Esercitazione 9

December 5, 2024

1 Denoising

In questa esercitazione vediamo come fare un denoising di alcune immagini (pulizia) e il riconoscimento di oggetti. Per questa seconda parte ci sono tecniche più evolute di quelle che vedremo, ma sono computazionalmente molto complicate

Carichiamo delle immagini che trovate anche in cartella

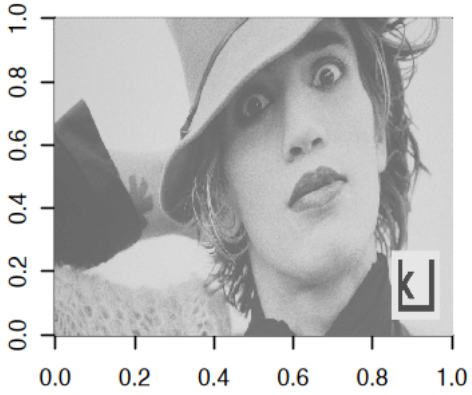
[158]:
```
library(imager)
library(jpeg)
library(ggplot2)
library(tidyverse)

# John Frusciante
data_image_all_frusciante <- readJPEG("/Users/gianlucamastrantonio/Dropbox␣
 ↪(Politecnico di Torino Staff)/Didattica/statistica computazionale/esercizi/
 ↪image/Niandra.jpeg")
# Jeff Buckley
data_image_all_buckley <- readJPEG("/Users/gianlucamastrantonio/Dropbox␣
 ↪(Politecnico di Torino Staff)/Didattica/statistica computazionale/esercizi/
 ↪image/buckley.jpeg")
# Rick
data_image_all_rick <- readJPEG("/Users/gianlucamastrantonio/Dropbox␣
 ↪(Politecnico di Torino Staff)/Didattica/statistica computazionale/esercizi/
 ↪image/rick.jpg")
```

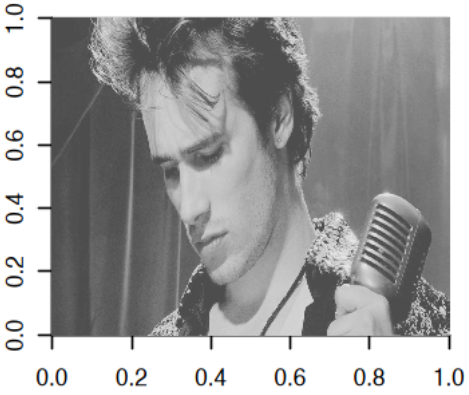
Le immagini sono caricate su tre canali (i colori), e trasformiamole in scala di grigi, così da avere un valore per ogni pixel. La classica trasformazione è

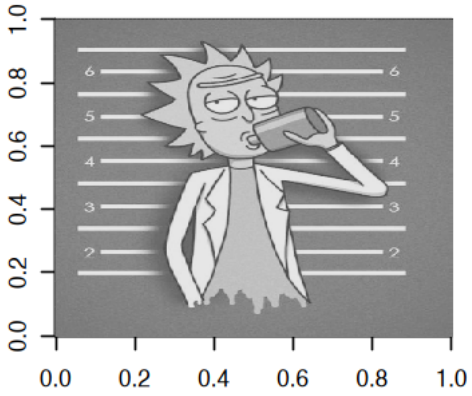
$$0.2990 * \text{rosso} + 0.5870 * \text{giallo} + 0.1140 * \text{blue}$$

[159]:
```
R <- t(data_image_all_frusciante[rev(1:dim(data_image_all_frusciante)[1]), , 1])
G <- t(data_image_all_frusciante[rev(1:dim(data_image_all_frusciante)[1]), , 2])
B <- t(data_image_all_frusciante[rev(1:dim(data_image_all_frusciante)[1]), , 3])
data_grey_image_all_frusciante <- 0.2990 * R + 0.5870 * G + 0.1140 * B



R <- t(data_image_all_buckley[rev(1:dim(data_image_all_buckley)[1]), , 1])
```

```
G <- t(data_image_all_buckley[rev(1:dim(data_image_all_buckley)[1]), , 2])
B <- t(data_image_all_buckley[rev(1:dim(data_image_all_buckley)[1]), , 3])
data_grey_image_all_buckley <- 0.2990 * R + 0.5870 * G + 0.1140 * B



R <- t(data_image_all_rick[rev(1:dim(data_image_all_rick)[1]), , 1])
G <- t(data_image_all_rick[rev(1:dim(data_image_all_rick)[1]), , 2])
B <- t(data_image_all_rick[rev(1:dim(data_image_all_rick)[1]), , 3])
data_grey_image_all_rick <- 0.2990 * R + 0.5870 * G + 0.1140 * B

par(mfrow=c(2,2))
image(data_grey_image_all_frusciante, col= grey.colors(100))
image(data_grey_image_all_buckley, col = grey.colors(100))
image(data_grey_image_all_rick, col = grey.colors(100))
par(mfrow = c(1, 1))
```







Per avere tempi computazionali accettabili, riduciamo il numero di pixele per immagine.

Aggiungiamo anche un rumore gaussiano a componenti indipendenti su ogni punto. fate attenzione che i dati appartengono a $[0, 1]$

[160]:
```
range(c(data_grey_image_all_frusciante))
range(c(data_grey_image_all_buckley))
range(c(data_grey_image_all_rick))
```

1. 0 2. 1

1. 0 2. 1

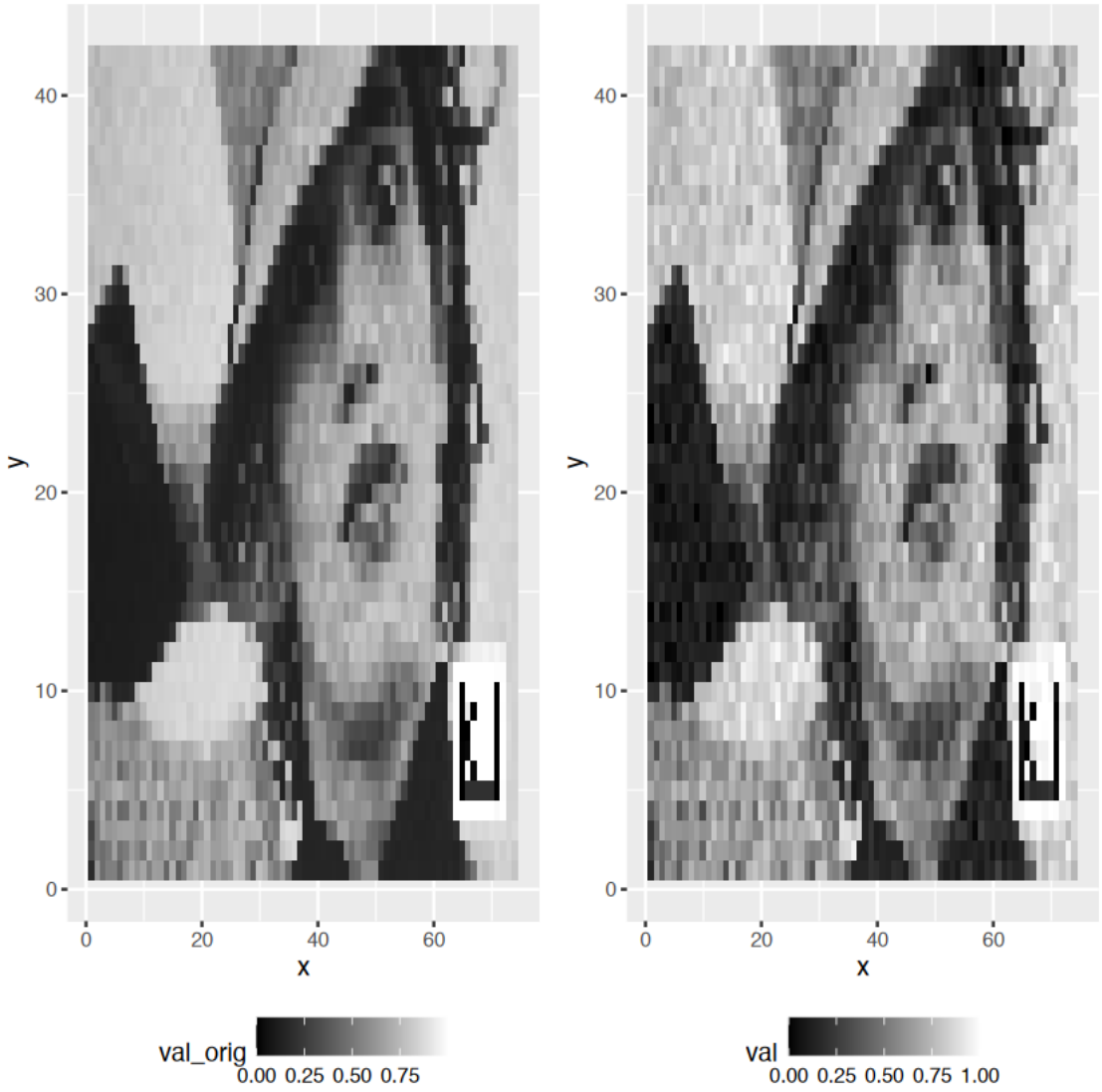
1. 0 2. 1

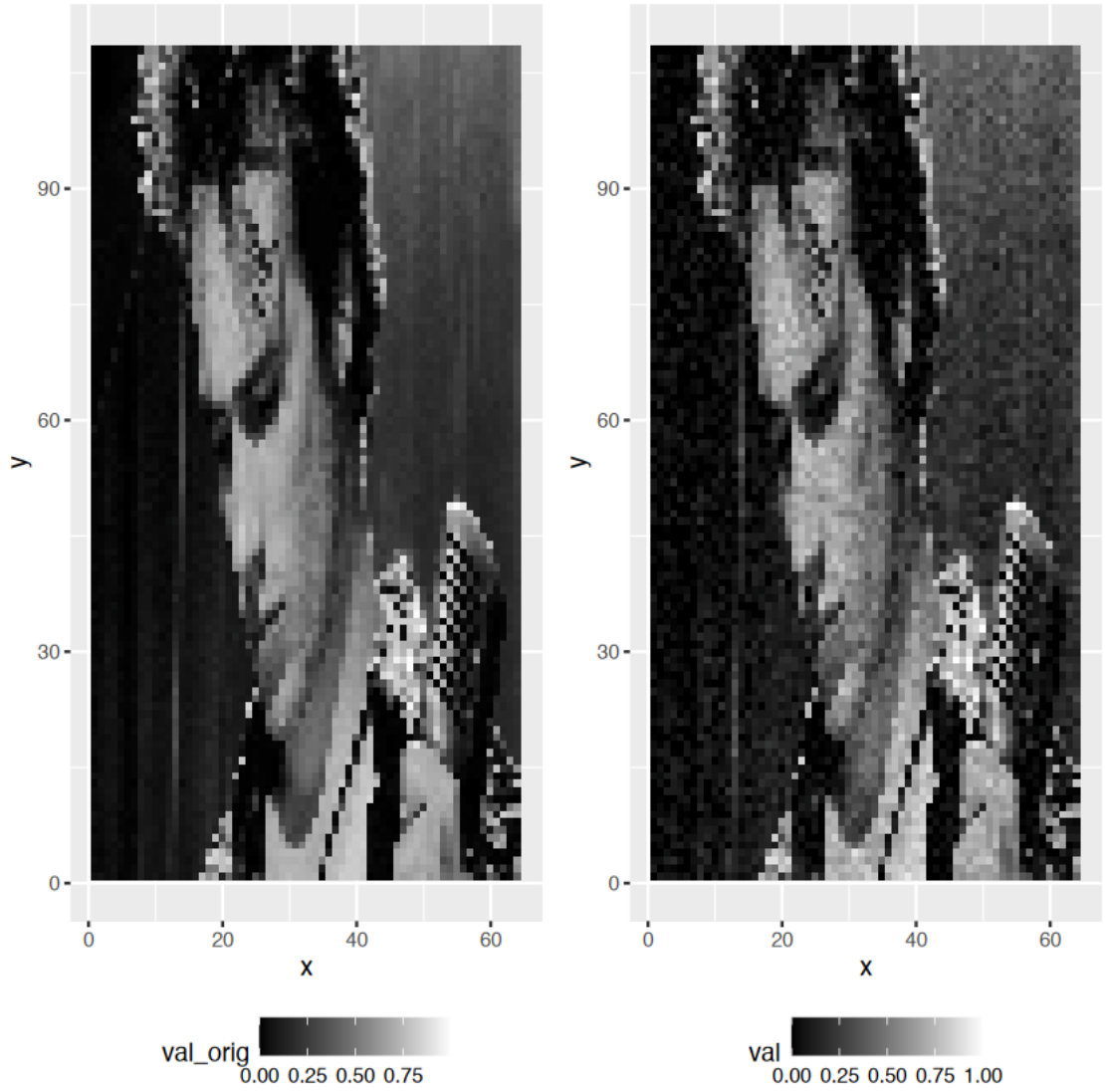
se aggiungiamo un rumore gaussiamo dobbiamo trasformare in 0 tutti i valori negativi, e in 1 tutti

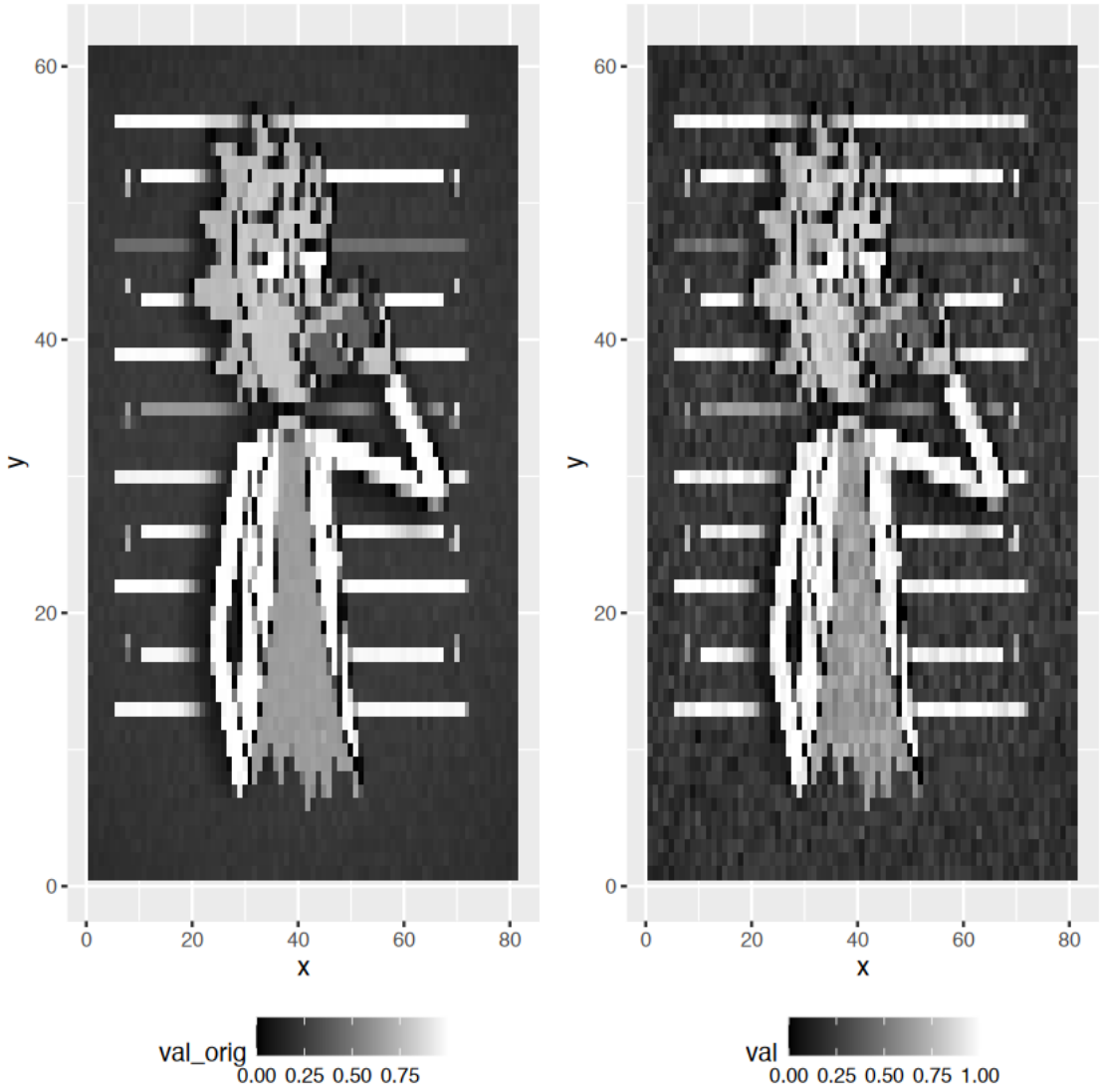
quelli superiori a 1.

