Inferencia bayesiana en la práctica: MCMC

Table of contents

Si tenemos datos y un modelo con previas, hay una distribución posterior.

¿Pero cómo accedemos a ella?

El numerador es fácil de evaluar, pero no la integral del denominador.

$$p(\alpha, \beta \mid y) = \frac{L(y \mid \alpha, \beta)\ p(\alpha)\ p(\beta)}{\int\int L(y \mid \alpha, \beta)\ p(\alpha)\ p(\beta)\ \mathrm{d}\alpha\ \mathrm{d}\beta}.$$

Probablemente nos interese conocer la distribución marginal de los parámetros. Esto requiere integrar:

$$p(\alpha \mid y) = \int p(\alpha, \beta \mid y)\ \mathrm{d}\beta$$
$$p(\beta \mid y) = \int p(\alpha, \beta \mid y)\ \mathrm{d}\alpha.$$

Y seguramente nos interesará conocer las esperanzas (promedios),

$$\mathbb{E}[\alpha \mid y] = \int \alpha \; p(\alpha \mid y) \; \mathrm{d}\alpha.$$

Y la función de distribución acumulada

$$F(\alpha \mid y) = \Pr(\alpha \leq x) = \int_{-\infty}^{x} \; p(\alpha \mid y) \; \mathrm{d}\alpha,$$

que invertida nos permite calcular los cuantiles (función cuantil):

$$Q(\mathrm{p}) = F^{-1}(\mathrm{p}).$$

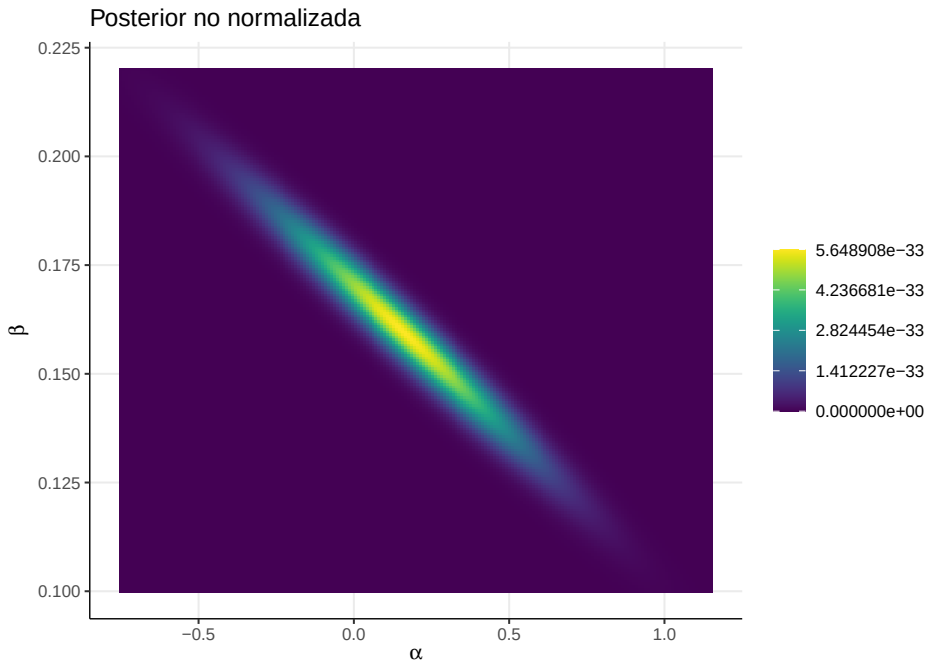
Todo esto requiere integrar.

Para generar una intuición de estos conceptos, tomemos la posterior del modelo de número de incendios por verano. Aproximaremos las integrales discretizando los parámetros, una estrategia llamada *integración numérica*.

