

es1

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("utility_functions.R");
source("bulky_functions.R");
source("data_generation.R")
sourceCpp("wade.cpp")
Rcpp::sourceCpp('UpdateParamsGSL.cpp')

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

Goal

Smoothing di dati funzionali simulati, con grafo e partizione tenuti fissi.

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

# object containing the true graph G, the true precision matrix K
simKG <- readRDS("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 = matrix(rnorm(n = p*n), nrow = p, ncol = n)

# Define the starting value of mu
mu = rnorm(n=p)

# Fix tau_eps (squared)
tau_eps = 100

# Define the starting precision matrix K and graph G: in this case we start
# from the true K and true G and we will not update them
K = simKG$Prec
G = simKG$Graph

# Define the true partition with which we have generated data
rho = c(13,13,14)
z = c(rep(1,13), rep(2,13), rep(3,14))

# 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 <- TRUE
Update_Mu <- TRUE
Update_Tau <- TRUE

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

```

sigma_mu <- 100

# Define variance of the Beta
beta_sig2 = 0.2

# Compute graph density
graph_density = sum(G) / (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)
)

# 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

# initialization of parameters for set_options
sigma <- 0.5 # parameter of the prior of the partition
theta <- 1 # parameter of the prior of the partition
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

```

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=sigma,
  sigma_prior_parameters=list("a"=1,"b"=1,"c"=1,"d"=1),
  theta_prior_0=theta,
  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=FALSE,
  update_theta_prior=FALSE,
  update_weights=FALSE,
  update_partition=FALSE,
  update_graph=FALSE,
  perform_shuffle=FALSE
)

# 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

# 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: 1. Plot smoothed curves

```

# Compute the mean of Beta in order to have data_post
sum_Beta <- matrix(0, p, n)
for(i in (burn_in+1):niter){
  sum_Beta <- sum_Beta + chains$Beta[[i]]
}
mean_Beta <- sum_Beta/(niter-burn_in)
data_post <- BaseMat %*% mean_Beta

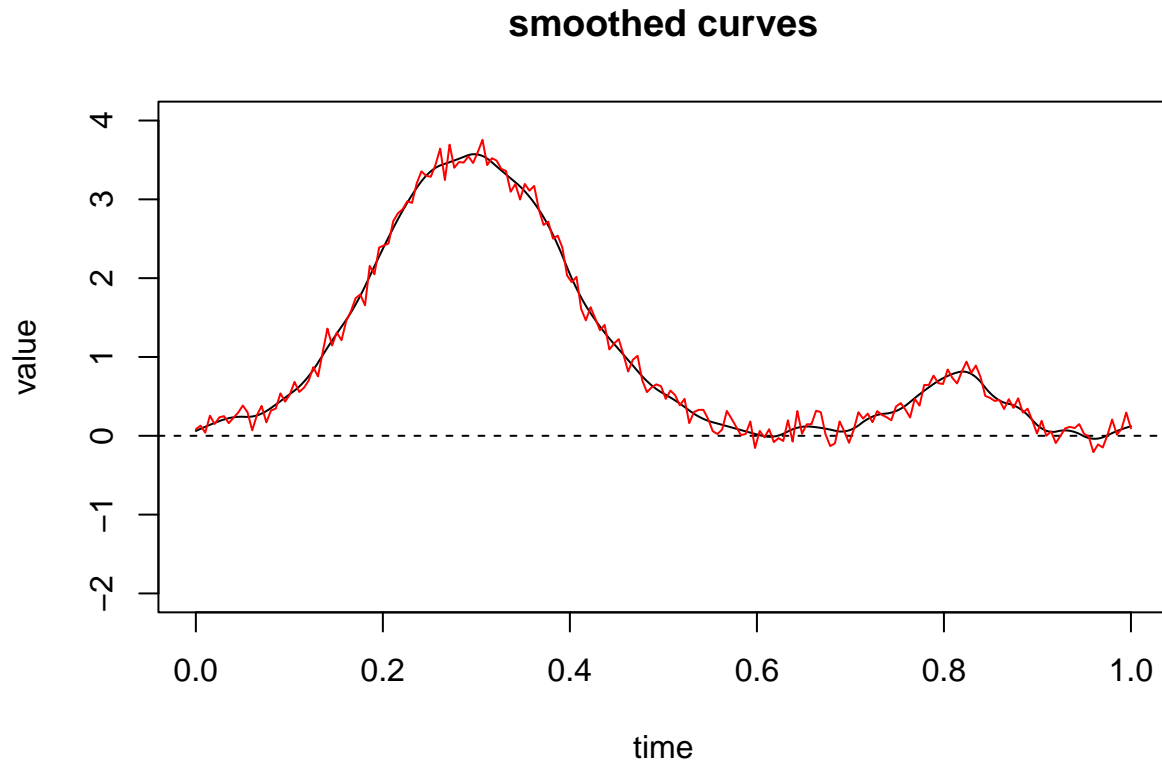
# Compute the x value, create the basis and the functional object
x <- seq(0, 1, length.out=r)
basis <- create.bspline.basis(rangeval=range(x), nbasis=40, norder=3)
data.fd <- Data2fd(y = data_post, argvals = x, basisobj = basis)