[161]:
```
set.seed(1)
library(gridExtra)
# Frusciante
data_image_frusciante <- data_grey_image_all_frusciante[seq(1,␣
 ↪nrow(data_grey_image_all_frusciante), by = 15), seq(1,␣
 ↪ncol(data_grey_image_all_frusciante), by = 15)]
data_long_frusciante <- data.frame(
  x = rep(1:nrow(data_image_frusciante), times = ncol(data_image_frusciante)),
  y = rep(1:ncol(data_image_frusciante), each = nrow(data_image_frusciante)),
  val_orig = c(data_image_frusciante),
  val = c(data_image_frusciante) + rnorm(nrow(data_image_frusciante) *␣
 ↪ncol(data_image_frusciante), 0, 0.05)
)
data_long_frusciante$val[data_long_frusciante$val<=0] <- 0
data_long_frusciante$val[data_long_frusciante$val >= 1] <- 1


p1 <- data_long_frusciante %>% ggplot(aes(x = x, y = y, fill = val_orig)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white") + theme(legend.position =␣
 ↪"bottom")

p2 <- data_long_frusciante %>% ggplot(aes(x = x, y = y, fill = val)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white") + theme(legend.position =␣
 ↪"bottom")

grid.arrange(p1, p2, ncol = 2)

data_image_buckley <- data_grey_image_all_buckley[seq(1,␣
 ↪nrow(data_grey_image_all_buckley), by = 30), seq(1,␣
 ↪ncol(data_grey_image_all_buckley), by = 10)]
data_long_buckley <- data.frame(
  x = rep(1:nrow(data_image_buckley), times = ncol(data_image_buckley)),
  y = rep(1:ncol(data_image_buckley), each = nrow(data_image_buckley)),
  val_orig = c(data_image_buckley),
  val = c(data_image_buckley) + rnorm(nrow(data_image_buckley) *␣
 ↪ncol(data_image_buckley), 0, 0.05)
)
data_long_buckley$val[data_long_buckley$val <= 0] <- 0
data_long_buckley$val[data_long_buckley$val >= 1] <- 1

p1 <- data_long_buckley %>% ggplot(aes(x = x, y = y, fill = val_orig)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white") +
```

```
  theme(legend.position = "bottom")

p2 <- data_long_buckley %>% ggplot(aes(x = x, y = y, fill = val)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white") +
  theme(legend.position = "bottom")

grid.arrange(p1, p2, ncol = 2)

data_image_rick <- data_grey_image_all_rick[seq(1,␣
↪nrow(data_grey_image_all_rick), by = 5), seq(1,␣
↪ncol(data_grey_image_all_rick), by = 8)]
data_long_rick <- data.frame(
  x = rep(1:nrow(data_image_rick), times = ncol(data_image_rick)),
  y = rep(1:ncol(data_image_rick), each = nrow(data_image_rick)),
  val_orig = c(data_image_rick),
  val = c(data_image_rick) + rnorm(nrow(data_image_rick) *␣
↪ncol(data_image_rick), 0, 0.05)
)
data_long_rick$val[data_long_rick$val <= 0] <- 0
data_long_rick$val[data_long_rick$val >= 1] <- 1

p1 <- data_long_rick %>% ggplot(aes(x = x, y = y, fill = val_orig)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white") +
  theme(legend.position = "bottom")

p2 <- data_long_rick %>% ggplot(aes(x = x, y = y, fill = val)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white") +
  theme(legend.position = "bottom")

grid.arrange(p1, p2, ncol = 2)
```


40
30
20
10
0
y
0
20
40
60
x
val_orig
0.00 0.25 0.50 0.75
40
30
20
10
0
y
0
20
40
60
x
val
0.00 0.25 0.50 0.75 1.00


90
60
30
0
y
0
20
40
60
x
val_orig
0.00 0.25 0.50 0.75
90
60
30
0
y
0
20
40
60
x
val
0.00 0.25 0.50 0.75 1.00



Immaginiamo che le immagini siano i nostri dati e siamo interessati e pulire le immagini (renderle meno rumorose). Per semplicità ne scegliamo 1.

[162]:
```
data_long <- data_long_frusciante
data_image <- data_image_frusciante
```

Indichiamo con

$$X_{i,j}$$

il valore sul pixel (i, j), con $X_{i,j} \in [0, 1]$. Per l'MCMC dobbiamo delle volte lavorare con vettori e quindi quanto uso un solo indice, tipo X_l, indico l-esimo valore del vettore che contiene tutti i punti griglia (vale per anche le altre variabili)

La distribuzione dovrebbe avere masse di probabilità su zero e 1. Quello che possiamo fare è

introdurre una variabile $Y_{i,j}$ che definita come

$$Y_{i,j} = X_{i,j} \text{ if } X_{i,j} \in (0,1)$$

$$Y_{i,j} < 0 \text{ if } X_{i,j} = 0$$

$$Y_{i,j} > 1 \text{ if } X_{i,j} = 1$$

Questo ci permette di lavorare con una variabile in $\mathbb{R}$. Naturalmente parte di Y non lo conosciamo e va stimato/campionato. Notate che

$$P(X_{i,j} = 0) = P(Y_{i,j} < 0)$$

$$P(X_{i,j} = 1) = P(Y_{i,j} > 1)$$

Il modello che vorremmo implementare è

$$Y_{i,j}|w_{i,j} \sim N(\mu + \beta w_{i,j}, \sigma^2)$$

con

$$\mathbf{w} \sim ICAR(\mathbf{H})$$

dove (i,j) sono le coordinate di riga e colonna dei pixel, e $ICAR(H)$ è un modello ICAR con matrice di vicinanza pari a $\mathbf{H}$. La matrice H in posizione j,i continene un 1 se e solo se i e j sono vicine (le osservazioni non sono vicine di se stesse). Data la matrice di vicini, la densità congiunta è pari a

$$f(\mathbf{w}) \propto \exp\left(-\mathbf{w}^T(\mathbf{D}_w - \mathbf{H})\mathbf{w}\right)$$

dove $\mathbf{D}_w$ è una matrice diagonale che sulla diagonale $i-$esima ha la somma degli elementi della riga i-esima di $\mathbf{H}$

la a posteriori di

$$\mu + \beta w_{i,j}$$

è quello che ci interessa visto che è la versione smooth dei dati.

Notate che se definisco

$$w^*_{i,j} = \beta w_{i,j}$$

ho che β^* è la varianza del processo. con un ICAR si preferisce scriverla così perchè non essendo una distribuzione proper non è sempre charo come scrivere la costante di normalizzazione, che dipende dalla varianza, visto che non integra a 1.

La prima cosa che dobbiamo fare è definire i vicini. Sotto trovate diverse funzioni che calcolano i vicini di ogni osservazione, assumendo che questi siano al più 4, o 8 o 20.

[163]:
```
find_neighbors_4 <- function(row, col, n, m) {
  neighbors <- list(
    c(row - 1, col),
    c(row + 1, col),
    c(row, col - 1),
    c(row, col + 1)
  )
```

```r
  # Filter out neighbors that are outside the grid boundaries
  neighbors <- Filter(function(cell) {
    cell[1] >= 1 && cell[1] <= n && cell[2] >= 1 && cell[2] <= m
  }, neighbors)

  return(neighbors)
}
data_neigh_4 <- list()
for (i in 1:nrow(data_long)) {
  coord_neigh <- find_neighbors_4(data_long$x[i], data_long$y[i],␣
 ↪nrow(data_image), ncol(data_image))

  w <- rep(NA, length(coord_neigh))
  for (i_neigh in 1:length(coord_neigh))
  {
    w[i_neigh] <- which(
      data_long$x == coord_neigh[[i_neigh]][1] &
        data_long$y == coord_neigh[[i_neigh]][2]
    )
  }
  data_neigh_4[[i]] <- w
}


find_neighbors_8 <- function(row, col, n, m) {
  neighbors <- list(
    c(row - 1, col),
    c(row + 1, col),
    c(row, col - 1),
    c(row, col + 1),
    c(row - 1, col - 1),
    c(row - 1, col + 1),
    c(row + 1, col - 1),
    c(row + 1, col + 1)
  )

  # Filter out neighbors that are outside the grid boundaries
  neighbors <- Filter(function(cell) {
    cell[1] >= 1 && cell[1] <= n && cell[2] >= 1 && cell[2] <= m
  }, neighbors)

  return(neighbors)
}

data_neigh_8 <- list()
for (i in 1:nrow(data_long)) {
```

```
  coord_neigh <- find_neighbors_8(data_long$x[i], data_long$y[i],␣
 ↪nrow(data_image), ncol(data_image))

  w <- rep(NA, length(coord_neigh))
  for (i_neigh in 1:length(coord_neigh))
  {
    w[i_neigh] <- which(
      data_long$x == coord_neigh[[i_neigh]][1] &
        data_long$y == coord_neigh[[i_neigh]][2]
    )
  }
  data_neigh_8[[i]] <- w
}


find_neighbors_20 <- function(row, col, n, m) {
  neighbors <- list(
    c(row - 1, col),
    c(row - 2, col),
    c(row + 1, col),
    c(row + 2, col),
    c(row, col - 1),
    c(row, col - 2),
    c(row, col + 1),
    c(row, col + 2),
    c(row - 1, col - 1),
    c(row - 1, col - 2),
    c(row - 2, col - 1),
    c(row - 1, col + 1),
    c(row - 1, col + 2),
    c(row - 2, col + 1),
    c(row + 1, col - 1),
    c(row + 1, col - 2),
    c(row + 2, col - 1),
    c(row + 1, col + 1),
    c(row + 1, col + 2),
    c(row + 2, col + 1)
  )

  # Filter out neighbors that are outside the grid boundaries
  neighbors <- Filter(function(cell) {
    cell[1] >= 1 && cell[1] <= n && cell[2] >= 1 && cell[2] <= m
  }, neighbors)

  return(neighbors)
}
```

```
data_neigh_20 <- list()
for (i in 1:nrow(data_long)) {
  coord_neigh <- find_neighbors_20(data_long$x[i], data_long$y[i],␣
 ↪nrow(data_image), ncol(data_image))

  w <- rep(NA, length(coord_neigh))
  for (i_neigh in 1:length(coord_neigh))
  {
    w[i_neigh] <- which(
      data_long$x == coord_neigh[[i_neigh]][1] &
        data_long$y == coord_neigh[[i_neigh]][2]
    )
  }
  data_neigh_20[[i]] <- w
}
```

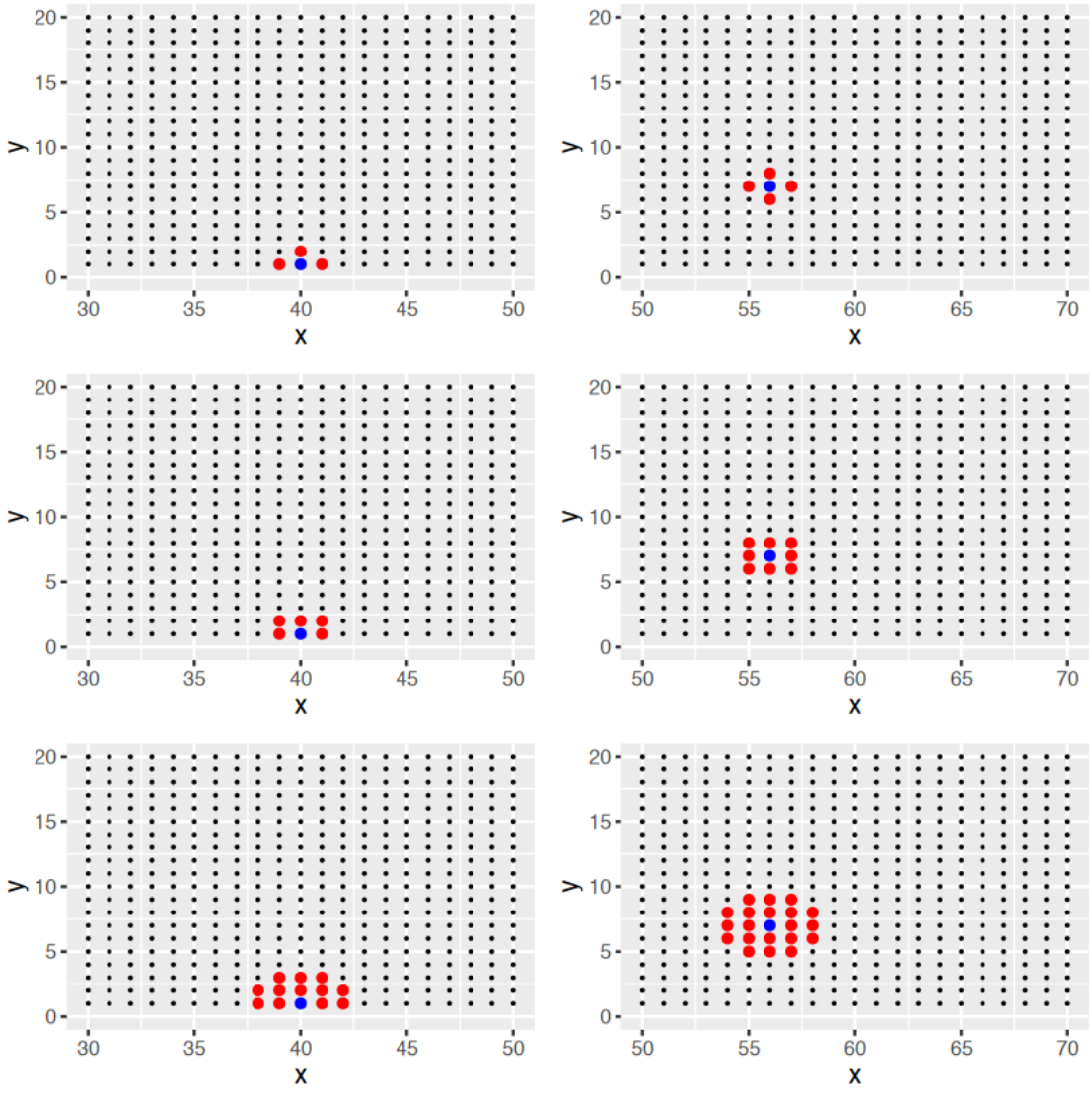
Vediamo i 3 tipi di vicini per un punto specifico

[182]:
```
i1 = 40
i2 <- 500

p4_1 <- data_long %>% ggplot(aes(x = x, y = y)) +
  geom_point(size = 0.001) + geom_point(data = data_long[data_neigh_4[[i1]],],␣
 ↪aes(x = x, y = y), color = "red") +
  geom_point(data = data_long[data_long$x == data_long$x[i1] & data_long$y ==␣
 ↪data_long$y[i1], ], aes(x = x, y = y), color = "blue") +
    xlim(30, 50) +
    ylim(0, 20)
p4_2 <- data_long %>% ggplot(aes(x = x, y = y)) +
  geom_point(size = 0.001) +
  geom_point(data = data_long[data_neigh_4[[i2]], ], aes(x = x, y = y), color =␣
 ↪"red") +
  geom_point(data = data_long[data_long$x == data_long$x[i2] & data_long$y ==␣
 ↪data_long$y[i2], ], aes(x = x, y = y), color = "blue") +
    xlim(50, 70) +
    ylim(0, 20)

p8_1 <- data_long %>% ggplot(aes(x = x, y = y)) +
  geom_point(size = 0.001) +
  geom_point(data = data_long[data_neigh_8[[i1]], ], aes(x = x, y = y), color =␣
 ↪"red") +
  geom_point(data = data_long[data_long$x == data_long$x[i1] & data_long$y ==␣
 ↪data_long$y[i1], ], aes(x = x, y = y), color = "blue") +
  xlim(30, 50) +
  ylim(0, 20)
p8_2 <- data_long %>% ggplot(aes(x = x, y = y)) +
  geom_point(size = 0.001) +
```

```
  geom_point(data = data_long[data_neigh_8[[i2]], ], aes(x = x, y = y), color =␣
 ↪"red") +
  geom_point(data = data_long[data_long$x == data_long$x[i2] & data_long$y ==␣
 ↪data_long$y[i2], ], aes(x = x, y = y), color = "blue") +
  xlim(50, 70) +
  ylim(0, 20)
p20_1 <- data_long %>% ggplot(aes(x = x, y = y)) +
  geom_point(size = 0.001) +
  geom_point(data = data_long[data_neigh_20[[i1]], ], aes(x = x, y = y), color␣
 ↪= "red") +
  geom_point(data = data_long[data_long$x == data_long$x[i1] & data_long$y ==␣
 ↪data_long$y[i1], ], aes(x = x, y = y), color = "blue") +
  xlim(30, 50) +
  ylim(0, 20)
p20_2 <- data_long %>% ggplot(aes(x = x, y = y)) +
  geom_point(size = 0.001) +
  geom_point(data = data_long[data_neigh_20[[i2]], ], aes(x = x, y = y), color␣
 ↪= "red") +
  geom_point(data = data_long[data_long$x == data_long$x[i2] & data_long$y ==␣
 ↪data_long$y[i2], ], aes(x = x, y = y), color = "blue") +
  xlim(50, 70) +
  ylim(0, 20)

grid.arrange(p4_1, p4_2, p8_1,p8_2, p20_1, p20_2, ncol=2)


#i = 100

#data_long %>% ggplot(aes(x = x, y = y)) +
#  geom_point(size = 0.001) +
#  geom_point(data = data_long[data_neigh[[i]], ], aes(x = x, y = y), color =␣
 ↪"red") +
#  geom_point(data = data_long[data_long$x == data_long$x[i] & data_long$y ==␣
 ↪data_long$y[i], ], aes(x = x, y = y), color = "blue")
```

```
Warning message:
"Removed 2688 rows containing missing values or values outside the
scale range
(`geom_point()`)."
Warning message:
"Removed 2688 rows containing missing values or values outside the
scale range
(`geom_point()`)."
Warning message:
"Removed 2688 rows containing missing values or values outside the
scale range
(`geom_point()`)."
```

```
Warning message:
"Removed 2688 rows containing missing values or values outside the
scale range
(`geom_point()`)."
Warning message:
"Removed 2688 rows containing missing values or values outside the
scale range
(`geom_point()`)."
Warning message:
"Removed 2688 rows containing missing values or values outside the
scale range
(`geom_point()`)."
```



Adesso che abbiamo i vicini, scegliamo una struttura e calcoliamo la matrice di precisione del

processo. Più vicini utilizzate e più smooth sarà il risultato

[165]:
```
data_neigh <- data_neigh_20
Hmat <- matrix(0, nrow = nrow(data_long), ncol = nrow(data_long))
Dh <- matrix(0, nrow = nrow(data_long), ncol = nrow(data_long))
for(i in 1:nrow(data_long)){

  Hmat[i, data_neigh[[i]]] <- 1
  Dh[i, i] <- sum(Hmat[i, ])

}
Sigma_inv <- Dh - Hmat
```

Intanto verifichiamo che la matrice non è a rango pieno (questo è un processo improper). Però se la usate nel secondo livello di un modello è possibile farlo (si dimostra che la a posteriori è proper)

[166]:
```
table(rowSums(Sigma_inv))
```

```
   0
3108
```

tutte le righe sommano a 1.

