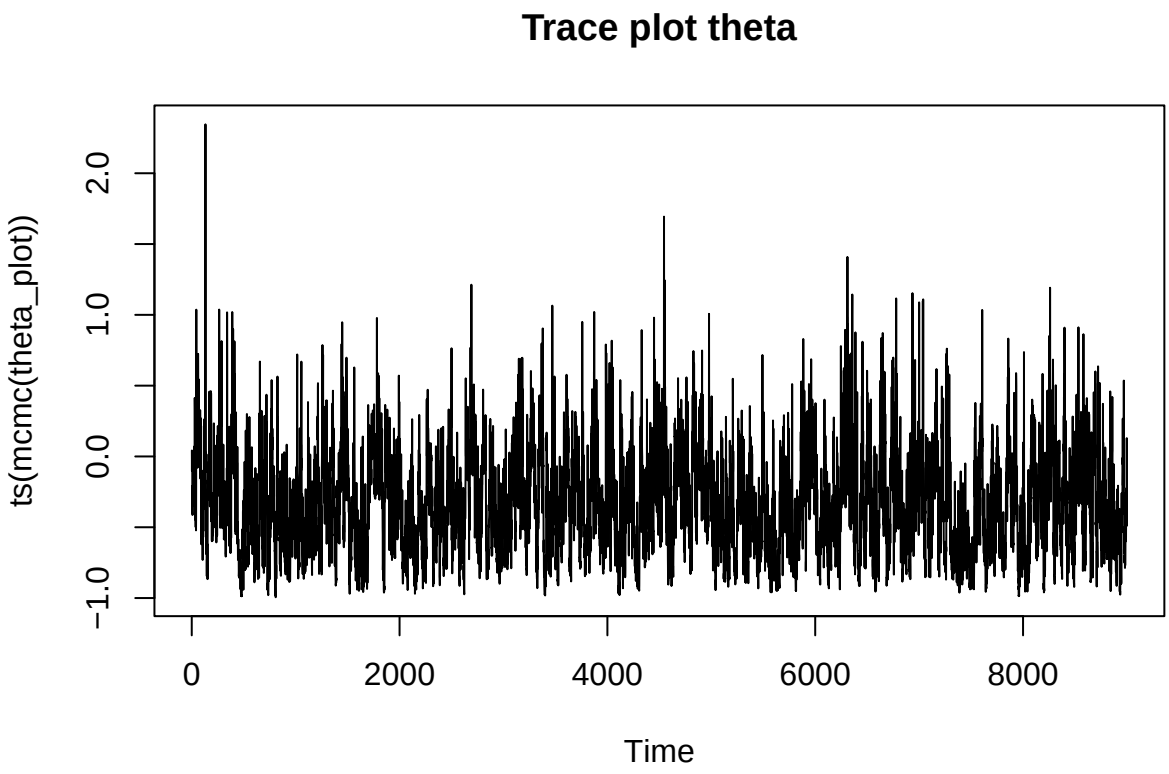
```
  # Run an iteration of the Gibbs Sampler
  res <- Gibbs_sampler(
    data = t(fit$Beta - fit$mu),
    niter = 1, # niter finali, già tolto il burn in
    nburn = 0,
    thin = 1,
    options = options,
    seed = 123456,
    print = FALSE
  )

  z = do.call(rbind, lapply(res$rho, rho_to_z))

  # Save rho
  chains$rho[[s]] <- res$rho[[1]]

  # Save K
  chains$K[[s]] <- res$K [[1]]

  # Save G
  chains$G[[s]] <- res$G [[1]]

  # Save z
  chains$z[[s]] <- z

  # Save times for each K
  chains$time[[s]] <- res$execution_time

  # Save sigma amd theta
  chains$sigma[[s]] <- res$sigma[[1]]
  chains$theta[[s]] <- res$theta[[1]]

  # Update quantities for the next iteration
  weights_a <- res$weights_a[[1]]
  weights_d <- res$weights_d[[1]]
  total_weights <- res$total_weights
  total_K <- res$total_K[[1]]
  total_graphs <- res$total_graphs[[1]]