0.0.1 Integración numérica

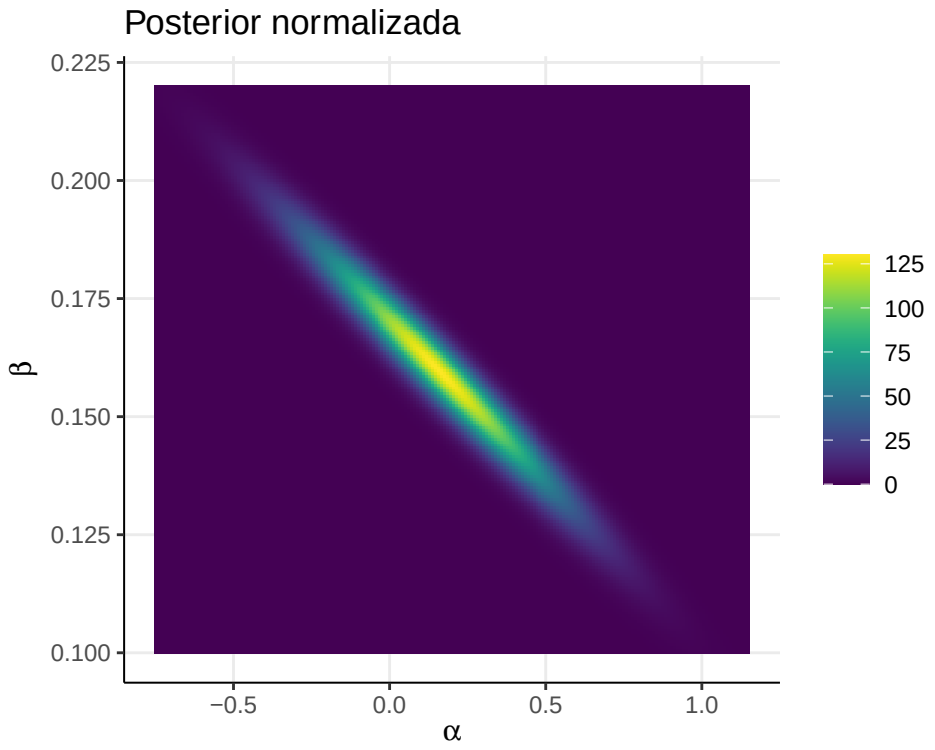
```
datos <- read.csv(here::here("datos", "barbera_data_fire_total_climate.csv"))

# definimos parámetros de la previa
mu_a <- 0; sigma_a <- 1
mu_b <- 0; sigma_b <- 0.1

# Función que evalúa la densidad posterior no normalizada
# ("un" de "unnormalized", porque siempre colonializado, nunca incolonializado)
post_fire_un <- function(alpha, beta, log = T) {

  ## Log Verosimilitud (mismo código que en like_fire)

  # Calculamos la media, lambda:
  lambda <- exp(alpha + beta * datos$fwi)
  # usando lambda evaluamos la verosimilitud de cada observación,
  # en escala log:
  like_pointwise <- dpois(datos$fires, lambda, log = T)
  # la log-verosimilitud conjunta es la suma de las log-verosimilitudes por
  # observación:
```

```r
  like <- sum(like_pointwise)

  ## Log Previa
  lprior_a <- dnorm(alpha, mean = mu_a, sd = sigma_a, log = T)
  lprior_b <- dnorm(beta, mean = mu_b, sd = sigma_b, log = T)
  lprior <- lprior_a + lprior_b

  ## Log Posterior
  lpost <- like + lprior

  if (log) return(lpost)
  else return(exp(lpost))
}

# Creamos grilla de valores de alpha y beta.
side <- 200 # cantidad de valores por lado

alpha_seq <- seq(-0.75, 1.15, length.out = side)
beta_seq <- seq(0.1, 0.22, length.out = side)

grilla <- expand.grid(
  alpha = alpha_seq,
  beta = beta_seq
)

# Evaluamos la densidad posterior para cada valor
size <- side ^ 2 # cantidad de valores en la grilla
for (i in 1:size) {
  grilla$post[i] <- post_fire_un(grilla$alpha[i], grilla$beta[i], log = F)
}

# Graficamos posterior no normalizada
ggplot(grilla, aes(x = alpha, y = beta, fill = post)) +
  geom_tile() +
  scale_fill_viridis() +
  theme(legend.title = element_blank()) +
  xlab(expression(alpha)) +
  ylab(expression(beta)) +
  ggtitle("Posterior no normalizada") +
  nice_theme()
```



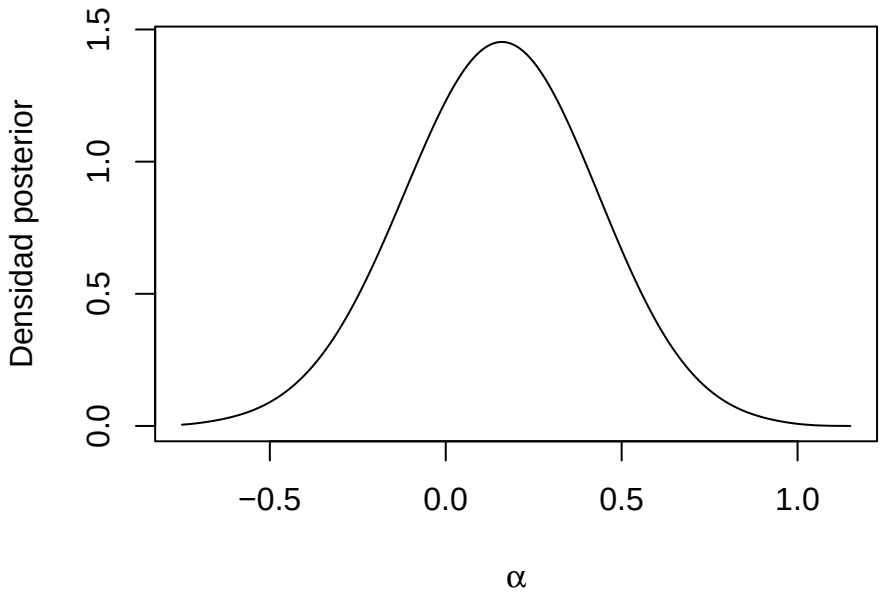
Calculamos la integral del denominador numéricamente, y obtenemos la posterior normalizada.

```
size_a <- diff(alpha_seq)[1]
size_b <- diff(beta_seq)[1]
area <- size_a * size_b
# área de cada celda en la grilla.

grilla$post <- normalize_dens(grilla$post, area)

# Graficamos posterior normalizada
ggplot(grilla, aes(x = alpha, y = beta, fill = post)) +
  geom_tile() +
  scale_fill_viridis() +
  theme(legend.title = element_blank()) +
  xlab(expression(alpha)) +
  ylab(expression(beta)) +
  ggtitle("Posterior normalizada") +
  nice_theme()
```