Per implementare il modello dobbiamo definire le prior, e usiamo

$$\beta \sim N(0, 10)$$

$$\mu \sim N(0, 10)$$

$$\sigma^2 \sim IG(1, 1)$$

Nell' 'MCMC, oltre a queste tree variabili (che hanno un update come standard parametri di una normale) dobbiamo campionare sia $\mathbf{w}$ che i valori di Y associate a X uguali a 0 e 1.

La full conditional di $Y_{i,j} = c$, con $c < 0$, e $X_{i,j} = 0$ si può calcolare come

$$f(Y_{i,j} = c|X_{i,j} = 0, w_{i,j}) = \frac{f(Y_{i,j} = c, X_{i,j} = 0|w_{i,j})}{f(X_{i,j} = 0|w_{i,j})} = \frac{f(Y_{i,j} = c|w_{i,j})}{f(X_{i,j} = 0|w_{i,j})} =$$

$$\frac{\phi(Y_{i,j}|\mu + \beta w_{i,j}, \sigma^2)}{\Phi(0|\mu + \beta w_{i,j}, \sigma^2)}$$

dove $\phi()$ e $\Phi()$ sono densità e comulata di una normale. Con calcoli simili abbiamo

$$f(Y_{i,j} = c|X_{i,j} = 1, w_{i,j}) = \frac{\phi(Y_{i,j}|\mu + \beta w_{i,j}, \sigma^2)}{1 - \Phi(1|\mu + \beta w_{i,j}, \sigma^2)}$$

se $c > 1$. In entrambi i casi la distribuzione è una normale troncata.

Per l'update di $\mathbf{w}$ potremmo campionarle tutte insieme, ma questo ci richiede di invertire una matrice molto grande. Possiamo però sfruttare il fatto che conosciamo la matrice di precisione

$$\Lambda = \mathbf{D}_w - \mathbf{H}$$

Quando calcoliamo la forma quadratica

$$\mathbf{w}^T \Lambda \mathbf{w}$$

abbiamo che gli unici termini che dipendono da $w_l = [\mathbf{w}]_l$, sono

$$w_l^2 [\Lambda]_{l,l} + 2 w_l [\Lambda]_{l,-l} \mathbf{w}_{-l}$$

dove ricordiamo che gli elementi di $[\Lambda]_{l,\cdot}$ sono tutti zero tranne per i vicini di l, che ha elementi pari a -1. Visto che nel kernel di una normale abbiamo che la forma quadratica univariata deve essere

$$w_l^2 V_c^{-1} - 2 w_l V_c^{-1} M_c$$

dove V_ce M_c sono media e varianza condizionata, possiamo facilmente trovare i loro valori. Quindi simulare un w alla volta non ci porta al calcolo di inverse

scriviamo il codice e otteniamo campioni (uso molte iterazioni per avere catene a convergenza, ma voi potete usarle di meno che altrimenti ci mette troppp)

[167]:
```r
set.seed(1)
# Load necessary library
library(truncnorm) # For truncated normal distributions

# Set parameters for MCMC simulation
niter <- 5000 # Total number of iterations
burnin <-3000 # Number of burn-in iterations
thin <- 2 # Thinning interval

# Identify indices where values in 'data_long$val' are 0 or 1
w_zero <- which(data_long$val == 0) # Indices of zeros
w_one <- which(data_long$val == 1) # Indices of ones

# Calculate the number of samples to save after burn-in and thinning
sample_to_save <- floor((niter - burnin) / thin)

# Prior hyperparameters
prior_par_mean_normal <- 0 # Mean for normal prior
prior_par_var_normal <- 10 # Variance for normal prior
prior_par_1_sigma2 <- 1 # Shape parameter for inverse-gamma prior
prior_par_2_sigma2 <- 1 # Scale parameter for inverse-gamma prior

# Initialize MCMC parameters
mu_mcmc <- 0 # Mean parameter
beta_mcmc <- 0.2 # Regression coefficient
sigma2_mcmc <- 0.5 # Variance of the errors
gp_mcmc <- matrix(0, nrow = nrow(data_long), ncol = 1) # Latent Gaussian process

# Matrices to store MCMC samples
mu_out <- matrix(NA, ncol = sample_to_save, nrow = 1) # Store samples of mu
beta_out <- matrix(NA, ncol = sample_to_save, nrow = 1) # Store samples of beta
```

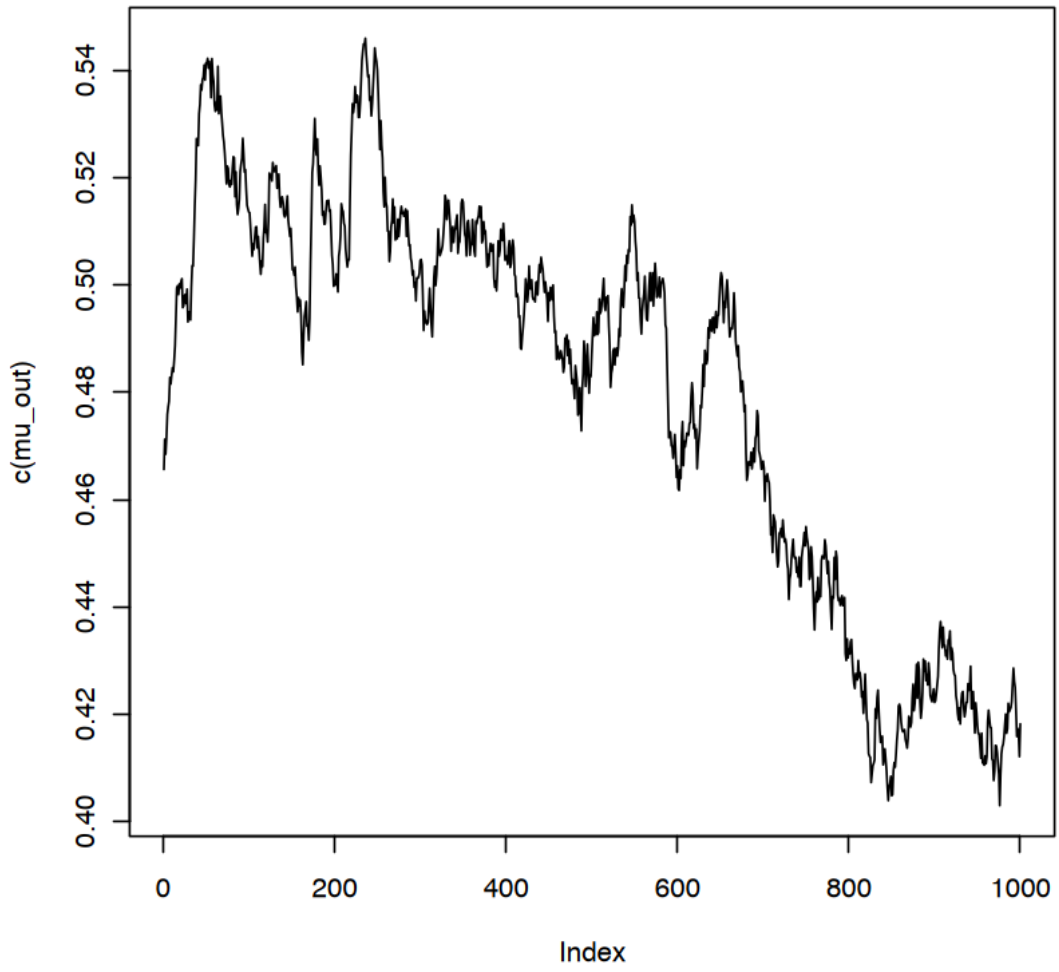
```r
sigma2_out <- matrix(NA, ncol = sample_to_save, nrow = 1) # Store samples of␣
 ↪sigma^2
gp_out <- matrix(NA, ncol = sample_to_save, nrow = nrow(data_long)) # Store GP␣
 ↪samples
mean_out <- matrix(NA, ncol = sample_to_save, nrow = nrow(data_long)) # Store␣
 ↪posterior means

# Initialize latent variable y
y_latent <- data_long$val # Start with observed values

# Update y_latent for observed zeros and ones using truncated normal sampling
if (length(w_one) > 0) {
  y_latent[w_one] <- rtruncnorm(length(w_one),
    a = 1, b = Inf,
    mean = mu_mcmc + beta_mcmc * gp_mcmc[w_one],
    sd = sqrt(sigma2_mcmc)
  )
}
if (length(w_zero) > 0) {
  y_latent[w_zero] <- rtruncnorm(length(w_zero),
    a = -Inf, b = 0,
    mean = mu_mcmc + beta_mcmc * gp_mcmc[w_zero],
    sd = sqrt(sigma2_mcmc)
  )
}

# Number of data points
n <- length(data_long$val)

# Set initial value for burn-in counter
app <- burnin

# Begin MCMC sampling
for (save_iter in 1:sample_to_save) {
  for (imcmc in 1:app) {
    # Update sigma2 using inverse-gamma posterior
    sigma2_mcmc <- 1 / rgamma(1,
      shape = n / 2 + prior_par_1_sigma2,
      rate = sum((y_latent - mu_mcmc - beta_mcmc * gp_mcmc)^2) / 2 +␣
 ↪prior_par_2_sigma2
    )

    # Update beta using normal posterior
    post_var <- 1 / (sum(gp_mcmc^2) / sigma2_mcmc + 1 / prior_par_var_normal)
    post_mean <- post_var * (sum(gp_mcmc * (y_latent - mu_mcmc)) / sigma2_mcmc +
      prior_par_mean_normal / prior_par_var_normal)
    beta_mcmc <- rnorm(1, mean = post_mean, sd = sqrt(post_var))
```

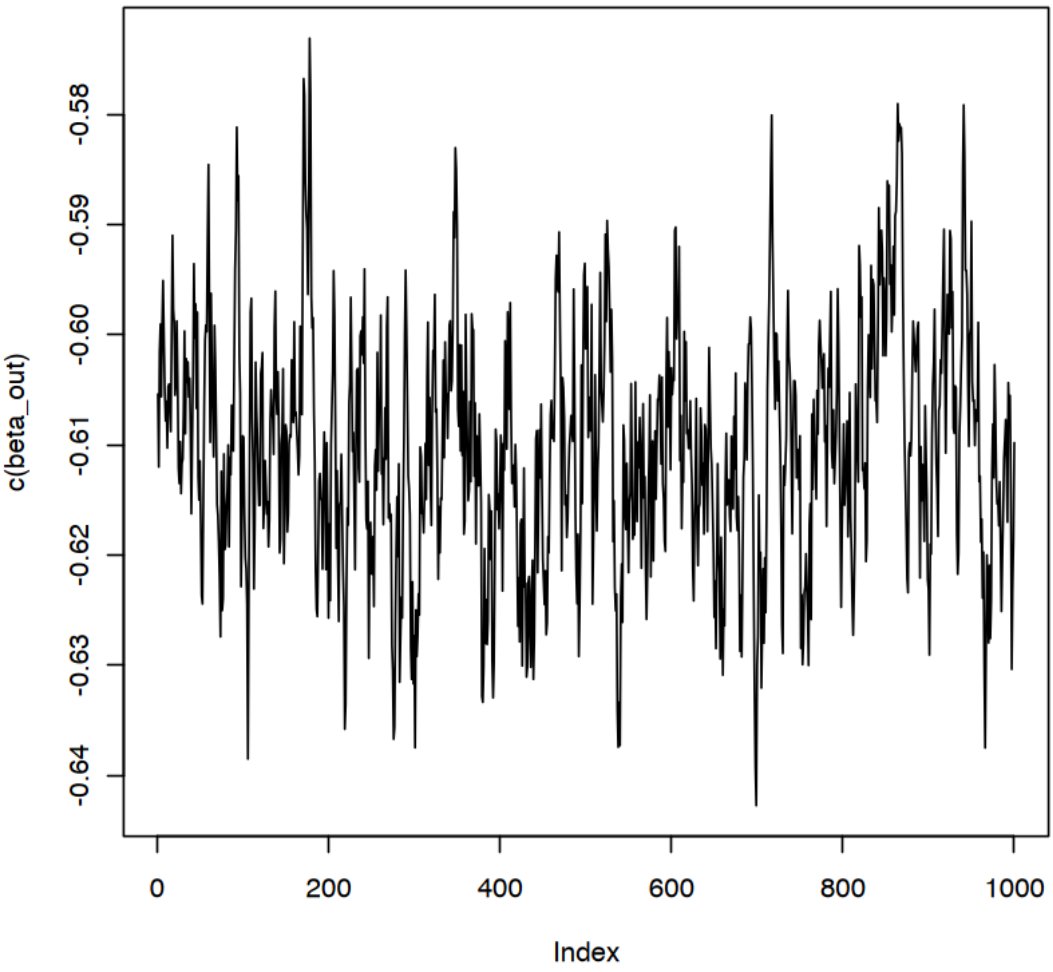
```
  # Update mu using normal posterior
  post_var <- 1 / (n / sigma2_mcmc + 1 / prior_par_var_normal)
  post_mean <- post_var * (sum(y_latent - beta_mcmc * gp_mcmc) / sigma2_mcmc +
    prior_par_mean_normal / prior_par_var_normal)
  mu_mcmc <- rnorm(1, mean = post_mean, sd = sqrt(post_var))

  # Update y_latent
  if (length(w_one) > 0) {
    y_latent[w_one] <- rtruncnorm(length(w_one),
      a = 1, b = Inf,
      mean = mu_mcmc + beta_mcmc * gp_mcmc[w_one],
      sd = sqrt(sigma2_mcmc)
    )
  }
  if (length(w_zero) > 0) {
    y_latent[w_zero] <- rtruncnorm(length(w_zero),
      a = -Inf, b = 0,
      mean = mu_mcmc + beta_mcmc * gp_mcmc[w_zero],
      sd = sqrt(sigma2_mcmc)
    )
  }

  # Update latent Gaussian process gp_mcmc
  for (i in 1:n) {
    neigh <- data_neigh[[i]] # Neighboring indices for conditional␣
↪distribution
    cond_var <- 1 / Sigma_inv[i, i] # Conditional variance
    cond_mean <- -(1 / Sigma_inv[i, i]) * Sigma_inv[i, neigh] %*%␣
↪gp_mcmc[neigh] # Conditional mean

    post_var <- 1 / (beta_mcmc^2 / sigma2_mcmc + 1 / cond_var) # Posterior␣
↪variance
    post_mean <- post_var * (beta_mcmc * (y_latent[i] - mu_mcmc) /␣
↪sigma2_mcmc +
      cond_mean / cond_var) # Posterior mean
    gp_mcmc[i] <- rnorm(1, mean = post_mean, sd = sqrt(post_var)) # Sample␣
↪from posterior
  }
}

# Update burn-in counter for subsequent iterations
app <- thin

# Save current samples
mu_out[save_iter] <- mu_mcmc
beta_out[save_iter] <- beta_mcmc
```