  graph_start <- res$bdgraph_start

}
```

## Useful plots

```
sigma_plot <- as.vector(chains$sigma)
sigma_plot <- sigma_plot[(burn_in+1):niter]
plot(ts(mcmc(sigma_plot)), main='Trace plot sigma')
```




```
theta_plot <- as.vector(chains$theta)
theta_plot <- theta_plot[(burn_in+1):niter]
plot(ts(mcmc(theta_plot)), main='Trace plot theta')
```

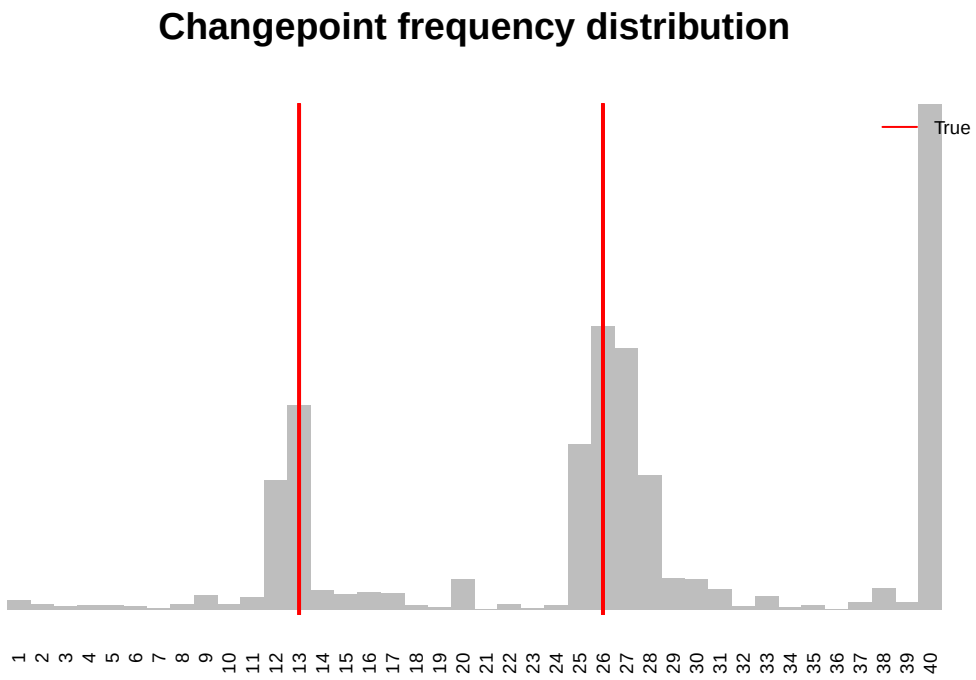



```
# Posterior analysis

## Recomputing the partition in other forms and the number of groups

rho_true = c(13,13,14)
r_true = rho_to_r(rho_true)
z_true = rho_to_z(rho_true)
p = length(z_true)
num_clusters_true = length(rho_true)
rho <- chains$rho
r = do.call(rbind, lapply(chains$rho, rho_to_r))
```

```
## Warning in (function (..., deparse.level = 1) : number of columns of result is
## not a multiple of vector length (arg 2)
```

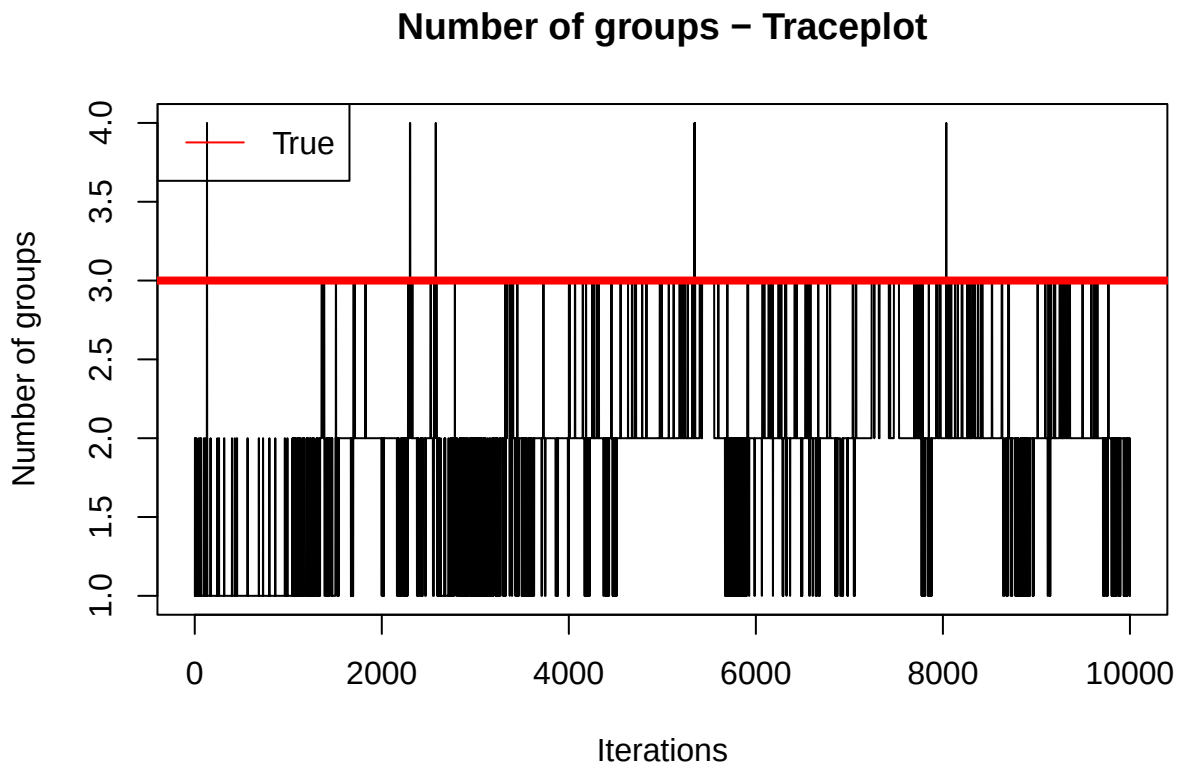
```
z = do.call(rbind, lapply(chains$rho, rho_to_z))