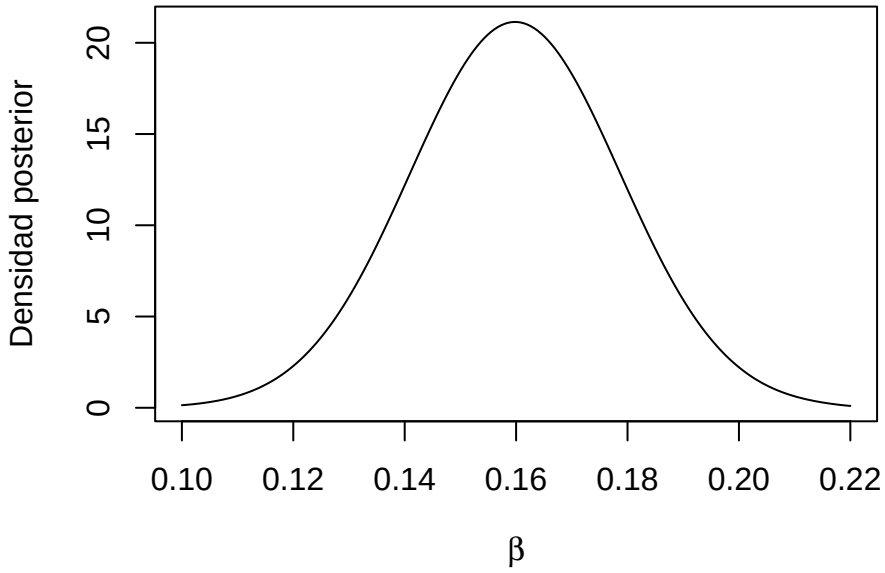
Una vez que está normalizada, podemos marginalizar sumando (integrando.)

```
# Para facilitar las cuentas, pondremos los valores de la posterior en una
# matriz, análoga a la grilla (beta varía entre filas, alpha, entre columnas)
post_mat <- matrix(grilla$post, side, side, byrow = T)

alpha_tab <- data.frame(
  alpha = alpha_seq,
  post = NA
)

for (i in 1:side) {
  alpha_tab$post[i] <- sum(post_mat[, i] * size_b)
}

plot(post ~ alpha, alpha_tab, type = "l", xlab = expression(alpha),
     ylab = "Densidad posterior")
```



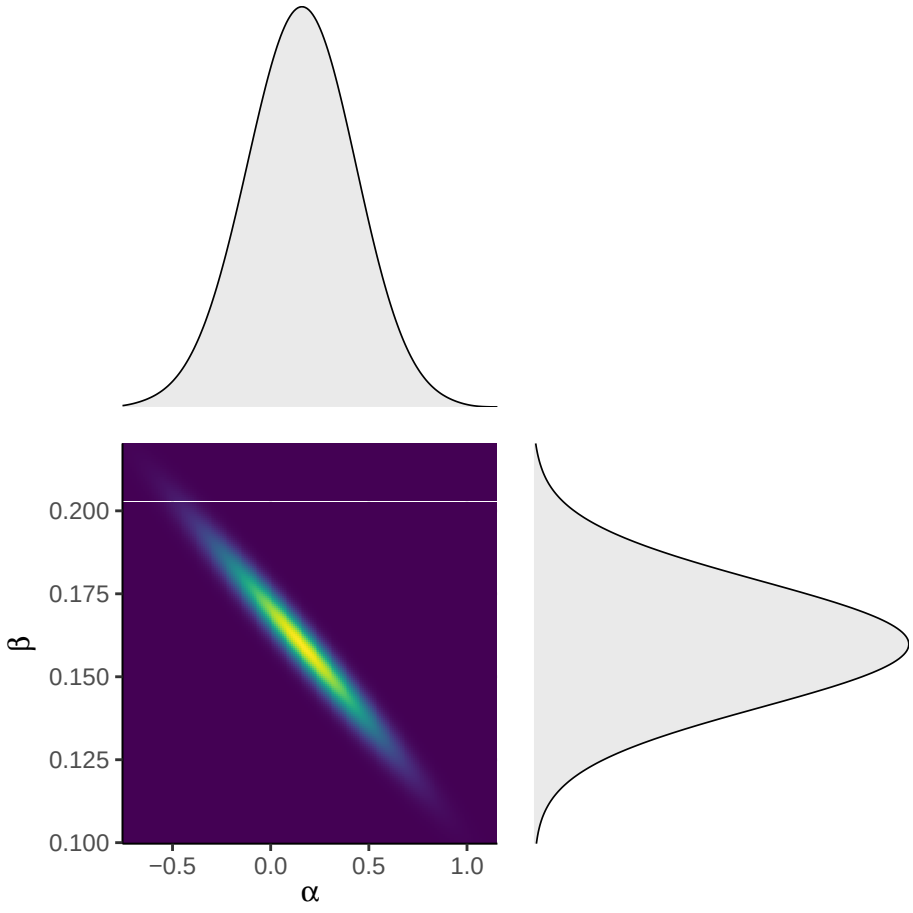
Ahora, la marginal de β:

```
beta_tab <- data.frame(
  beta = beta_seq,
  post = NA
)

for (i in 1:side) {
  beta_tab$post[i] <- sum(post_mat[i, ] * size_a)
}

plot(post ~ beta, beta_tab, type = "l", xlab = expression(beta),
     ylab = "Densidad posterior")
```




```
joint <-
ggplot(grilla, aes(x = alpha, y = beta, fill = post)) +
  geom_tile() +
  scale_fill_viridis() +
  scale_x_continuous(expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  theme(legend.position = "none") +
  xlab(expression(alpha)) +
  ylab(expression(beta)) +
  nice_theme()

ma <- ggplot(alpha_tab, aes(alpha, post, ymin = 0, ymax = post)) +
  geom_line(linewidth = 0.3) +
  geom_ribbon(color = NA, alpha = 0.1) +
  scale_x_continuous(expand = c(0, 0)) +
  scale_y_continuous(expand = c(0, 0)) +
  theme(axis.line = element_blank(),
        axis.text = element_blank(),
        axis.ticks = element_blank(),
        axis.title = element_blank())

mb <- ggplot(beta_tab, aes(beta, post, ymin = 0, ymax = post)) +
  geom_line(linewidth = 0.3) +
  geom_ribbon(color = NA, alpha = 0.1) +
  coord_flip() +
  scale_x_continuous(expand = c(0, 0)) +
```

```
  scale_y_continuous(expand = c(0, 0)) +
  theme(axis.line = element_blank(),
        axis.text = element_blank(),
        axis.ticks = element_blank(),
        axis.title = element_blank())

layout <- "
AA##
CCBB
"

ma + mb + joint + plot_layout(design = layout)
```