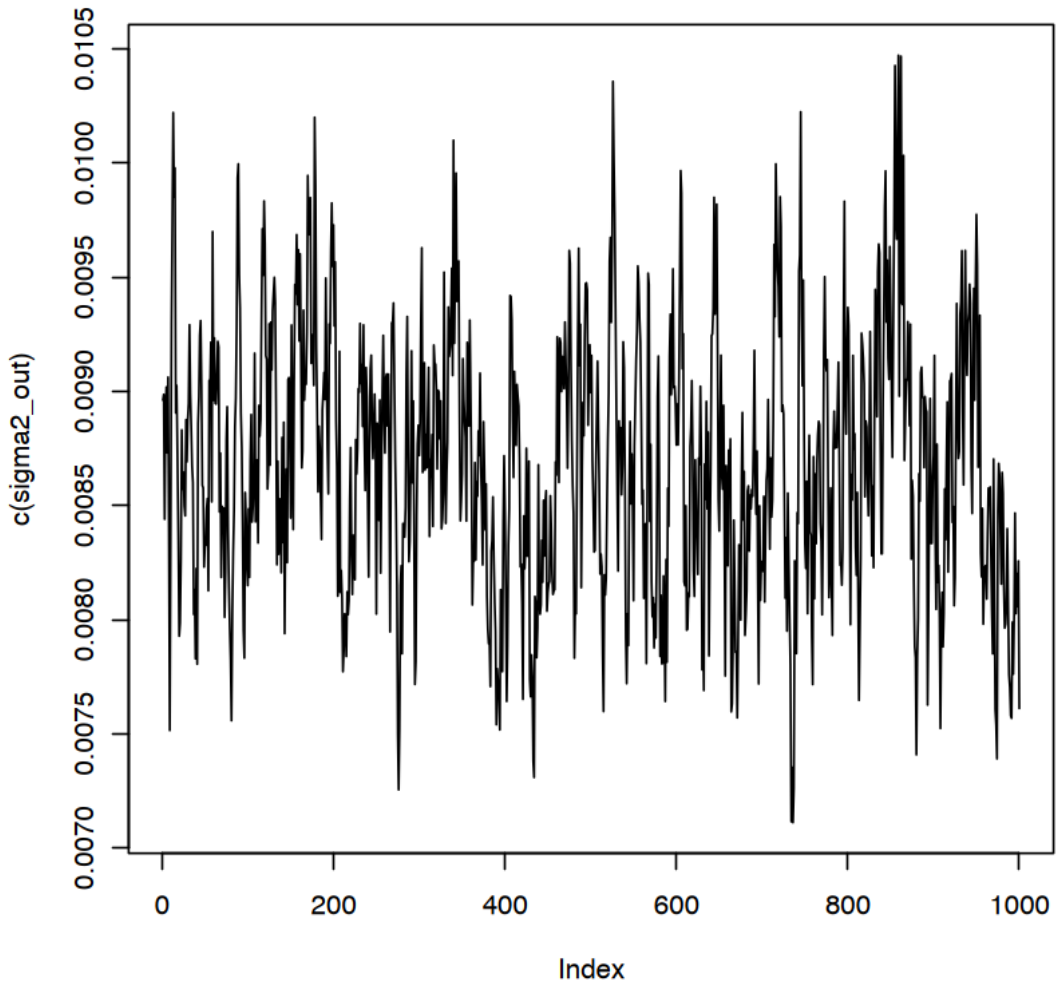
```
  sigma2_out[save_iter] <- sigma2_mcmc
  gp_out[, save_iter] <- gp_mcmc
  mean_out[, save_iter] <- mu_mcmc + beta_mcmc * gp_mcmc
  mean_out[which(mean_out[, save_iter] < 0), save_iter] <- 0
  mean_out[which(mean_out[, save_iter] > 1), save_iter] <- 1
}
```

Vediamo le catene dei parametri

[168]:
```
plot(c(mu_out), type="l")
plot(c(beta_out), type = "l")
plot(c(sigma2_out), type = "l")
```




c(beta_out)
-0.58
-0.59
-0.60
-0.61
-0.62
-0.63
-0.64
0
200
400
600
800
1000
Index

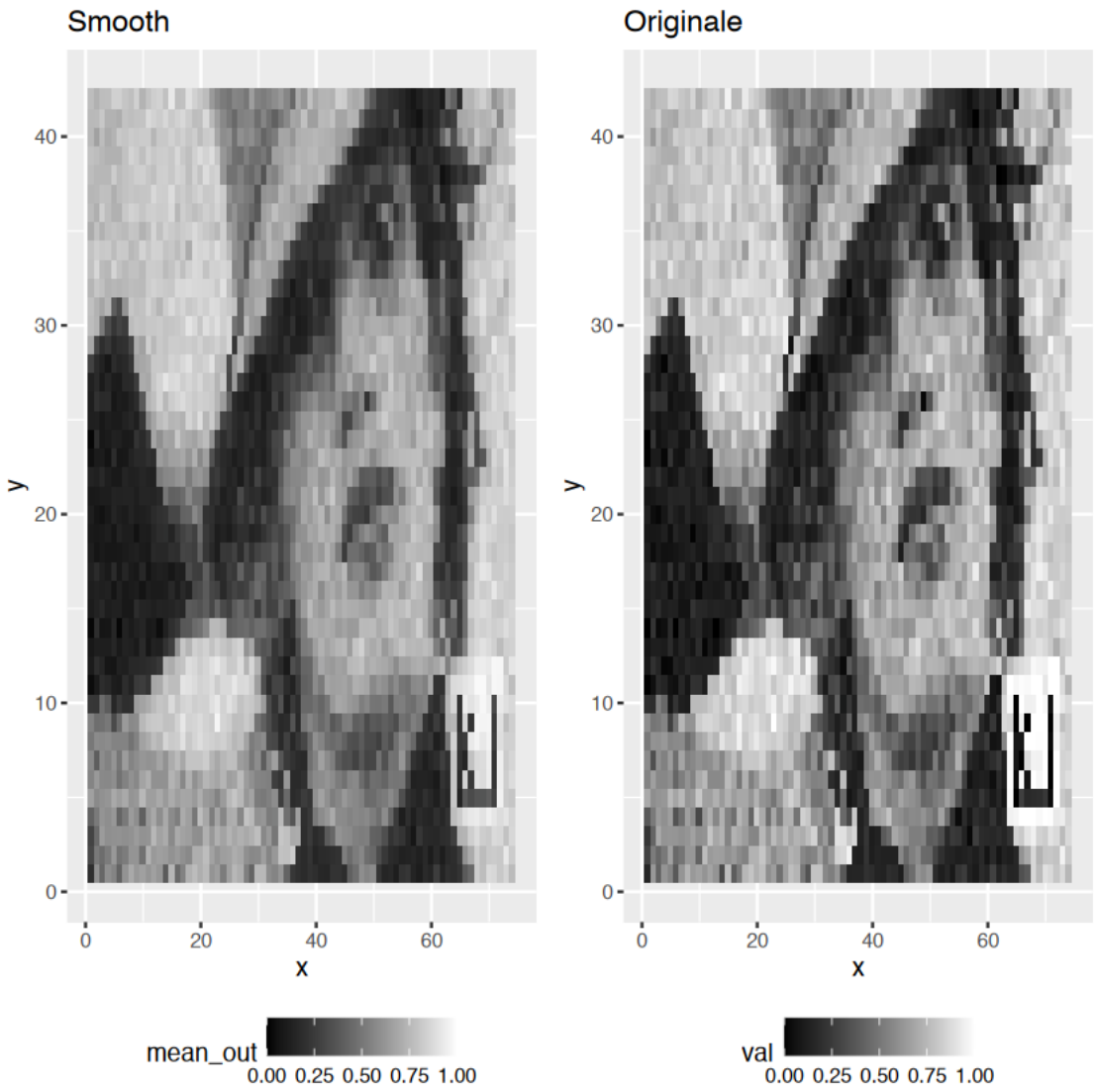


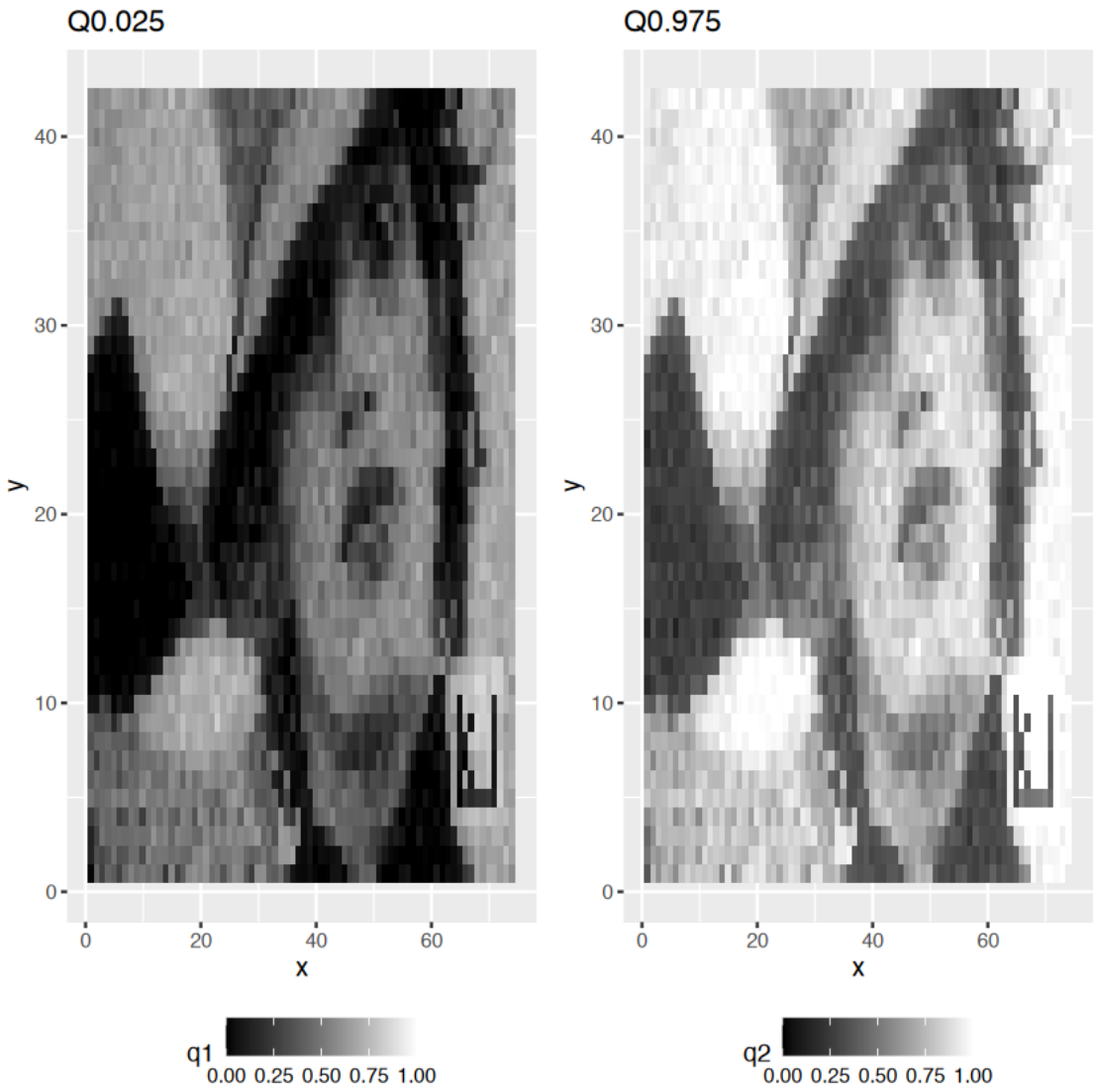
Forse servivano più iterazioni, visto che c'è molta autocorrelazione per μ, ma per motivi computazioneli (ci mette un po') accontentiamoci

Possiamo adesso mostrare la media a posteriori della figura, insieme ai limiti inferiori (pixel by pixel) e superiori della medie

[169]:
```
data_long$mean_out <- rowMeans(mean_out)
data_long$q1 <- apply(mean_out,1, function(x) quantile(x, probs = 0.025))
data_long$q2 <- apply(mean_out,1, function(x) quantile(x, probs = 0.975))


p1 <- data_long %>% ggplot(aes(x = x, y = y, fill = mean_out)) +
  geom_tile() +
```

```
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +␣
 ↪ggtitle("Smooth") + theme(legend.position = "bottom")
p2 <-data_long %>% ggplot(aes(x = x, y = y, fill = val)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +␣
 ↪ggtitle("Originale") + theme(legend.position = "bottom")

grid.arrange(p1,p2, ncol=2)

p1 <- data_long %>% ggplot(aes(x = x, y = y, fill = q1)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +
  ggtitle("Q0.025") +
  theme(legend.position = "bottom")
p2 <- data_long %>% ggplot(aes(x = x, y = y, fill = q2)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +
  ggtitle("Q0.975") +
  theme(legend.position = "bottom")

grid.arrange(p1, p2, ncol = 2)
```


Smooth
Originale
40
30
20
10
0
y
0
20
40
60
x
mean_out
0.00 0.25 0.50 0.75 1.00
val
0.00 0.25 0.50 0.75 1.00



L'immagine è sicuramente più pulita.

Il problema di questo modello, è che noi stiamo usano un modello gaussiano, che per costruzione si modifica in maniera continua e liscia. Ma la figura non lo è (c'è il volto, i capelli, e il background che cambiano "improvvisamente'' e non in maniera smooth). Questa è anche, probabilmente, la ragione per cui al catena di μ ha difficoltà e è molto autocorrelata.

Possimao fare una cosa più interessante per tener conto delle considerazioni fatte. Introduciamo una variabile

$$Z_l \equiv Z_{i,j} \in \{1, 2, \dots, K\}$$

che rappresenta a quale delle K componenti/feature (capelli, volto, background etc) appartiene il punto l-esimo. Imponiamo poi che la distribuzione delle Y sia

$$Y_{i,j}|w_{i,j}, z_{i,j} \sim N(\mu_{z_{i,j}} + \beta w_{i,j}, \sigma^2)$$

Quindi, invece di un solo valore di μ, ne abbiamo K. Questo fa in modo che in base alla feature a cui appartiene il punto (i, j) la media possa modificarsi. Potremmo fare la stessa cosa anche per β e σ^2, ma teniamo le cose semplici.

Dobbiamo adesso definire un modello per $z_{i,j}$. Anche qui, come per l'ICAR, ha senso definire le full conditional. Per esempio possiamo dire che

$$P(Z_l = k|\mathbf{Z}_{-l}) \propto \exp\left(\rho \sum_{h\sim l} I(z_h = k)\right)$$

dove

$$I(z_h = k)$$

è uguale a 1 se z_h è uguale a k, 0 altrimenti, e $\rho > 0$, che è chiamato anche inverse-temperature, è un parametro Quindi se Z_l ha tanti vicini che hanno un valore pari a k, la probabilità di k aumenta. Questo induce della dipendenza spaziale.

Potremmo usare il Brook's lemma per determinare la distribuzione congiunta, che in questo caso è

$$f(\mathbf{Z}) = \frac{\exp\left(-\rho \sum_{l=1}^{n} \sum_{h\sim l, h>l} I(z_h = z_l)\right)}{\sum_{d_1=1}^{K} \sum_{d_2=1}^{K} \cdots \sum_{d_n=1}^{K} \exp\left(-\rho \sum_{l=1}^{n} \sum_{h\sim l, h>l} I(d_h = d_l)\right)}$$

La costante di normalizzazione è impossibile da calcolare direttamente (esistono metodi approssimativi). Possiamo risolvere imponendo ρ fisso, e settarlo a un valore (potremmo provarne diversi e poi usare AIC e similare per decidere il valore migliore) **NOTA** Per il calcolo della congiunta, e per utilizzare i vari teoremi, potete fare finta che invece di

$$Z_l \in \{1, 2, \dots, K\}$$

avete

$$Z_l \in \{0, 1, \dots, K-1\}$$

che del punto di vista matematico è equivalente, ma vi permette di determinare le funzioni Q.

Nell'MCMC possiamo simulare le Z_l una alla volta. Notate che visto che le Z_l sono discrete e assumono un numero finito di valori, fintanto che ogni full conditional è finita, anche la congiunta è finita e quindi una congiunta proper.

La full conditional di Z_i, visto che è discreta è ancora una distribuzione discreta con

$$P(Z_l = k| \dots) \propto \exp\left(\rho \sum_{h\sim l} I(z_h = k)\right) \phi(Y_l|\mu_k + \beta w_l, \sigma^2)$$

Stimiamo il modello assumendo

$$K = 10 \qquad \rho = 0.3$$

Quando simuliamo dalla full conditional di Z_l, uso un trick, che spiego nella lezione sui modelli mistura. Si fa perchè altrimenti si eprde precisione nei valori $P(Z_l = k|....)$ dovuto al fatto che le probabilità non normalizzate possono essere tutte vicine a zero.