num_clusters = do.call(rbind, lapply(chains$rho, length))
num_clusters = as.vector(num_clusters)



### Barplot of changepoints
bar_heights = colSums(r)
cp_true = which(r_true==1)
color <- ifelse(seq_along(bar_heights) %in% c(cp_true), "red", "gray")
```

```r
barplot(
  bar_heights,
  names = seq_along(bar_heights),
  border = "NA",
  space = 0,
  yaxt = "n",
  main="Changepoint frequency distribution",
  #col = color,
  cex.names=.6,
  las=2
)

abline(v=cp_true-0.5, col="red", lwd=2)
legend("topright", legend=c("True"), col=c("red"),
       bty = "n",
       lty = 1,
       cex = 0.6)
```

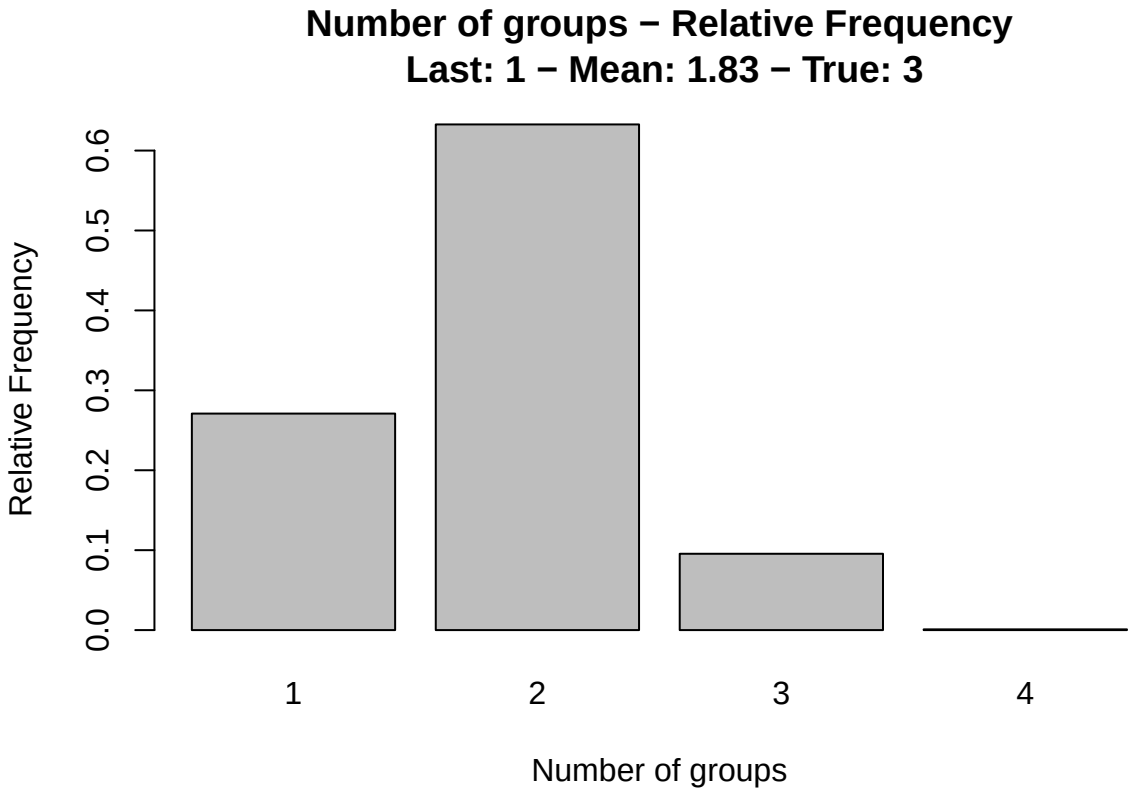



```r
### Evolution of the number of clusters
plot(
  x = seq_along(num_clusters),
  y = num_clusters,
  type = "n",
  xlab = "Iterations",
  ylab = "Number of groups",
```

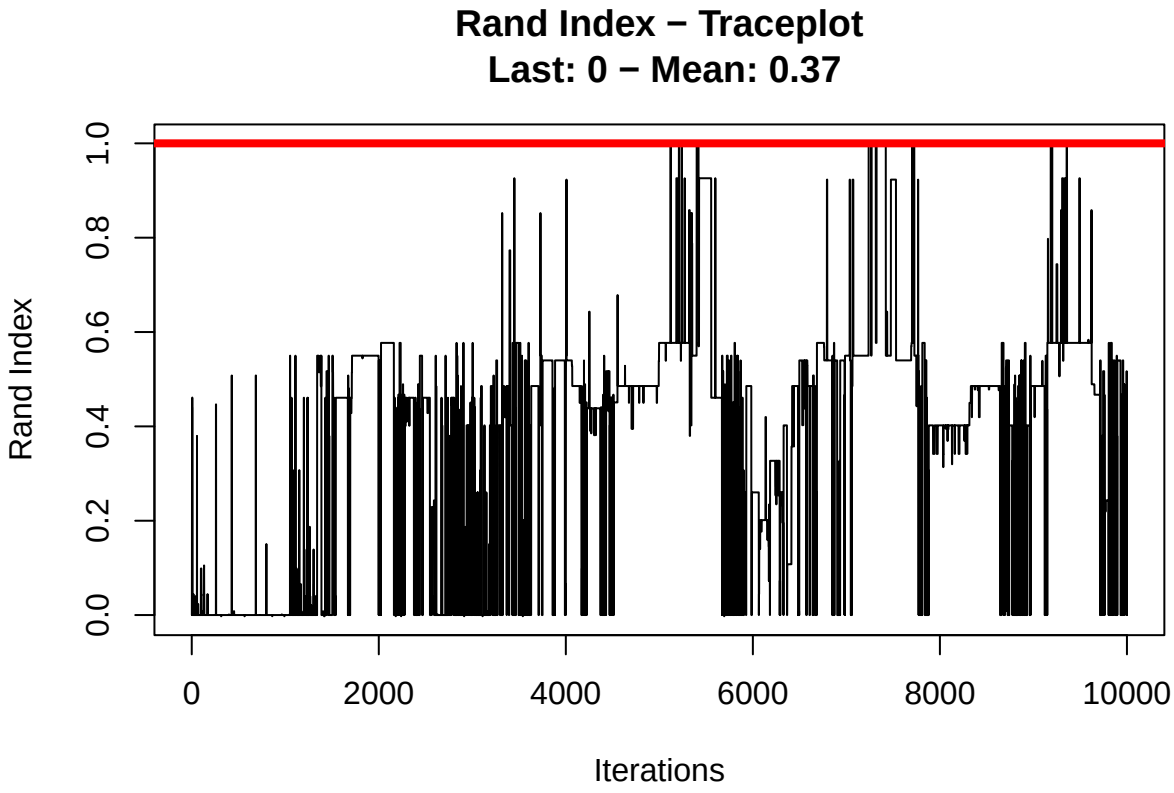
```
  main = "Number of groups - Traceplot"