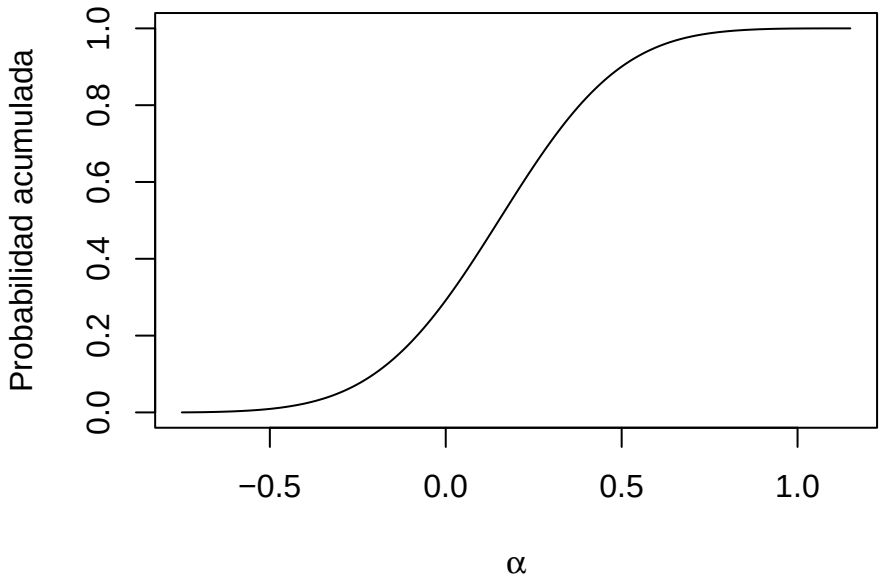
Desde acá podemos calcular promedios

```r
alpha_tab$mass <- alpha_tab$post * size_a # los pesos, suman 1
beta_tab$mass <- beta_tab$post * size_b

# Esperanza de alpha
ea <- sum(alpha_tab$alpha * alpha_tab$mass)

# Esperanza de beta
eb <- sum(beta_tab$beta * beta_tab$mass)

c(ea, eb)
```

```
[1] 0.1539278 0.1598357
```

Y cuantiles

```r
# obtenemos la función de distribución acumulada marginal para alpha
alpha_tab$cdf <- cumsum(alpha_tab$mass)
plot(cdf ~ alpha, alpha_tab, type = "l", xlab = expression(alpha),
     ylab = "Probabilidad acumulada")
```




```r
# creamos la función cuantil, es decir, la inversa de la acumulada.
# ajusta una spline. Al tomar como x la probabilidad acumulada (cdf) y como
# Y los valores de alpha, es la inversa de la acumulada (que es cdf = f(alpha))
alpha_icdf <- splinefun(x = alpha_tab$cdf, y = alpha_tab$alpha,
                        method = "monoH.FC")
# esta función tiene como argumento la probabilidad acumulada (entre 0 y 1),
# y devuelve el valor de alpha que acumula esa probabilidad (cuantil).
```

```r
probs <- c(0, 0.025, 0.5, 0.975, 1)
quants <- alpha_icdf(probs)
names(quants) <- paste(probs * 100, "%")
quants
```

```
       0 %      2.5 %       50 %     97.5 %      100 %
-0.7578735 -0.3917649  0.1509636  0.6796309  1.1500000
```

Todo esto puede ser inviable si la posterior tiene > 3 dimensiones (parámetros).

En algunos casos, la posterior tiene una solución analítica y además resulta en una distribución para la cual se conocen las marginales, con sus funciones de probabilidad acumulada y la inversa (función cuantil).

Pero eso es la excepción. Entonces, necesitamos formas más robustas de caracterizar las posteriores.