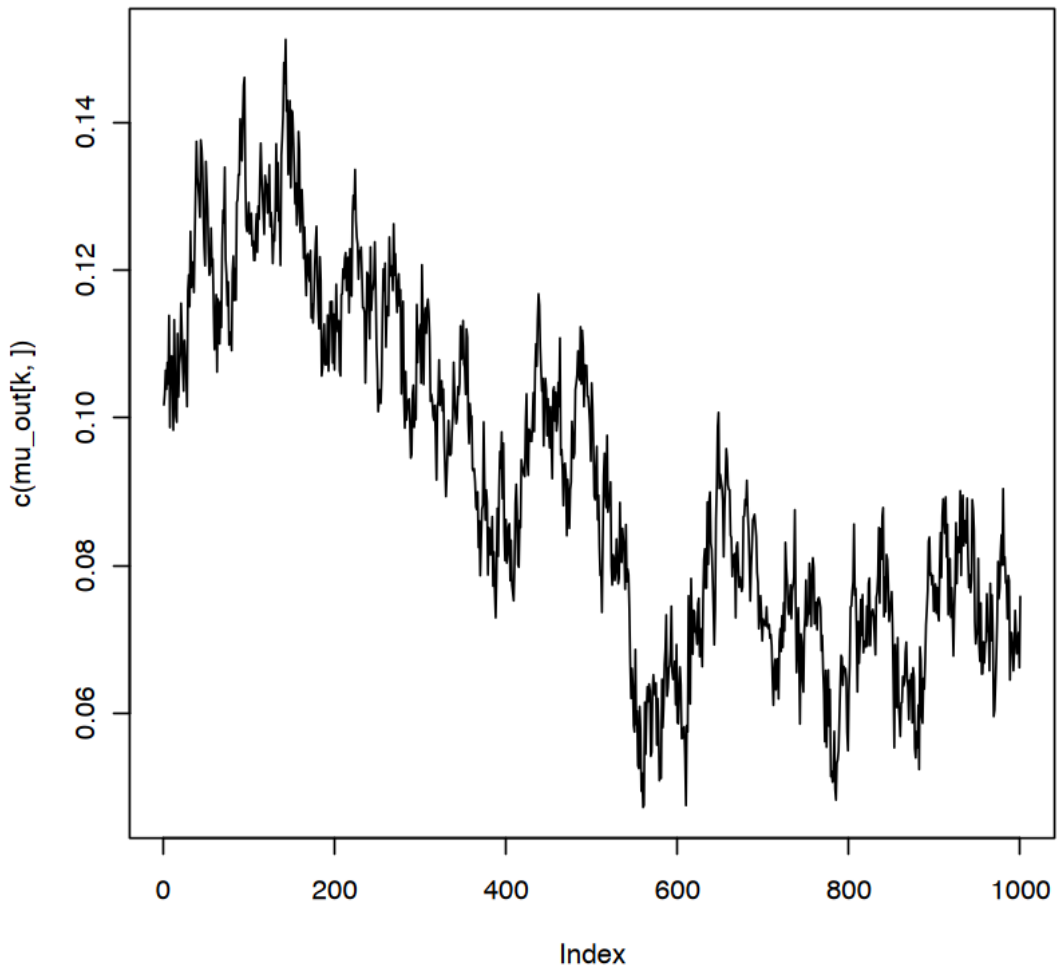
[170]:
```r
set.seed(1)
# Define MCMC parameters
niter <- 5000        # Total number of MCMC iterations
burnin <- 3000       # Number of burn-in iterations to discard
thin <- 2            # Thinning interval
n <- length(data_long$val) # Number of observations
# Compute the number of samples to save after thinning and burn-in
sample_to_save <- floor((niter - burnin) / thin)

# Hyperparameters for priors
prior_par_mean_normal <- 0       # Mean of the normal prior for beta and mu
prior_par_var_normal <- 10 # Variance of the normal prior for beta and mu
prior_par_1_sigma2 <- 1          # Shape parameter for the inverse-gamma prior␣
 ↪on sigma^2
prior_par_2_sigma2 <- 1          # Scale parameter for the inverse-gamma prior␣
 ↪on sigma^2
inv_temp_zeta <- 0.3             # Inverse temperature parameter for zeta␣
 ↪updates
K <- 10                           # Number of mixture components for zeta

# Initialize model parameters
mu_mcmc <- matrix(seq(0.1,0.99, length.out=K), nrow = K)                    #␣
 ↪Mixture means (one for each component)
beta_mcmc <- 0.2                                    # Regression coefficient
sigma2_mcmc <-0.5                                  # Variance of the errors
gp_mcmc <- matrix(0, nrow = nrow(data_long), ncol = 1)  # Latent Gaussian␣
 ↪process

# creo una variabile che accorpa in K gruppi le osservazioni, basandoci sul␣
 ↪loro valore. La uso per inizializzare zeta
order_var <- floor((rank(data_long$val, ties.method = "first") - 1) / (n - 1) *␣
 ↪(K -1)) + 1
zeta_mcmc <- matrix(order_var, nrow = nrow(data_long), ncol = 1) # Cluster␣
 ↪assignments



# Matrices to store MCMC samples
mu_out <- matrix(NA, ncol = sample_to_save, nrow = K)        # Store samples of␣
 ↪mu
beta_out <- matrix(NA, ncol = sample_to_save, nrow = 1)      # Store samples of␣
 ↪beta
sigma2_out <- matrix(NA, ncol = sample_to_save, nrow = 1)    # Store samples of␣
 ↪sigma^2
```

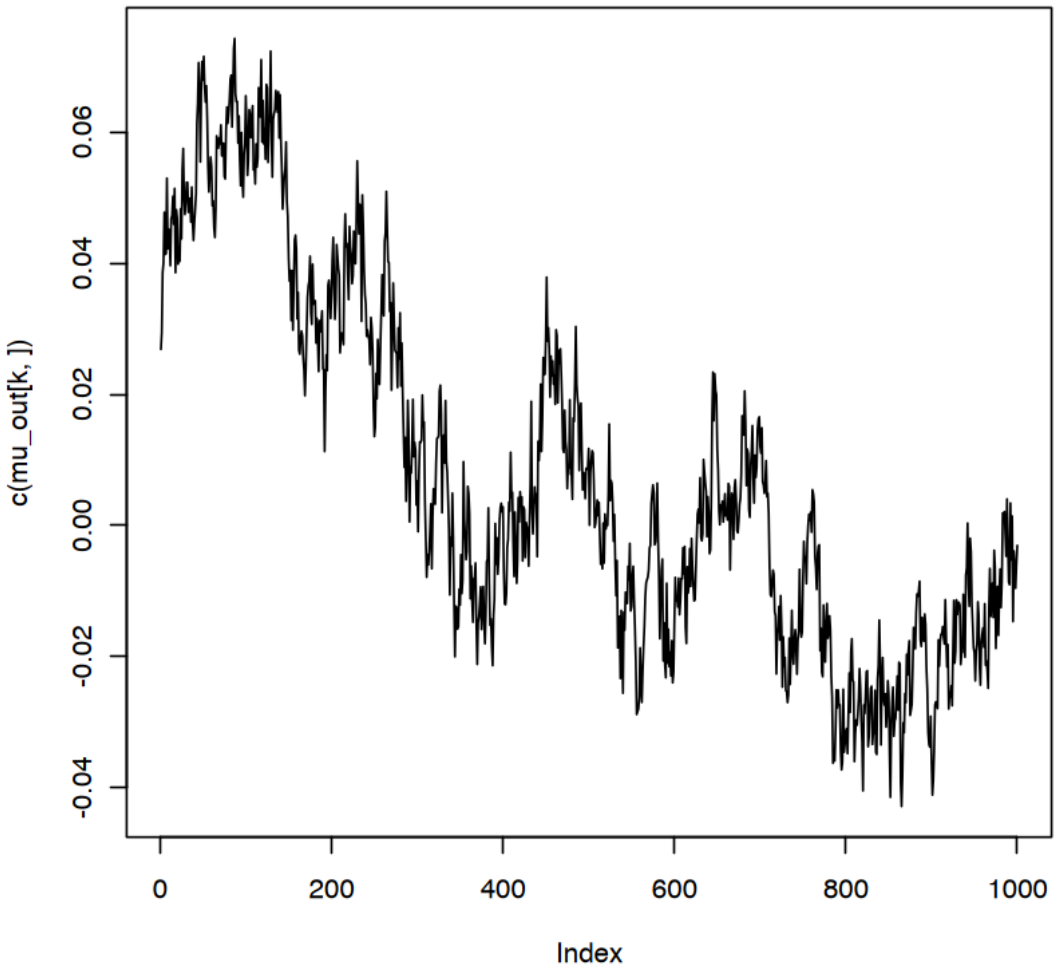
```r
gp_out <- matrix(NA, ncol = sample_to_save, nrow = nrow(data_long))  # Store␣
 ↪samples of latent Gaussian process
mean_out <- matrix(NA, ncol = sample_to_save, nrow = nrow(data_long)) # Store␣
 ↪predicted means
zeta_out <- matrix(NA, ncol = sample_to_save, nrow = nrow(data_long)) # Store␣
 ↪cluster assignments

# Initialize latent variable
y_latent <- data_long$val
if (length(w_one) > 0) {
  # Update y_latent for observations with value 1
  y_latent[w_one] <- rtruncnorm(length(w_one), a = 1, b = Inf, mean =␣
 ↪mu_mcmc[zeta_mcmc[w_one]] + beta_mcmc * gp_mcmc[w_one], sd =␣
 ↪sqrt(sigma2_mcmc))
}
if (length(w_zero) > 0) {
  # Update y_latent for observations with value 0
  y_latent[w_zero] <- rtruncnorm(length(w_zero), a = -Inf, b = 0, mean =␣
 ↪mu_mcmc[zeta_mcmc[w_zero]] + beta_mcmc * gp_mcmc[w_zero], sd =␣
 ↪sqrt(sigma2_mcmc))
}


app <- burnin              # Number of iterations before saving samples

# Main MCMC loop
for (save_iter in 1:sample_to_save) {
  for (imcmc in 1:app) {
    # Update sigma^2
    sigma2_mcmc <- 1 / rgamma(1, shape = n / 2 + prior_par_1_sigma2, rate =␣
 ↪sum((y_latent - mu_mcmc[zeta_mcmc] - beta_mcmc * gp_mcmc)^2) / 2 +␣
 ↪prior_par_2_sigma2)

    # Update beta
    post_var <- 1 / (sum(gp_mcmc^2) / sigma2_mcmc + 1 / prior_par_var_normal)
    post_mean <- post_var * (sum(gp_mcmc * (y_latent - mu_mcmc[zeta_mcmc])) /␣
 ↪sigma2_mcmc + prior_par_mean_normal / prior_par_var_normal)
    beta_mcmc <- rnorm(1, mean = post_mean, sd = sqrt(post_var))

    # Update mu for each mixture component
    for (k in 1:K) {
      w <- which(zeta_mcmc == k)  # Indices of observations in cluster k
      post_var <- 1 / (length(w) / sigma2_mcmc + 1 / prior_par_var_normal)
      post_mean <- post_var * (sum(y_latent[w] - beta_mcmc * gp_mcmc[w]) /␣
 ↪sigma2_mcmc + prior_par_mean_normal / prior_par_var_normal)
      mu_mcmc[k] <- rnorm(1, mean = post_mean, sd = sqrt(post_var))
```

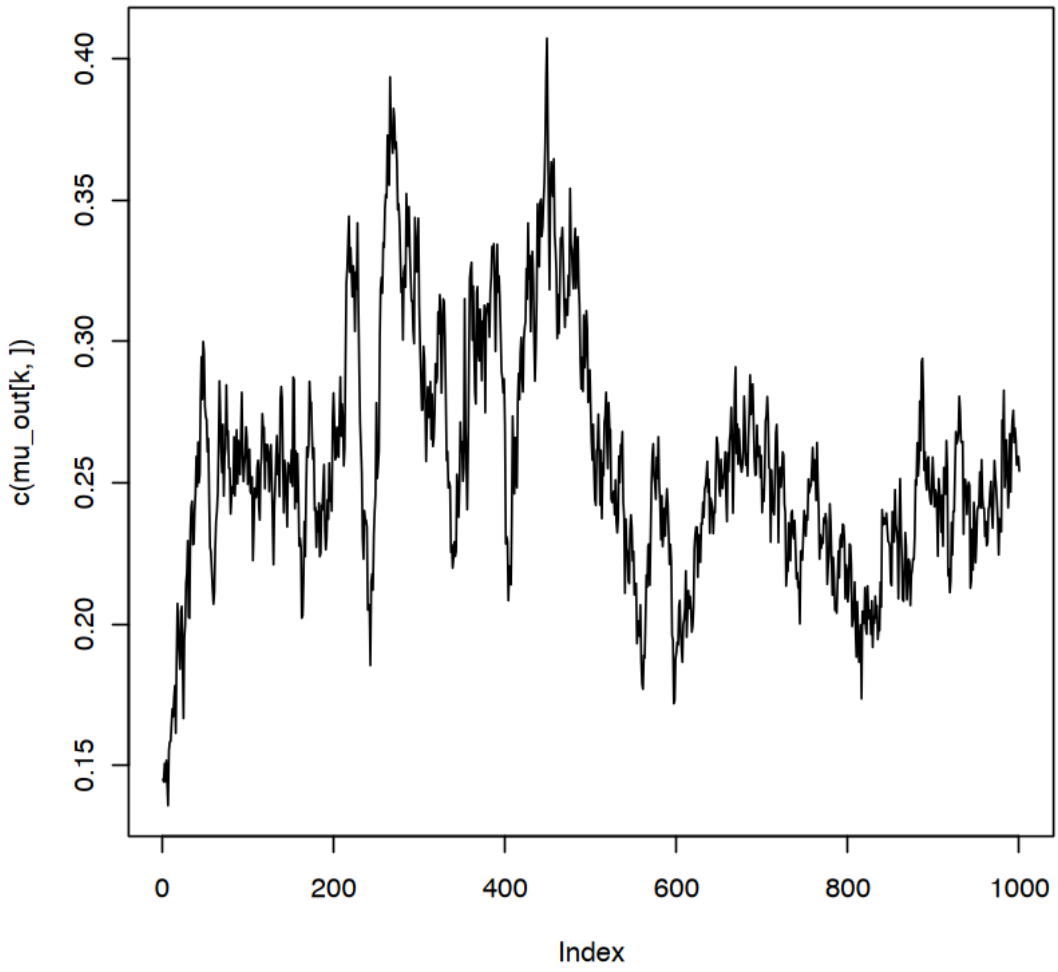
```
  }

  # Update y_latent
  if (length(w_one) > 0) {
    y_latent[w_one] <- rtruncnorm(length(w_one), a = 1, b = Inf, mean =␣
↪mu_mcmc[zeta_mcmc[w_one]] + beta_mcmc * gp_mcmc[w_one], sd =␣
↪sqrt(sigma2_mcmc))
  }
  if (length(w_zero) > 0) {
    y_latent[w_zero] <- rtruncnorm(length(w_zero), a = -Inf, b = 0, mean =␣
↪mu_mcmc[zeta_mcmc[w_zero]] + beta_mcmc * gp_mcmc[w_zero], sd =␣
↪sqrt(sigma2_mcmc))
  }

  # Update gp_mcmc (latent Gaussian process)
  for (i in 1:n) {
    neigh <- data_neigh[[i]]  # Neighbor indices for observation i
    cond_var <- 1 / Sigma_inv[i, i]
    cond_mean <- (-(1 / Sigma_inv[i, i]) * Sigma_inv[i, neigh]) %*%␣
↪gp_mcmc[neigh]

    post_var <- 1 / (beta_mcmc^2 / sigma2_mcmc + 1 / cond_var)
    post_mean <- post_var * (beta_mcmc * (y_latent[i] -␣
↪mu_mcmc[zeta_mcmc[i]]) / sigma2_mcmc + cond_mean / cond_var)
    gp_mcmc[i] <- rnorm(1, mean = post_mean, sd = sqrt(post_var))
  }

  # Update zeta (cluster assignments)
  for (i in 1:n) {
    neigh <- data_neigh[[i]]
    log_prob <- rep(NA, K)
    for (k in 1:K) {
      log_prob[k] <- inv_temp_zeta * sum(zeta_mcmc[neigh] == k) +
                     dnorm(y_latent[i], mean = mu_mcmc[k] + beta_mcmc *␣
↪gp_mcmc[i], sd = sqrt(sigma2_mcmc), log = TRUE)
    }
    zeta_mcmc[i] <- sample(1:K, size = 1, prob = exp(log_prob -␣
↪max(log_prob)))
  }
}

# Save samples after thinning
app <- thin
mu_out[, save_iter] <- mu_mcmc
beta_out[save_iter] <- beta_mcmc
sigma2_out[save_iter] <- sigma2_mcmc
```