)
lines(x = seq_along(num_clusters), y = num_clusters)
abline(h = length(z_to_rho(z_true)),
       col = "red",
       lwd = 4)
legend("topleft", legend=c("True"), col=c("red"),
       lty = 1,
       cex = 1)
```




```
barplot(
  prop.table(table(num_clusters)),
  xlab = "Number of groups",
  ylab = "Relative Frequency",
  main = paste(
    "Number of groups - Relative Frequency\n",
    "Last:",
    tail(num_clusters, n = 1),
    "- Mean:",
    round(mean(num_clusters), 2),
    "- True:",
    num_clusters_true
  )
)
```




```
### Evolution of the Rand Index

# computing rand index for each iteration
rand_index = apply(z, 1, mcclust::arandi, z_true)

# plotting the traceplot of the index
plot(
  x = seq_along(rand_index),
  y = rand_index,
  type = "n",
  xlab = "Iterations",
  ylab = "Rand Index",
  main = paste(
    "Rand Index - Traceplot\n",
    "Last:",
    round(tail(rand_index, n=1), 3),
    "- Mean:",
    round(mean(rand_index), 2)
  )
)
lines(x = seq_along(rand_index), y = rand_index)
abline(h = 1, col = "red", lwd = 4)
```




```
### Retrieving best partition using VI on visited ones (order is guaranteed here)
# compute VI
sim_matrix <- salso::psm(z)
dists <- VI_LB(z, psm_mat = sim_matrix)

# select best partition (among the visited ones)
best_partition_index = which.min(dists)
rho_est = rho[[best_partition_index]]
z_est = z[best_partition_index,]

# VI loss
dists[best_partition_index]
```

```
## [1] 0.5802937
```

```
# select best partition
unname(z_est)
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
## [39] 1 1
```

```
# compute Rand Index
mcclust::arandi(z_est, z_true)
```

```
## [1] 0
```

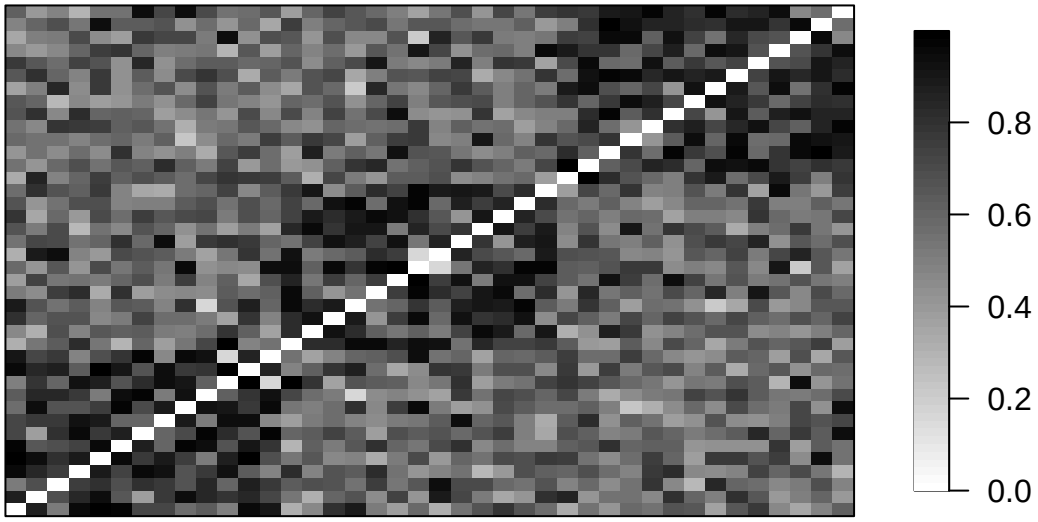
```r
## Graph

# Extract last plinks
last_plinks = tail(chains$G, n=1)[[1]]