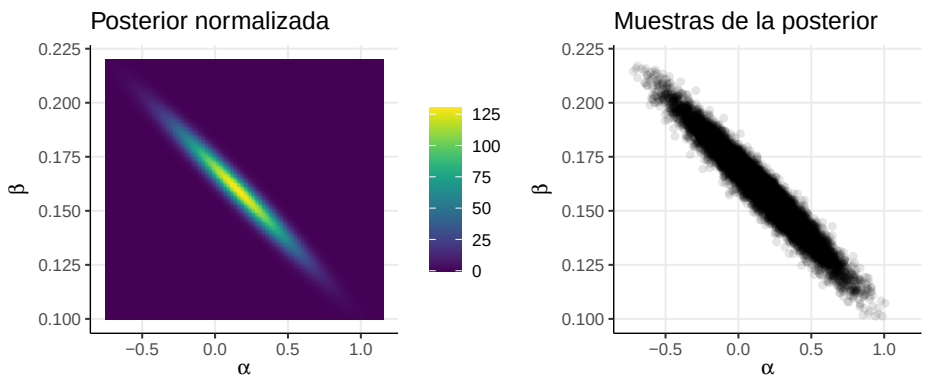
0.1 Enfoques para caracterizar distribuciones posteriores

- Solución analítica con integrales manejables
- Integración numérica
- Aproximaciones determinísticas (Inferencia Variacional, INLA, Aproximación de Laplace)
- Simulación

0.1.1 Simulación a partir de una grilla

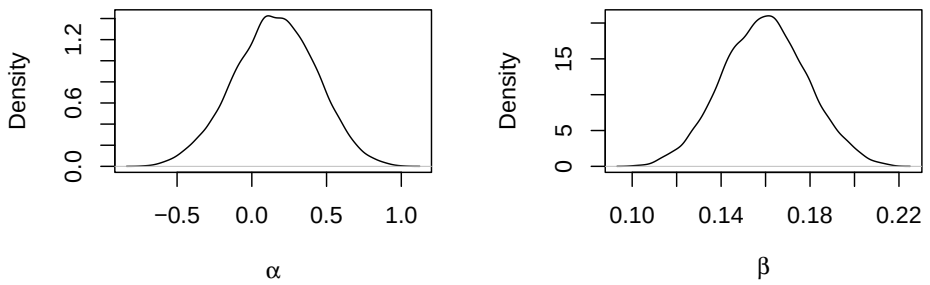
```r
# (El mismo gráfico de antes)
p1 <-
ggplot(grilla, aes(x = alpha, y = beta, fill = post)) +
  geom_tile() +
  scale_fill_viridis() +
  theme(legend.title = element_blank()) +
  xlab(expression(alpha)) +
  ylab(expression(beta)) +
  ggtitle("Posterior normalizada") +
  ylim(min(grilla$beta) - size_b, max(grilla$beta) + size_b) +
  xlim(min(grilla$alpha) - size_a, max(grilla$alpha) + size_a) +
  nice_theme()

# Remuestreamos celdas de la grilla con reemplazo, donde la probabilidad es
```

```r
# proporcional a la densidad posterior
nsim <- 10000 # número de muestras
rows_sim <- sample(1:size, size = nsim, replace = T, prob = grilla$post)

# Tomamos las filas que salieron sorteadas
draws_table <- grilla[rows_sim, ]

# Graficamos muestras
p2 <-
ggplot(draws_table, aes(x = alpha, y = beta)) +
  geom_point(alpha = 0.1) +
  ggtitle("Muestras de la posterior") +
  xlab(expression(alpha)) +
  ylab(expression(beta)) +
  ylim(min(grilla$beta) - size_b, max(grilla$beta) + size_b) +
  xlim(min(grilla$alpha) - size_a, max(grilla$alpha) + size_a) +
  nice_theme()


(p1 | p2)
```



A partir de las muestras es muy sencillo obtener las marginales y medidas de resumen:

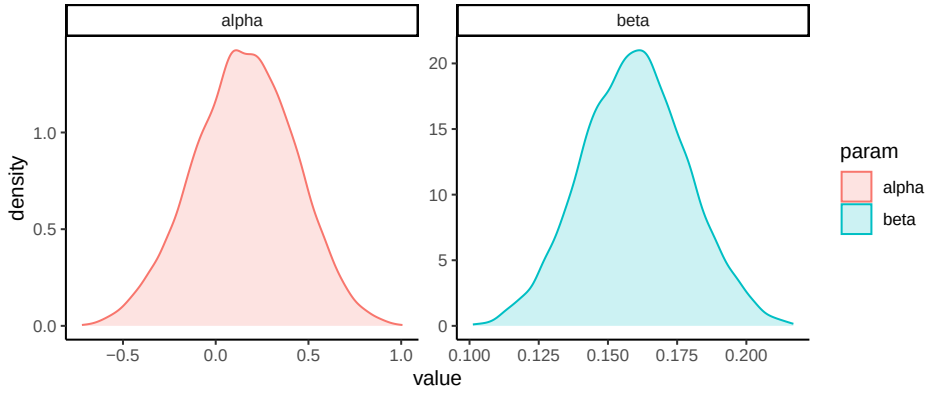
```r
# Cada fila en draws_table es una muestra de la posterior conjunta (bivariada).
# Si queremos ver la marginal de alpha o beta, simplemente miramos esa columna
# en draws_table, ignorando los valores de la otra variable. La forma más rápida
# de resumir una marginal es graficando su densidad empírica.

par(mfrow = c(1, 2))
plot(density(draws_table$alpha), xlab = expression(alpha), main = NA)
plot(density(draws_table$beta), xlab = expression(beta), main = NA)
```




```
par(mfrow = c(1, 1))

# También podemos calcular medidas de resumen:
summary(grilla[, c("alpha", "beta")])
```

```
     alpha              beta
 Min.   :-0.750   Min.   :0.10
 1st Qu.:-0.275   1st Qu.:0.13
 Median : 0.200   Median :0.16
 Mean   : 0.200   Mean   :0.16
 3rd Qu.: 0.675   3rd Qu.:0.19
 Max.   : 1.150   Max.   :0.22
```

```
# Para graficar ambas densidades en ggplot, alargamos:
draws_long <- pivot_longer(draws_table[, c("alpha", "beta")], # sólo esas columnas
                           cols = 1:2, names_to = "param", values_to = "value")

ggplot(draws_long, aes(x = value, color = param, fill = param)) +
  geom_density(alpha = 0.2) +
  facet_wrap(vars(param), scales = "free")
```