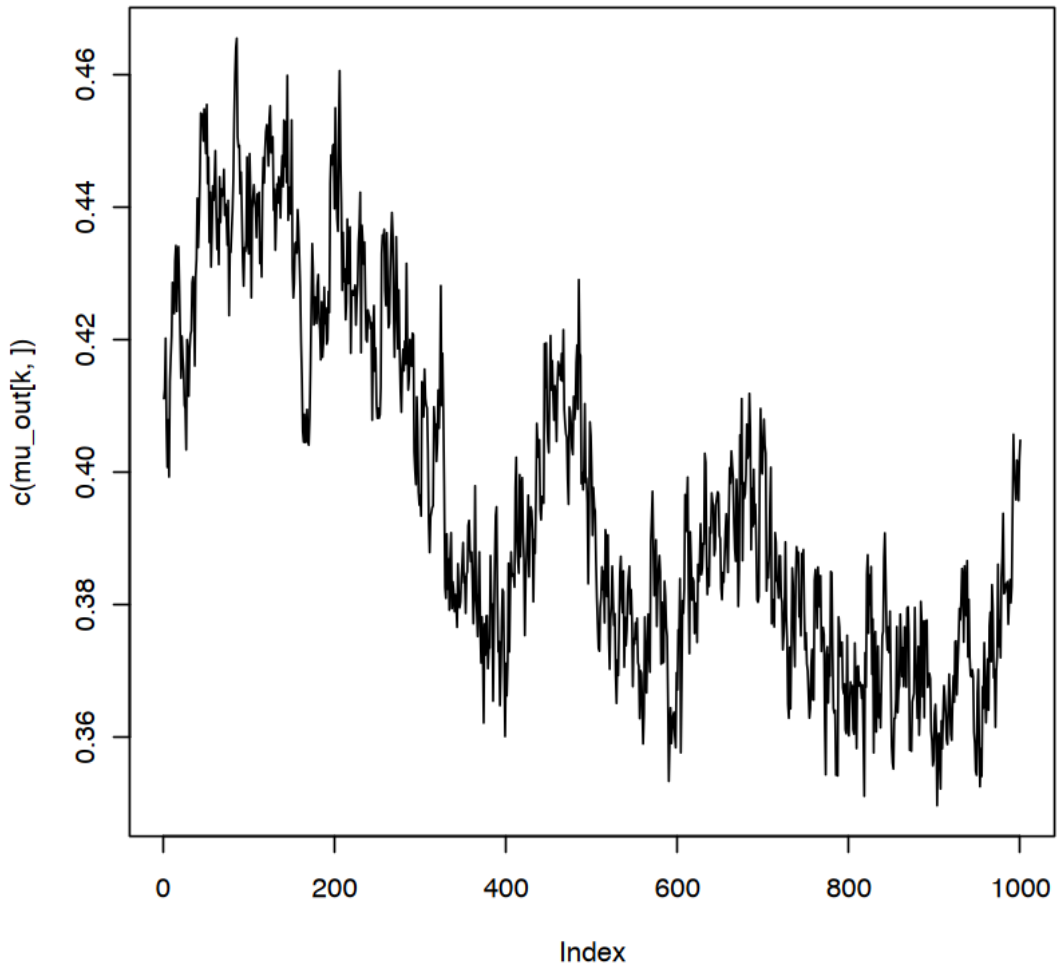
```
  gp_out[, save_iter] <- gp_mcmc
  mean_out[, save_iter] <- mu_mcmc[zeta_mcmc] + beta_mcmc * gp_mcmc
  zeta_out[, save_iter] <- zeta_mcmc
  mean_out[which(mean_out[, save_iter] < 0), save_iter] <- 0
  mean_out[which(mean_out[, save_iter] > 1), save_iter] <- 1
}
```

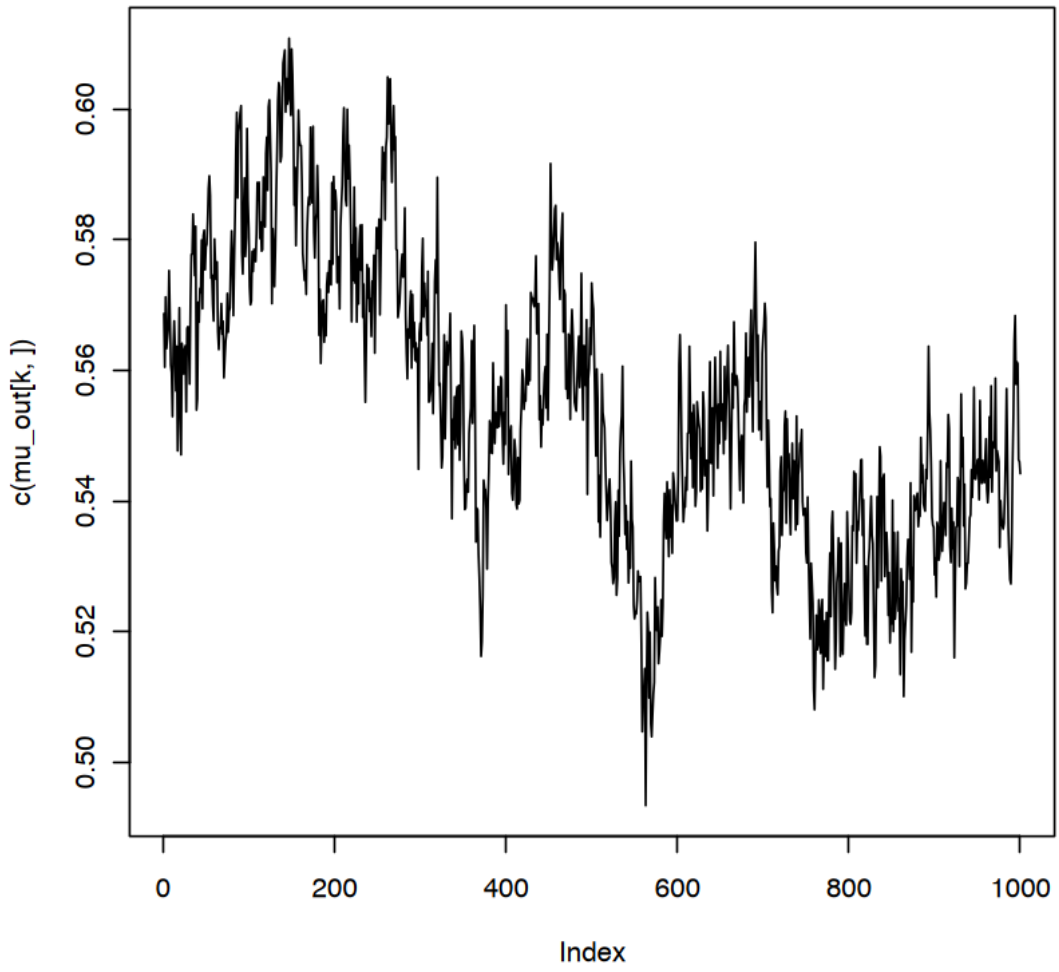
[171]:
```
for(k in 1:K)
{
  plot(c(mu_out[k,]), type="l")
}

plot(c(beta_out), type = "l")
plot(c(sigma2_out), type = "l")
```