# Criterion 1 to select the threshold (should not work very well) and assign final graph
threshold = 0.5
G_est <- matrix(0,p,p)
G_est[which(last_plinks>threshold)] = 1

#Criterion 2 to select the threshold
bfdr_select = BFDR_selection(last_plinks, tol = seq(0.1, 1, by = 0.001))

# Inspect the threshold and assign final graph
bfdr_select$best_treshold
```

```
## [1] 0.911
```

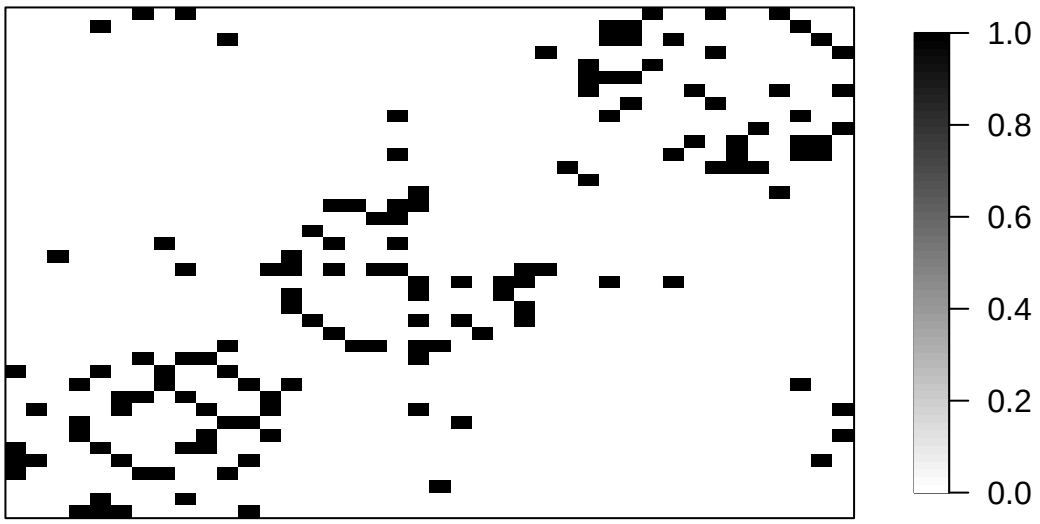
```r
G_est = bfdr_select$best_truncated_graph

### Standardized Hamming distance
SHD = sum(abs(simKG$graph - G_est)) / (p^2 - p)
SHD
```

```
## [1] 0
```

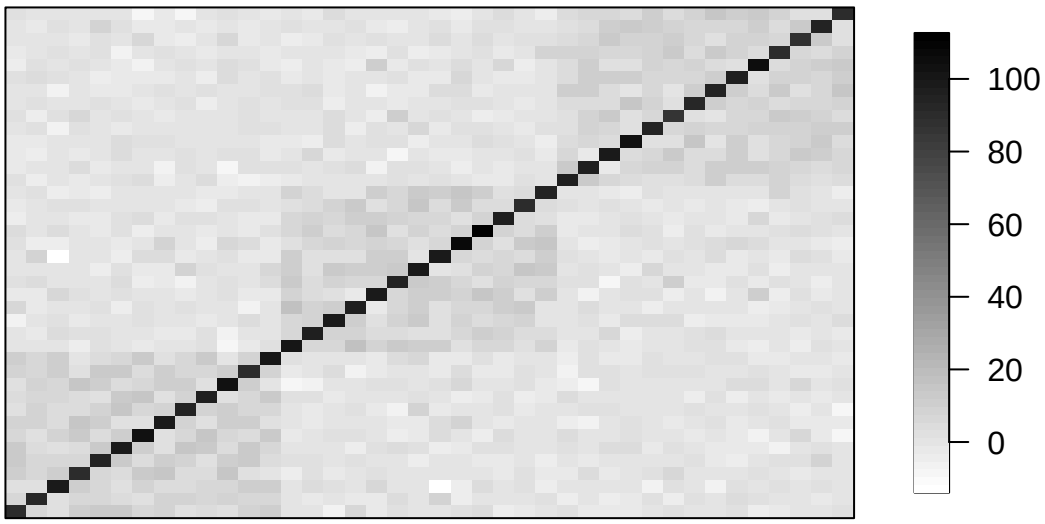
```r
### Plot estimated matrices
ACutils::ACheatmap(
  last_plinks,
  use_x11_device = F,
  horizontal = F,
  main = "Estimated plinks matrix",
  center_value = NULL,
  col.upper = "black",
  col.center = "grey50",
  col.lower = "white"
)
```



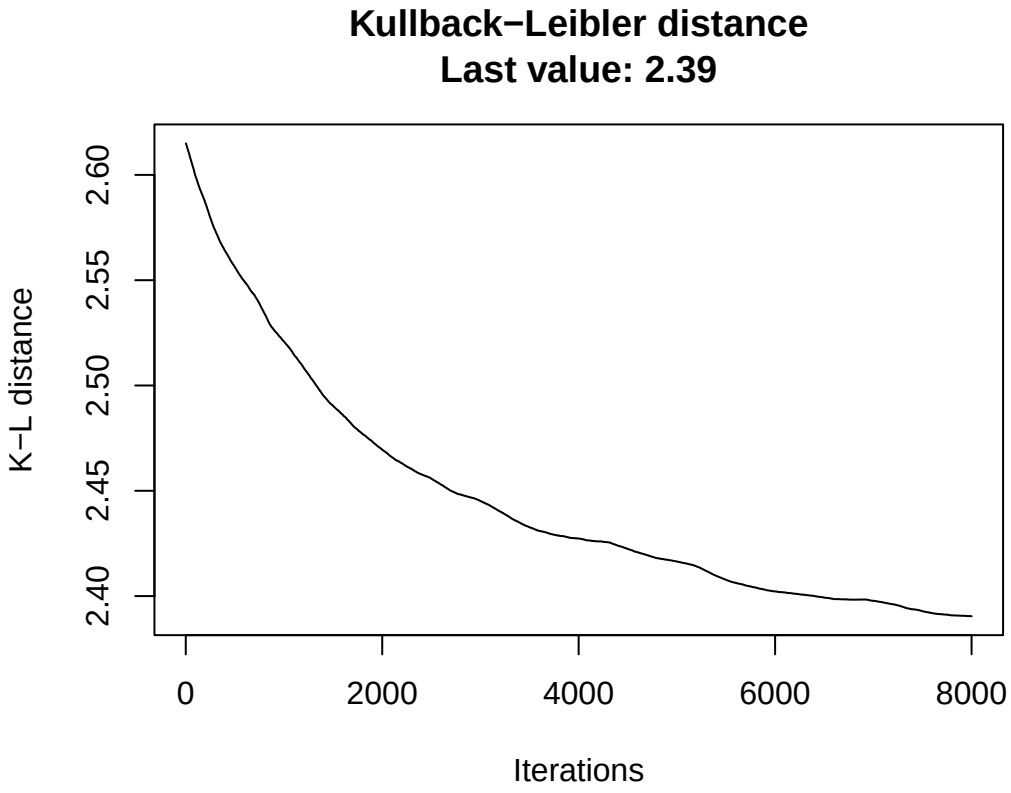

```
ACutils::ACheatmap(
  G_est,
  use_x11_device = F,
  horizontal = F,
  main = "Estimated Graph",
  center_value = NULL,
  col.upper = "black",
  col.center = "grey50",
  col.lower = "white"
)
```




```
ACutils::ACheatmap(
  tail(chains$K,n=1)[[1]],
  use_x11_device = F,
  horizontal = F,
  main = "Estimated Precision matrix",
  center_value = NULL,
  col.upper = "black",
  col.center = "grey50",
  col.lower = "white"
)
```




```
### Evolution of the Kullback-Leibler
kl_dist = do.call(rbind, lapply(chains$K, function(k) {
  ACutils::KL_dist(simKG$Prec, k)
}))

last = round(tail(kl_dist, n=1), 3)
plot(
  x = seq_along(kl_dist[2000:length(kl_dist)]),
  y = kl_dist[2000:length(kl_dist)],
  type = "n",
  xlab = "Iterations",
  ylab = "K-L distance",
  main = paste("Kullback-Leibler distance\nLast value:", last)
)
lines(x = seq_along(kl_dist[2000:length(kl_dist)]), y = kl_dist[2000:length(kl_dist)])
```




```
# non c'è burn in !!!!
```