También podemos calcular la probabilidad posterior de cualquier tipo de afir-

mación sobre los parámetros o sobre funciones de los parámetros:

```
# Pr(alpha > 0)
sum(draws_table$alpha > 0) / nsim
```

```
[1] 0.7177
```

```
# Pr(0 < alpha < 0.5)
sum(draws_table$alpha > 0 & draws_table$alpha < 0.5) / nsim
```

```
[1] 0.611
```

```
# Pr(exp(beta) > 1.2)
ebeta <- exp(draws_table$beta)
sum(ebeta > 1.2) / nsim
```

```
[1] 0.1162
```

```
# Pr(lambda > 20 | FWI = 18) // lambda es la media
FWI <- 18
lambda <- exp(draws_table$alpha + draws_table$beta * FWI)
sum(lambda > 20) / nsim
```

```
[1] 0.6372
```

```
# Pr(y > 20 | FWI = 18) // y es la respuesta
FWI <- 18
lambda <- exp(draws_table$alpha + draws_table$beta * FWI)
ysim <- rpois(nsim, lambda)
sum(ysim > 20) / nsim
```

```
[1] 0.5069
```

Sin utilizar muestras, todo esto se calcularía evaluando integrales sobre las marginales o sobre transformaciones de las marginales.

0.2 Métodos de simulación

- Grilla
- Muestreo por rechazo (rejection sampling)
- Muestreo por importancia (importance sampling)
- Montecarlo basado en cadenas de Markov (MCMC)
- Montecarlo secuencial (SMC)

0.3 MCMC: Montecarlo basado en cadenas de Markov

Una cadena de Markov es una secuencia de variables aleatorias $\{X_1, X_2, X_3, \dots, X_t\}$ tal que

$$\Pr(X_{t+1}|X_t, X_{t-1}, X_{t-2}, \dots) = \Pr(X_{t+1}|X_t)$$

- MCMC consiste en simular una cadena markoviana cuya distribución estacionaria es la distribución posterior.
- Recorre el espacio de parámetros de forma estocástica, tomando muestras con una frecuencia proporcional a la probabilidad posterior.
- Genera una secuencia de muestras de la posterior conjunta, generalmente correlacionadas.

0.3.1 Propiedades de las cadenas de Markov requeridas en MCMC

1. Irreducibilidad: desde cualquier punto debo poder llegar, en algún momento, a cualquier otro punto en el espacio de parámetros.
2. Aperiodicidad: no entra en ciclos infinitos.
3. Recurrencia positiva: en algún momento, podés volver a un lugar en dondes estuviste.

Bajo estas condiciones, luego de muchas iteraciones, las muestras tomadas corresponden a la distribución objetivo (la posterior), sin importar dónde comenzó la cadena.

0.3.2 Algoritmo de Metropolis (el MCMC más simple)

- Defiinir el número de muestras a tomar (N), el valor inicial de los parámetros (θ_1), y una distribución de propuestas (d) simétrica. Evaluar $p(\theta_1|y)$ (densidad posterior probablemente no normalizada).
- Para i de 2 a N, repetir:
- Simular una muestra de la propuesta d centrada en θ_{i-1}. A esta muestra propuesta la llamaremos θ^*.
- Evaluar $p(\theta^*|y)$.
- Calcular $q = \frac{p(\theta^*|y)}{p(\theta_{i-1}|y)}$.

- Aceptar la propuesta ($\theta_i = \theta^*$) con probabilidad $\min(1, q)$. De lo contrario, definir $\theta_i = \theta_{i-1}$.