```

```
# Plot smoothed curves
plot.fd(data.fd[1,], main="smoothed curves", ylim=c(-2,4))
```

```
## [1] "done"
```

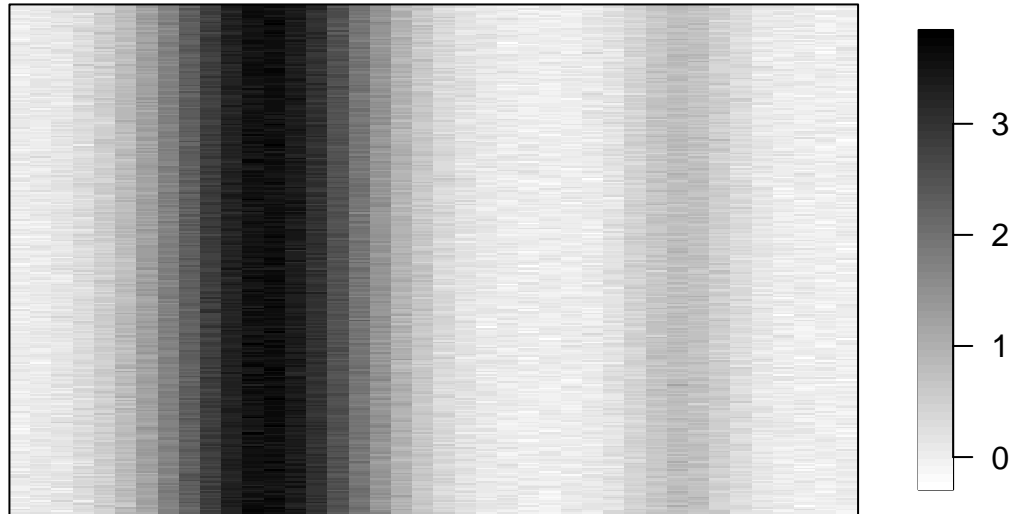
```
# plot(x, data_post[,1], type='l', ylim=c(-2,4))
lines(x,y_hat_true[1,], main="smoothed curves", col='red')
```



Useful plots: 2. Plot of the final Beta matrix

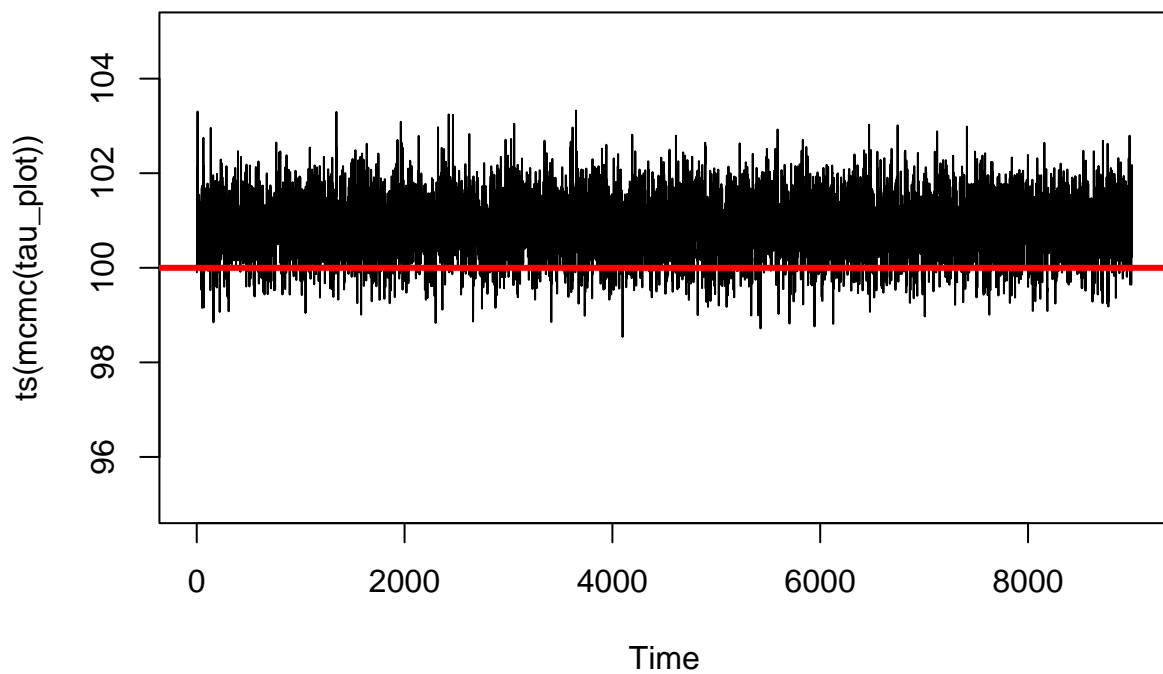
```
ACutils::ACheatmap(
  chains$Beta[[niter]],
  use_x11_device = F,
  horizontal = F,
  main = "Estimated Beta matrix",
  center_value = NULL,
  col.upper = "black",
  col.center = "grey50",
  col.lower = "white"
)
```