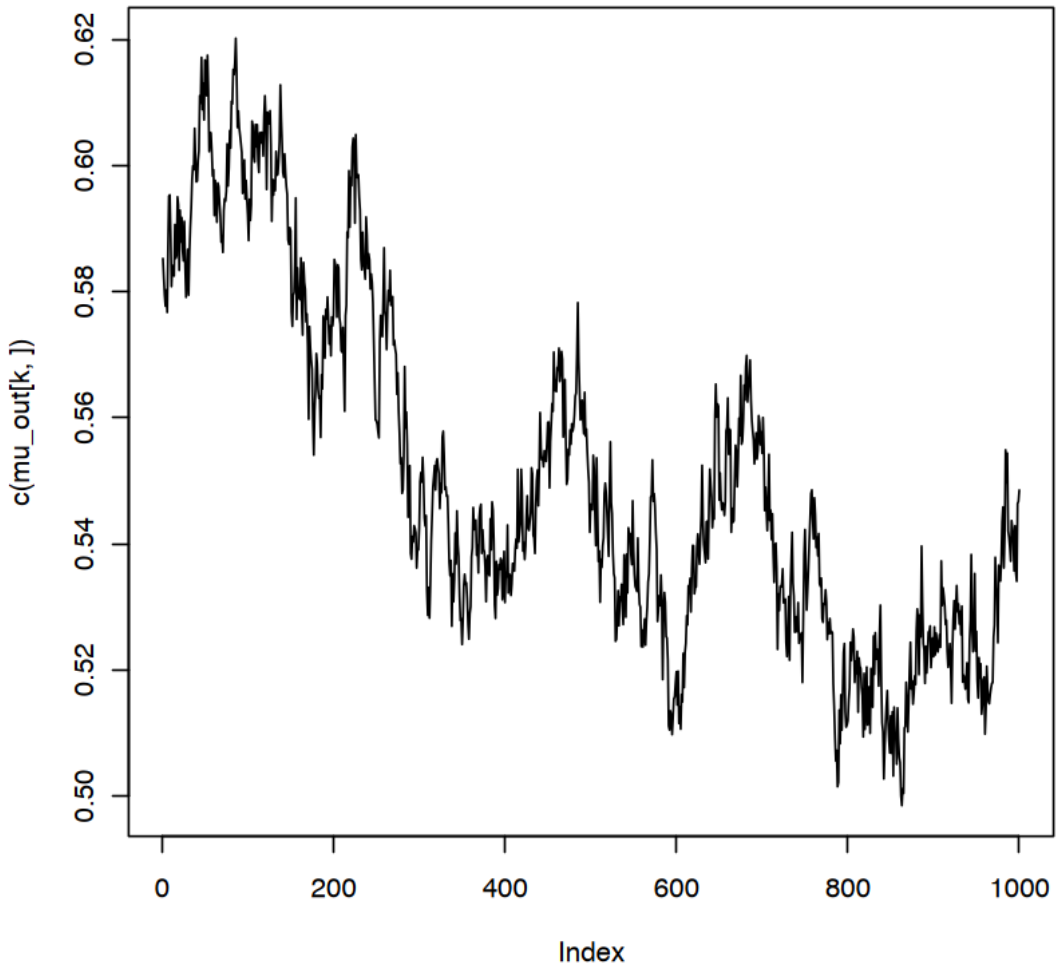


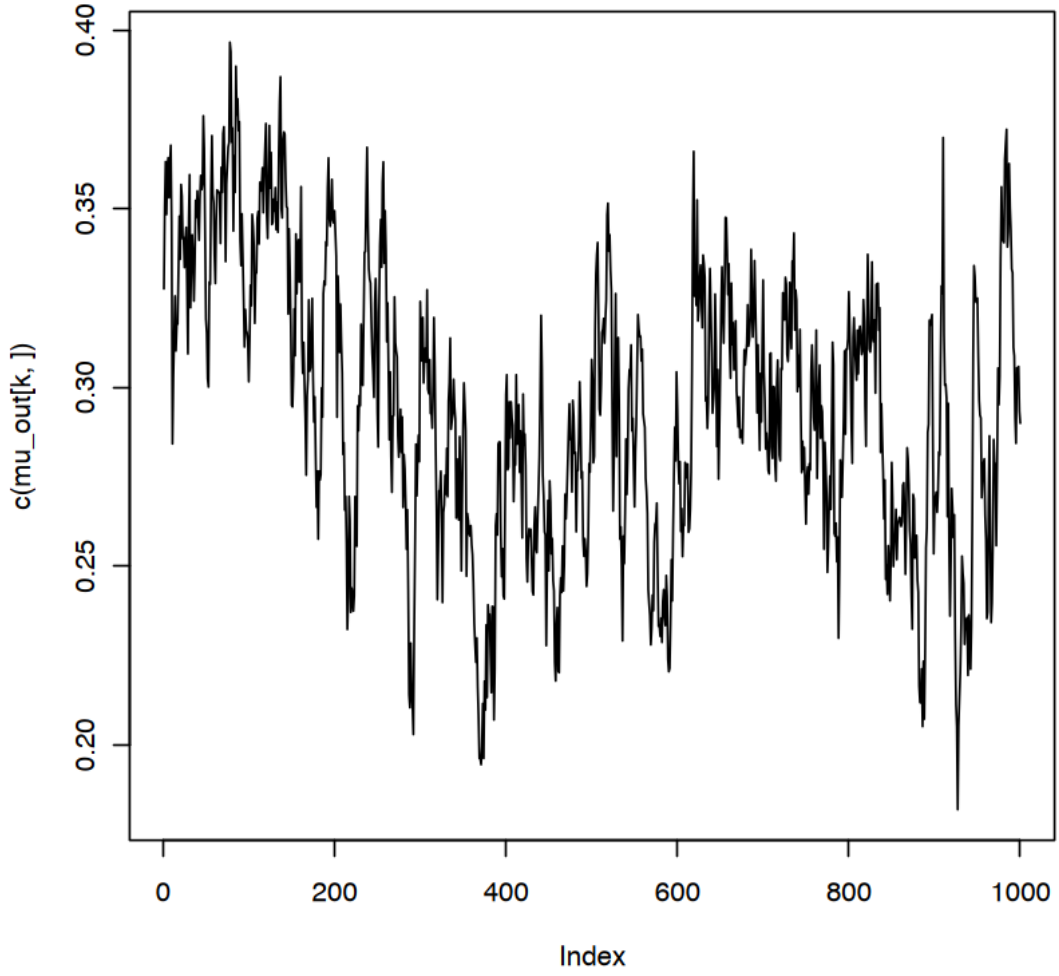

c(mu_out[k,])
0.06
0.04
0.02
0.00
-0.02
-0.04
0
200
400
600
800
1000
Index

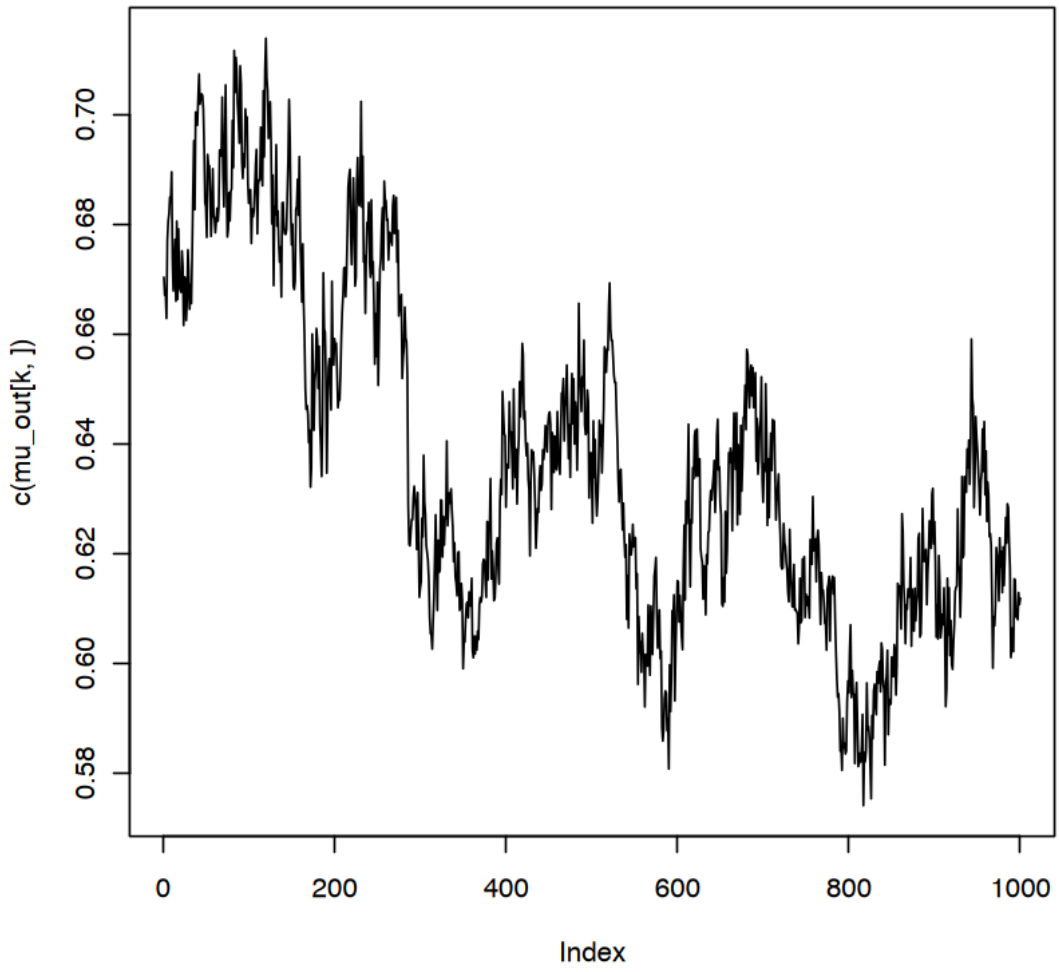

c(mu_out[k,])
0.15
0.20
0.25
0.30
0.35
0.40
0
200
400
600
800
1000
Index

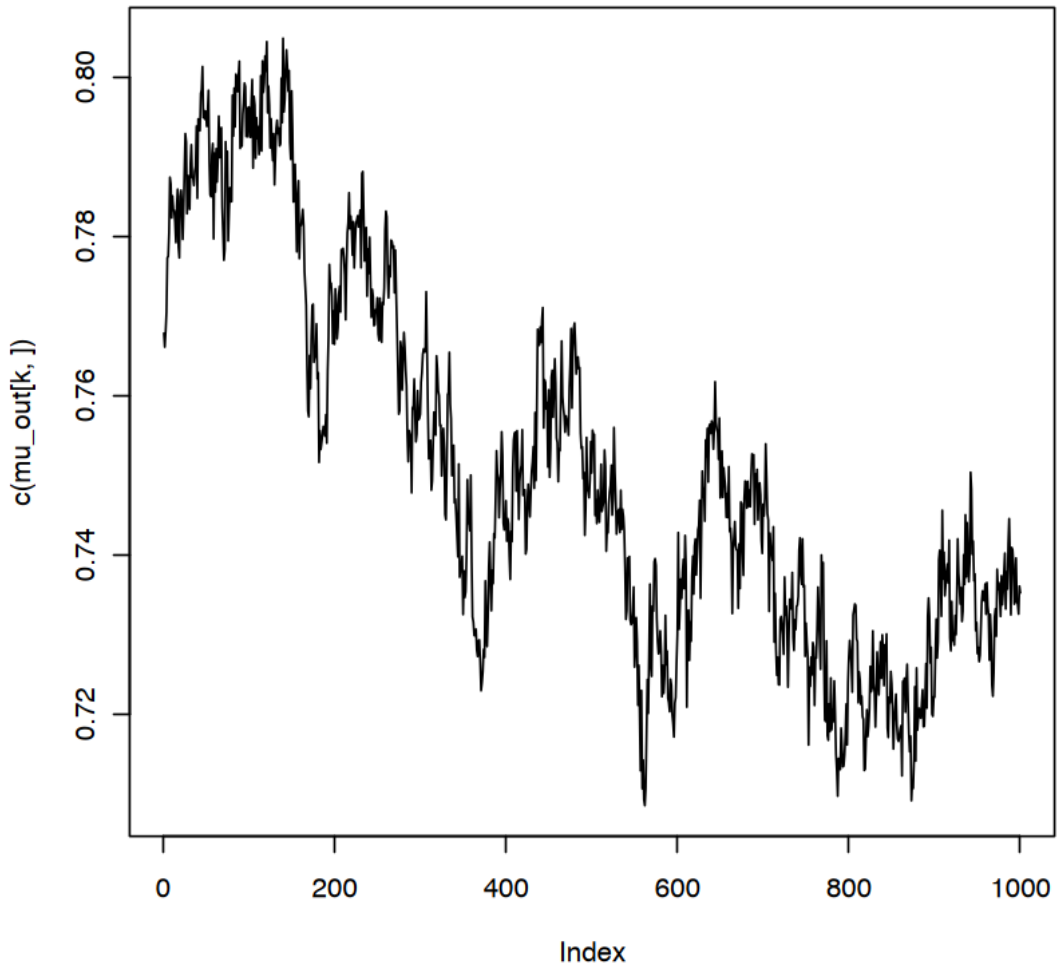

c(mu_out[k,])
0.36
0.38
0.40
0.42
0.44
0.46
0
200
400
600
800
1000
Index

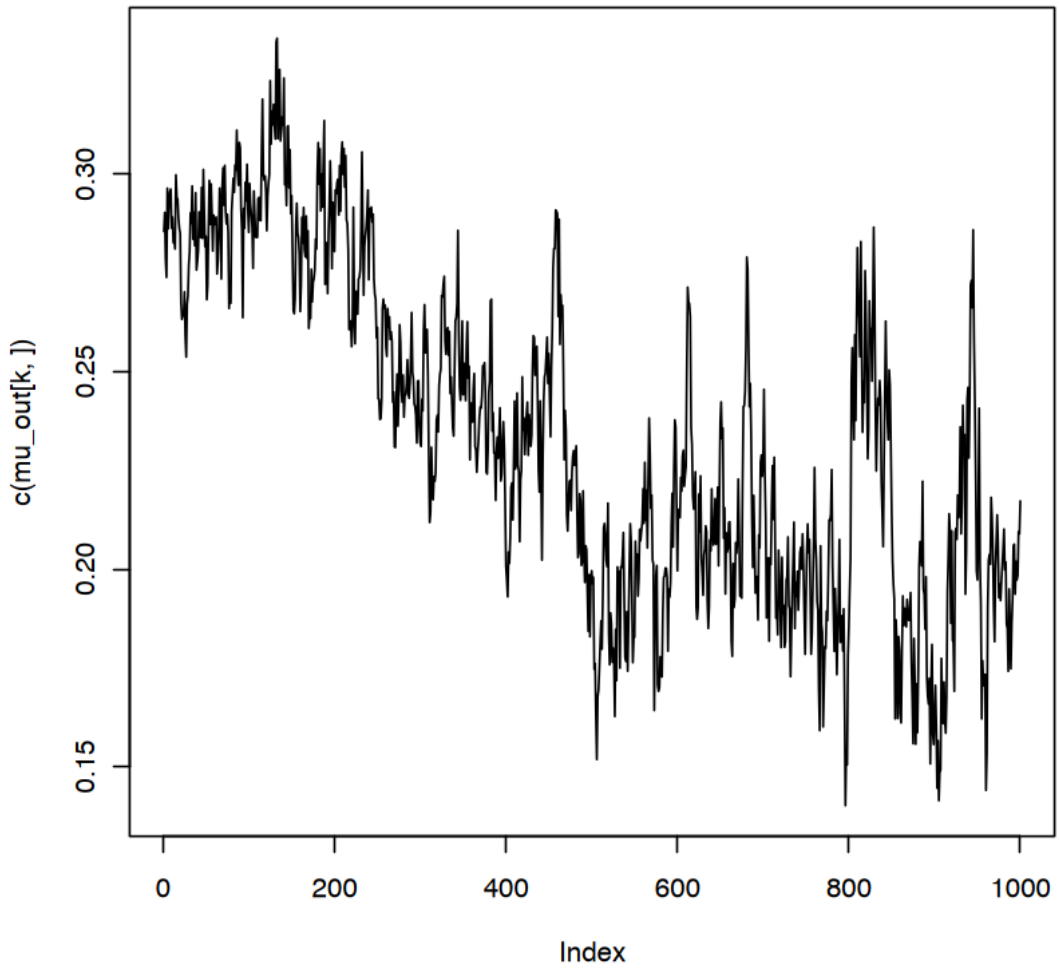

c(mu_out[k,])
0.50
0.52
0.54
0.56
0.58
0.60
0
200
400
600
800
1000
Index

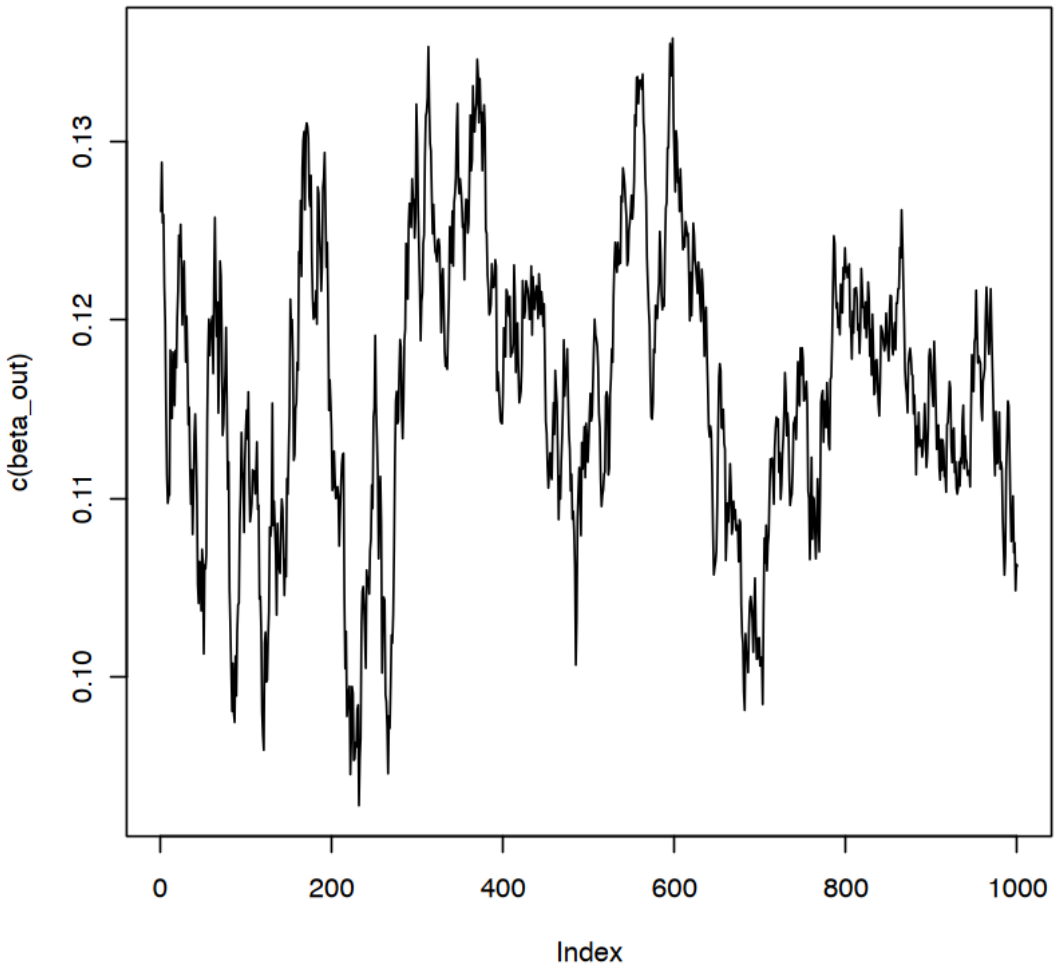

c(mu_out[k,])
0.50
0.52
0.54
0.56
0.58
0.60
0.62
0
200
400
600
800
1000
Index

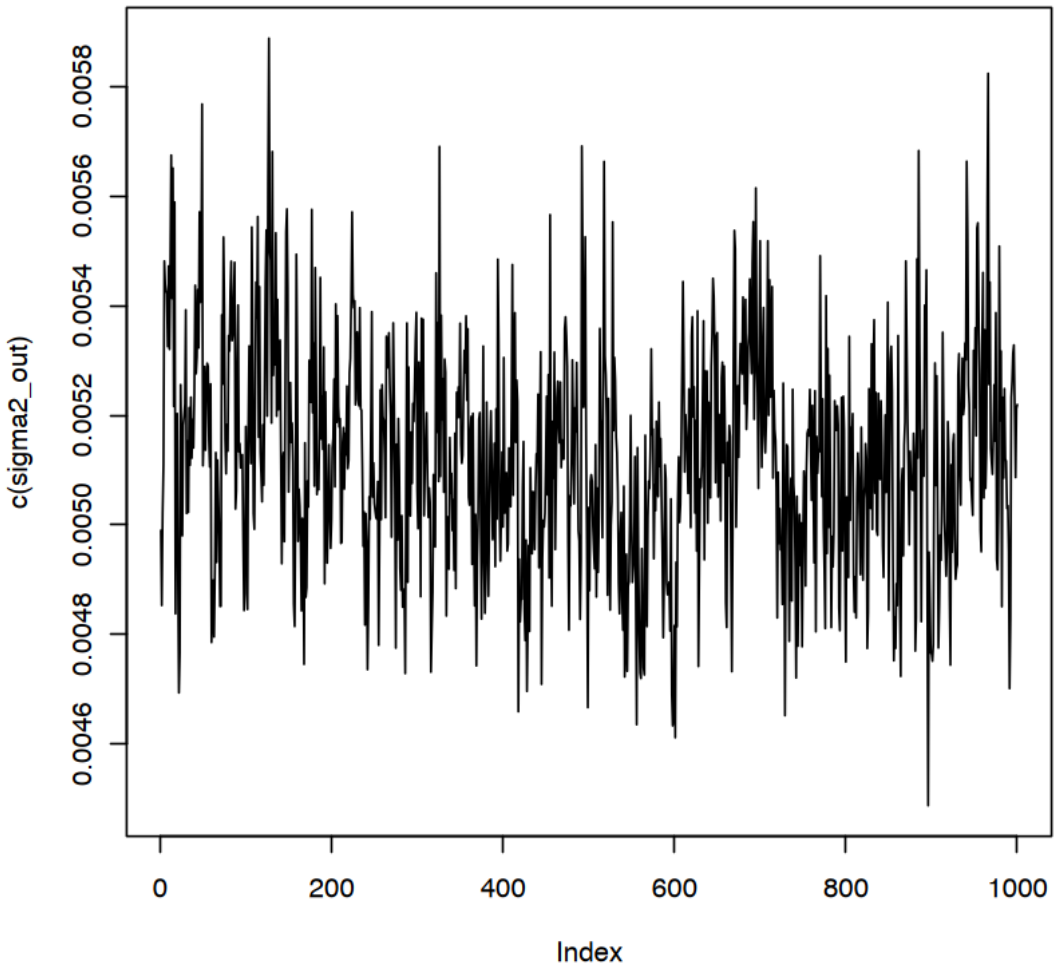

c(mu_out[k,])
0.20
0.25
0.30
0.35
0.40
0
200
400
600
800
1000
Index


c(mu_out[k,])
0.58
0.60
0.62
0.64
0.66
0.68
0.70
0
200
400
600
800
1000
Index


c(mu_out[k,])
0.72
0.74
0.76
0.78
0.80
0
200
400
600
800
1000
Index


c(mu_out[k,])
0.15
0.20
0.25
0.30
0
200
400
600
800
1000
Index


c(beta_out)
0.10
0.11
0.12
0.13
0
200
400
600
800
1000
Index



Anche in questo caso servirebbero più iterazioni per risolver ei problemi di μ. Ma probabilmente ne servono troppe, a meno che non si implementi un sampler un po' più "furbo' ', tipo birth/death, e merge/split

facciamo grafici simili a prima, ma in questo caso vogliamo anche mostrare anche una stima a posteriori della classificazione (Z). Per trovare una stima, invece della media a posteriori (che non ha senso), usiamo la moda. Indichiamo la moda come $\hat{Z}_l$