Metropolis en `R`

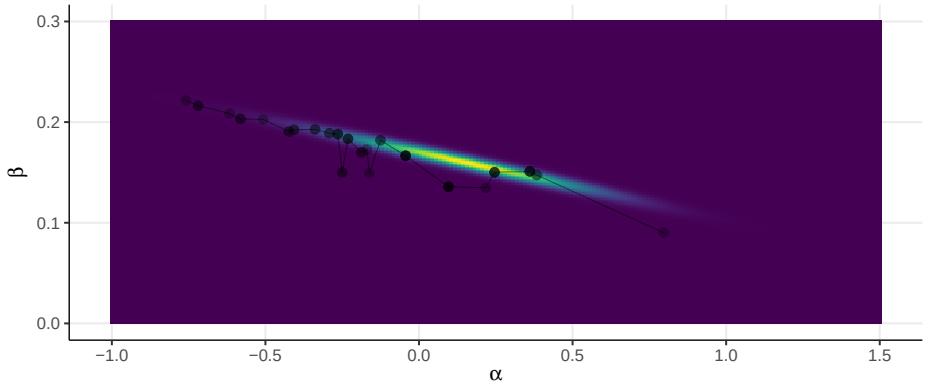
```r
# Definimos la distribución de propuesta como una normal bivariada, y
# aproximando la matriz de varianza-covarianza en base a la curvatura en el
# modo. Para ello, optimizamos y obtenemos la hessiana.

# optim necesita que la función a optimizar tome como argumento un vector
# de parámetros
post_fire_opt <- function(x) {
  alpha <- x[1]
  beta <- x[2]
  lp <- post_fire_un(alpha, beta)
  return(lp)
}

opt <- optim(
  par = c(0 , 0),               # vector inicial
  fn = post_fire_opt,           # función a optimizar
  method = "BFGS",              # método basado en gradientes (cool)
  control = list(fnscale = -1), # maximizar en vez de minimizar
  hessian = TRUE                # que calcule la hessiana
)

# La matriz de varianza-covarianza, asumiento normalidad es la inversa de la
# -hessiana
V <- solve(-opt$hessian)

# Para muestrear con una normal multivariada de propuesta, V puede escalarse
# por un factor, que según la teoría, lo óptimo es 2.38 ^ 2 / d, con d siendo
# la dimensión de la posterior
optimal_factor <- 2.38 ^ 2 / 2

# Creamos una función que genera una cadena markoviana para nuestra posterior.
# start es un vector con el valor inicial de alpha y beta. n es el número de
# muestras a tomar (incluye el start). factor es un factor que multiplica a
# la matriz de varianza-covarianza de la propuesta; un buen valor es 2, pero es
# interesante variarlo para ver cómo afecta la eficiencia.
mcmc_fire <- function(start, n, factor = optimal_factor) {
  # Matriz para guardar las muestras
  draws <- matrix(NA, nrow = n, ncol = 2)
  colnames(draws) <- c("alpha", "beta")
```

```r
  # Vector para almacenar la log posterior, necesaria para comparar valores
  lp <- numeric(n)

  # Iniciamos la matriz de muestras y la lp
  draws[1, 1:2] <- start
  lp[1] <- post_fire_un(start[1], start[2], log = T)

  # Loop recursivo para hacer avanzar las coordenadas
  for (i in 2:n) {
    # Proponemos un nuevo valor (vector), simulando de la distribución de
    # propuesta, centrada en draws[i-1, ] y con matriza de vcov V.
    prop <- rmvn(1, mu = draws[i-1, ], V * factor)

    # Evaluamos la probabilidad posterior en la propuesta
    lp_prop <- post_fire_un(prop[1], prop[2], log = T)

    # Calculamos el cociente de densidad posterior propuesta / valor anterior.
    # En escala log, la resta equivale al cociente:
    lp_diff <- lp_prop - lp[i-1]
    p_q <- exp(lp_diff)
    # equivale a
    # p_q = exp(lp_prop) / exp(lp[i-1])
    # pero en log es más estable

    # Aceptamos si p_q > 1 (mayor densidad en la propuesta), y sorteamos que
    # la propuesta se acepte, con p_q si p_q es < 1.
    pkeep <- min(1, p_q)
    keep_prop <- ifelse(runif(1) < pkeep, T, F)

    if (keep_prop) { # si aceptamos la propuesta, la guardamos
      draws[i, ] <- prop
      lp[i] <- lp_prop   # para comparar con el siguiente
    } else {         # si no aceptamos, repetimos la muestra anterior
      draws[i, ] <- draws[i-1, ]
      lp[i] <- lp[i-1]   # para comparar con el siguiente
    }
  }

  return(draws)
}
```

Simulamos una cadena corta

```r
# Para graficar sobre la grilla, la extendemos un poco. Esto permite ver cómo la
# cadena se mueve por zonas de baja densidad también, probablemente porque la
# iniciamos en una zona de baja densidad.
alpha_seq <- seq(-1, 1.5, length.out = side)  # límites más amplios que antes
beta_seq <- seq(0, 0.3, length.out = side)

size_a <- diff(alpha_seq)[1]
size_b <- diff(beta_seq)[1]

grilla <- expand.grid(
  alpha = alpha_seq,
  beta = beta_seq
)

for (i in 1:size) {
  grilla$post[i] <- post_fire_un(grilla$alpha[i], grilla$beta[i], log = F)
}
```

```r
# Corremos una cadena corta comenzando dentro de la grilla
run1 <- mcmc_fire(start = c(-0.25, 0.15), n = 50)
draws_table <- as.data.frame(run1)

# Graficamos
ggplot(grilla, aes(x = alpha, y = beta, fill = post)) +
  geom_tile() +
  scale_fill_viridis() +
  geom_point(data = draws_table, mapping = aes(x = alpha, y = beta),
             inherit.aes = F, size = 2, alpha = 0.3) +
  geom_line(data = draws_table, mapping = aes(x = alpha, y = beta),
             inherit.aes = F, alpha = 0.5, linewidth = 0.2) +
  theme(legend.position = "none") +
  xlab(expression(alpha)) +
  ylab(expression(beta)) +
  ylim(min(grilla$beta) - size_b, max(grilla$beta) + size_b) +
  xlim(min(grilla$alpha) - size_a, max(grilla$alpha) + size_a) +
  nice_theme()
```