Estimated Beta matrix

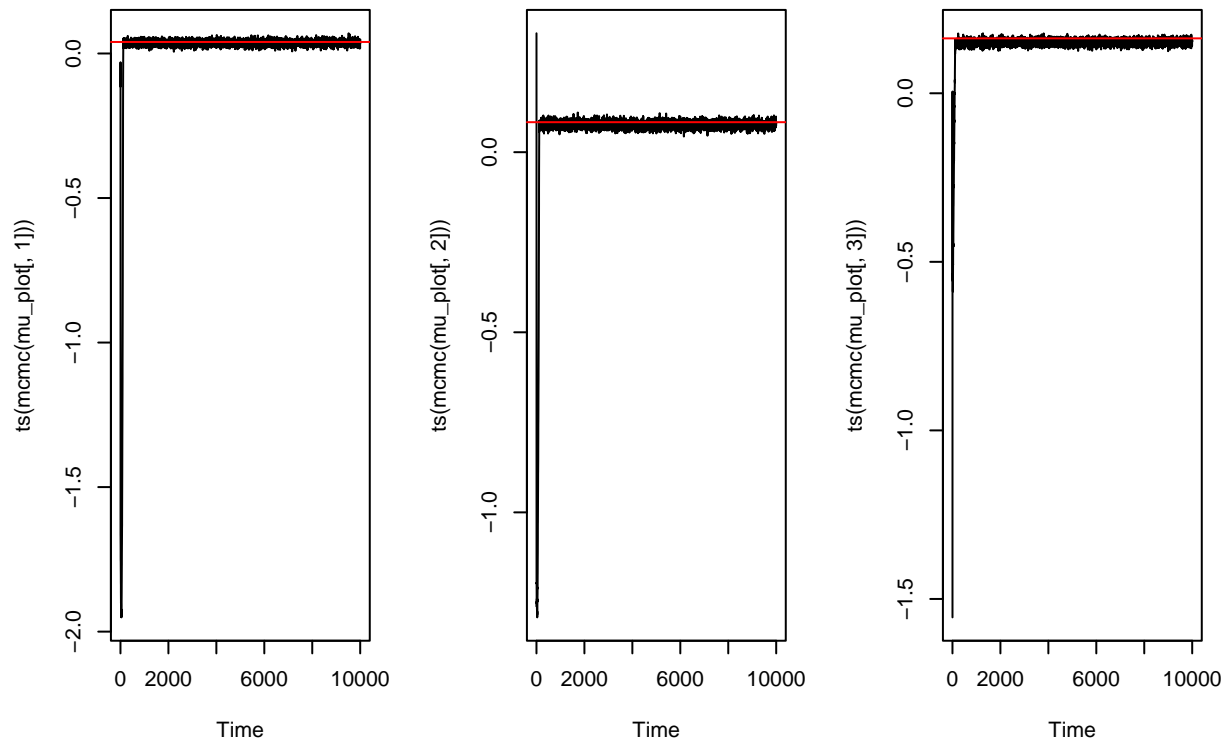


Useful plots: 2. Traceplots (tau_eps, mu, beta)

```
# tau_eps
tau_plot <- as.vector(chains$tau_eps)
tau_plot <- tau_plot[(burn_in+1):niter]
plot(ts(mcmc(tau_plot)), ylim=c(95,105))
abline(h=tau_eps, col='red', lwd=3)
```



```
# mu
mu_plot <- matrix(0, niter, p)
for(i in 1:niter){
  mu_plot[i, ] <- chains$mu[[i]]
}
par(mfrow=c(1,3))
plot(ts(mcmc(mu_plot[, 1])))
abline(h=mu_true[1], col='red')
plot(ts(mcmc(mu_plot[, 2])))
abline(h=mu_true[2], col='red')
plot(ts(mcmc(mu_plot[, 3])))
abline(h=mu_true[3], col='red')
```

```
# first element of first beta
beta1_plot <- rep(0, niter)
for(i in 1:niter){
  beta1_plot[i] <- chains$Beta[[i]][1,1]
}
# and 10th element of first beta
beta10_plot <- rep(0, niter)
for(i in 1:niter){
  beta10_plot[i] <- chains$Beta[[i]][10,1]
}

par(mfrow=c(1,2))
plot(ts(mcmc(beta1_plot)))
abline(h=beta_true[1,1], col='red')

plot(ts(mcmc(beta10_plot)))
abline(h=beta_true[1,10], col='red')
```