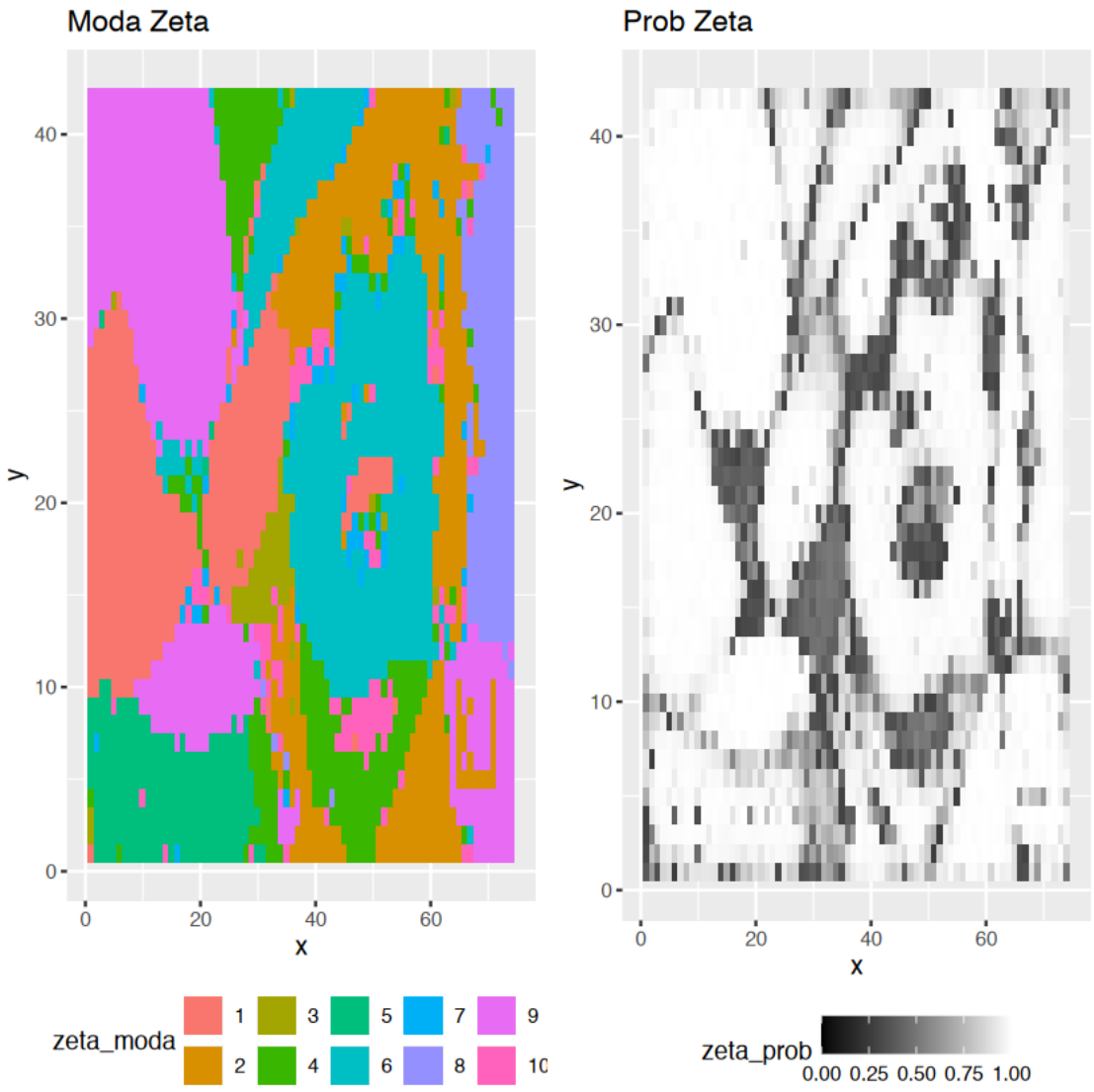
Insieme alla moda, per avere una stima della variabilità, mostriamo anche

$$P(Z_l = \hat{Z}_l|\mathbf{x})$$

Se questo valore è vicino a 1, allora la a posteriori è concentrata sulla moda, altrimento se è piccolo, la moda è poco rappresentativa

[172]:
```
findmode <- function(x) {
  TT <- table(as.vector(x))
  return(as.numeric(names(TT)[TT == max(TT)][1]))
}
```

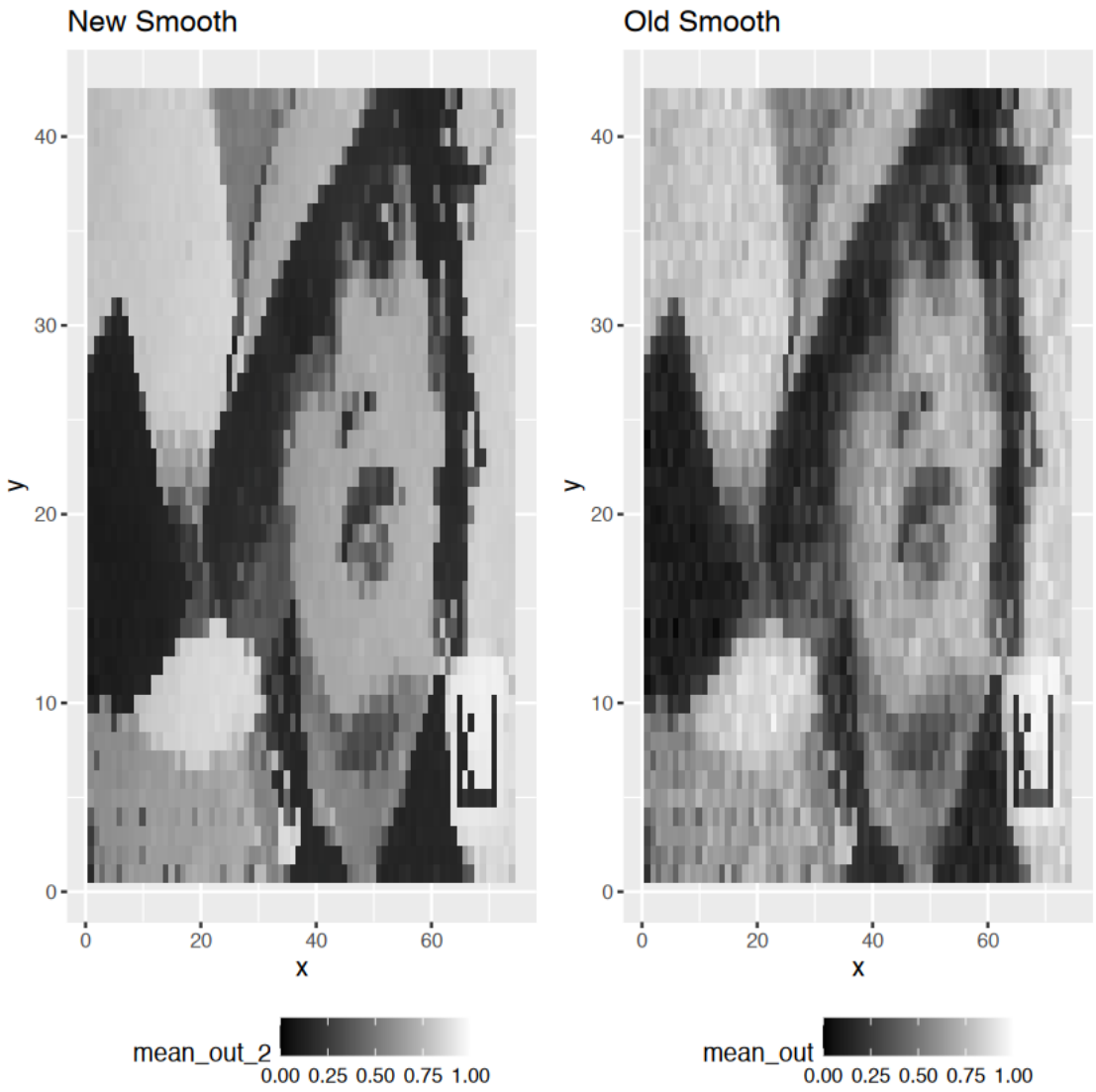
[173]:
```
data_long$mean_out_2 <- rowMeans(mean_out)
data_long$q1_2 <- apply(mean_out, 1, function(x) quantile(x, probs = 0.025))
data_long$q2_2 <- apply(mean_out, 1, function(x) quantile(x, probs = 0.975))

# moda
data_long$zeta_moda <- factor(apply(zeta_out, 1, findmode))
# calcolo P(Z_l = Z_l|x)
zeta_out_diff <- matrix(NA, nrow=nrow(zeta_out), ncol=ncol(zeta_out))
for(i in 1:ncol(zeta_out))
{
  zeta_out_diff[, i] <- zeta_out[, i] == data_long$zeta_moda
}

data_long$zeta_prob <- rowMeans(zeta_out_diff)


p1 <- data_long %>% ggplot(aes(x = x, y = y, fill = zeta_moda)) +
  geom_tile() + ggtitle("Moda Zeta") +
    theme(legend.position = "bottom")
p2 <- data_long %>% ggplot(aes(x = x, y = y, fill = zeta_prob)) +
  geom_tile() +
  ggtitle("Prob Zeta") + scale_fill_gradient(low = "black", high = "white",␣
 ↪limits = c(0, 1)) +
  theme(legend.position = "bottom")
grid.arrange(p1, p2, ncol = 2)
```

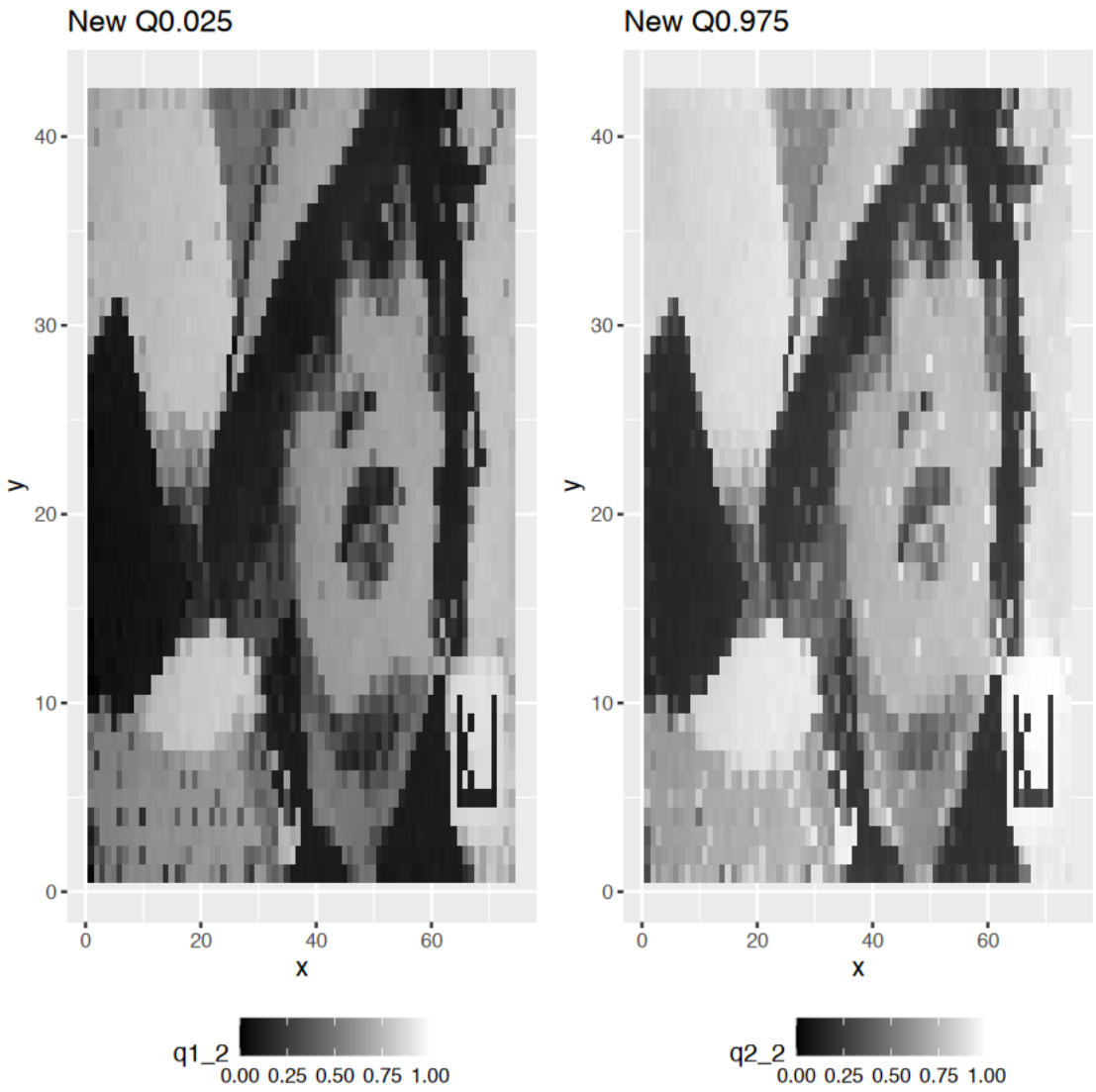


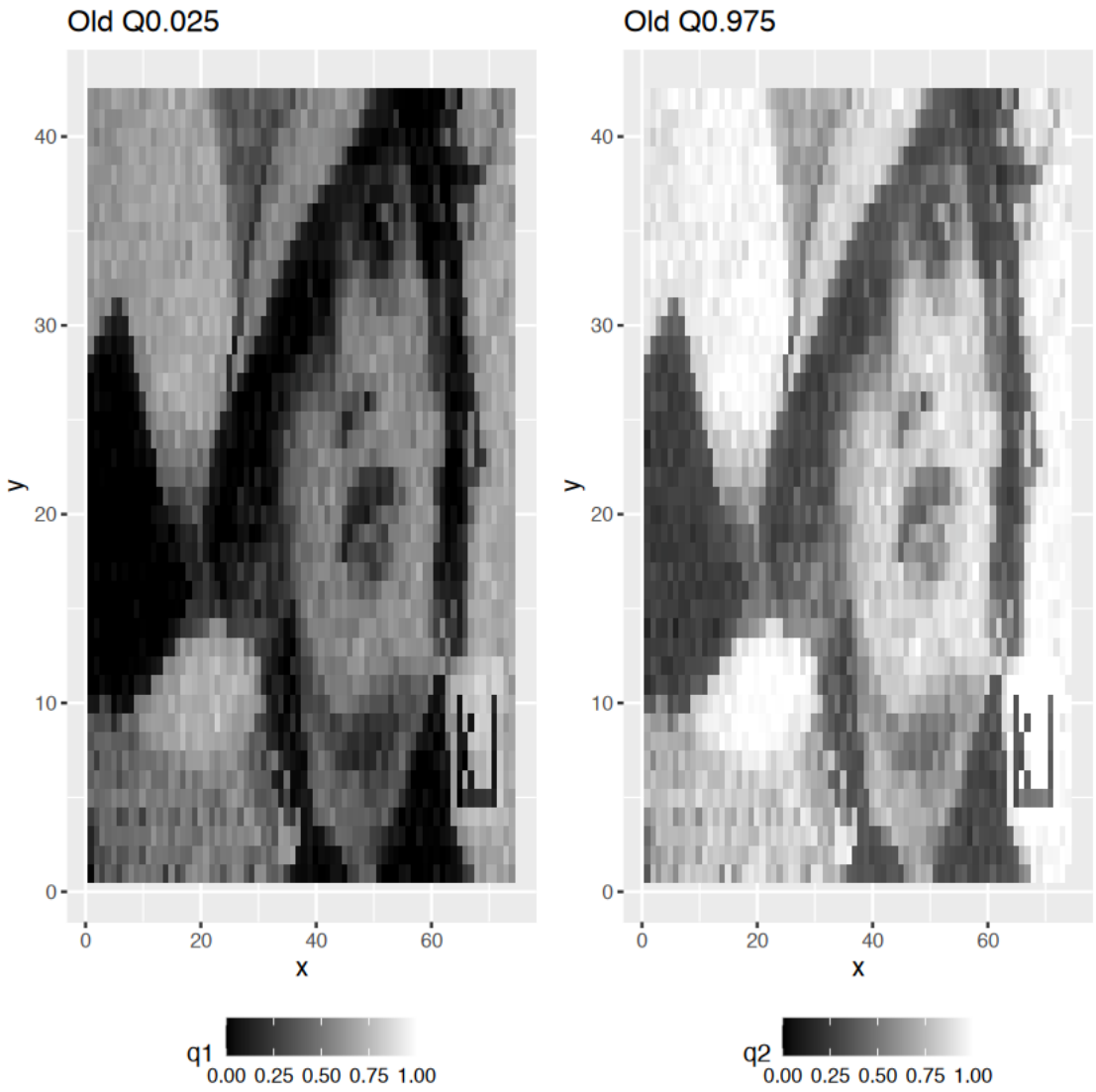
I punti in cui il modello è meno sicuro del valore di Z sono le zone in cui differenti feature della figura si incontrano. Volendo potremmo usare anche questo indice per trovare i contorni degli oggetti.

[174]:
```
p1 <- data_long %>% ggplot(aes(x = x, y = y, fill = mean_out_2)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +
  ggtitle("New Smooth") +
  theme(legend.position = "bottom")
p2 <- data_long %>% ggplot(aes(x = x, y = y, fill = mean_out)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +
  ggtitle("Old Smooth") +
```

```
  theme(legend.position = "bottom")

grid.arrange(p1, p2, ncol = 2)


p1 <- data_long %>% ggplot(aes(x = x, y = y, fill = q1_2)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +
  ggtitle("New Q0.025") +
  theme(legend.position = "bottom")
p2 <- data_long %>% ggplot(aes(x = x, y = y, fill = q2_2)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +
  ggtitle("New Q0.975") +
  theme(legend.position = "bottom")

grid.arrange(p1, p2, ncol = 2)

p1 <- data_long %>% ggplot(aes(x = x, y = y, fill = q1)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +
  ggtitle("Old Q0.025") +
  theme(legend.position = "bottom")
p2 <- data_long %>% ggplot(aes(x = x, y = y, fill = q2)) +
  geom_tile() +
  scale_fill_gradient(low = "black", high = "white", limits = c(0, 1)) +
  ggtitle("Old Q0.975") +
  theme(legend.position = "bottom")

grid.arrange(p1, p2, ncol = 2)
```


New Smooth
Old Smooth
y
x
0
10
20
30
40
0
20
40
60
mean_out_2
mean_out
0.00 0.25 0.50 0.75 1.00


New Q0.025
New Q0.975
y
x
0
10
20
30
40
0
20
40
60
q1_2
q2_2
0.00 0.25 0.50 0.75 1.00


Old Q0.025
Old Q0.975
y
x
0
10
20
30
40
0
20
40
60
q1
q2
0.00 0.25 0.50 0.75 1.00