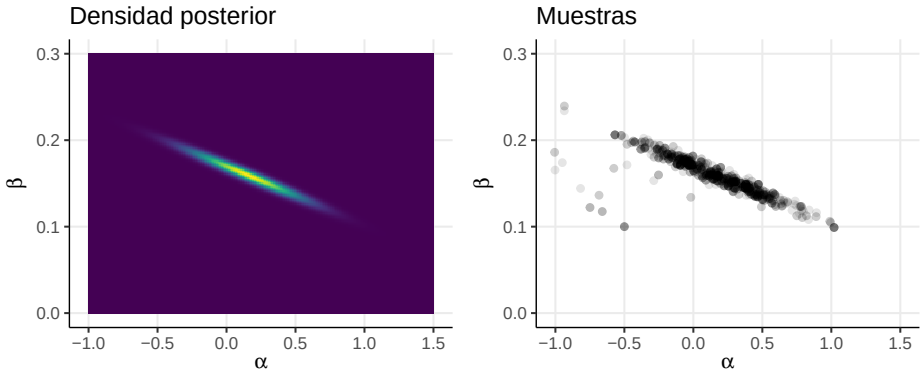


Ahora un poco más

```
nsim <- 1000
set.seed(12359) # para reproducibilidad
run1 <- mcmc_fire(start = c(-0.5, 0.1), n = nsim)
draws_table <- as.data.frame(run1)

p1 <-
ggplot(grilla, aes(x = alpha, y = beta, fill = post)) +
  geom_tile() +
  scale_fill_viridis() +
  theme(legend.position = "none") +
  ggtitle("Densidad posterior") +
  xlab(expression(alpha)) +
  ylab(expression(beta)) +
  ylim(min(grilla$beta) - size_b, max(grilla$beta) + size_b) +
  xlim(min(grilla$alpha) - size_a, max(grilla$alpha) + size_a) +
  nice_theme()

p2 <-
ggplot(draws_table, aes(x = alpha, y = beta)) +
  geom_point(alpha = 0.1) +
  ggtitle("Muestras") +
  xlab(expression(alpha)) +
  ylab(expression(beta)) +
  ylim(min(grilla$beta) - size_b, max(grilla$beta) + size_b) +
  xlim(min(grilla$alpha) - size_a, max(grilla$alpha) + size_a) +
  nice_theme()

(p1 | p2)
```

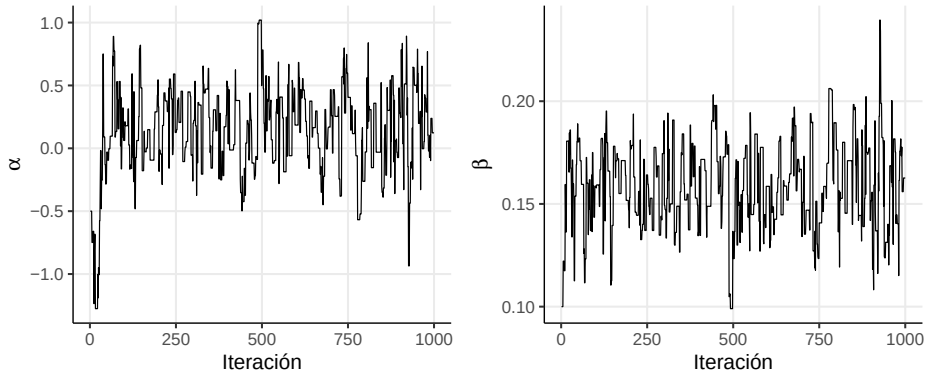
```
Warning: Removed 9 rows containing missing values or values outside the scale range
```

(`geom_point()`).



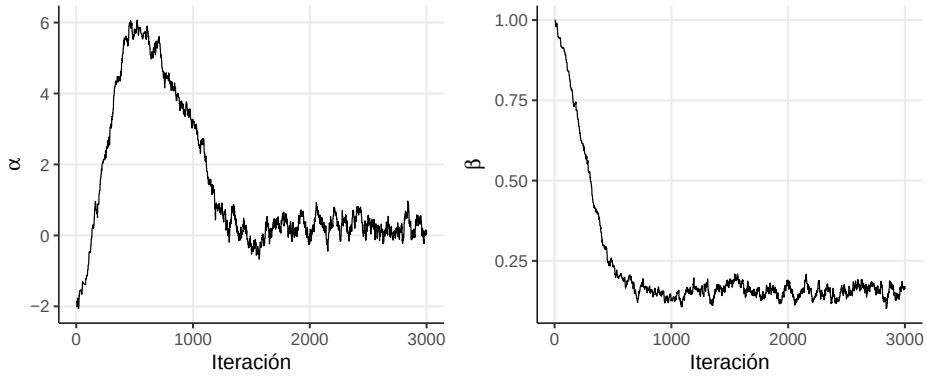
Visualizamos la serie temporal

```
draws_table$time <- 1:nsim

p3 <-
  ggplot(draws_table, aes(time, alpha)) +
  geom_line(linewidth = 0.3) +
  nice_theme() +
  ylab(expression(alpha)) +
  xlab("Iteración")

p4 <-
  ggplot(draws_table, aes(time, beta)) +
  geom_line(linewidth = 0.3) +
  nice_theme() +
  ylab(expression(beta)) +
  xlab("Iteración")

(p3 | p4)
```



Achicamos la amplitud de la propuesta para que se mueva lento, y le hacemos comenzar en una zona con baja probabilidad

```
nn <- 3000
run2 <- mcmc_fire(start = c(-2, 1), n = nn, factor = 0.1)
draws_table <- as.data.frame(run2)
draws_table$time <- 1:nn

p3 <-
  ggplot(draws_table, aes(time, alpha)) +
  geom_line(linewidth = 0.3) +
  nice_theme() +
  ylab(expression(alpha)) +
  xlab("Iteración")

p4 <-
  ggplot(draws_table, aes(time, beta)) +
  geom_line(linewidth = 0.3) +
  nice_theme() +
  ylab(expression(beta)) +
  xlab("Iteración")

(p3 | p4)
```



0.3.3 Dianósticos de MCMC

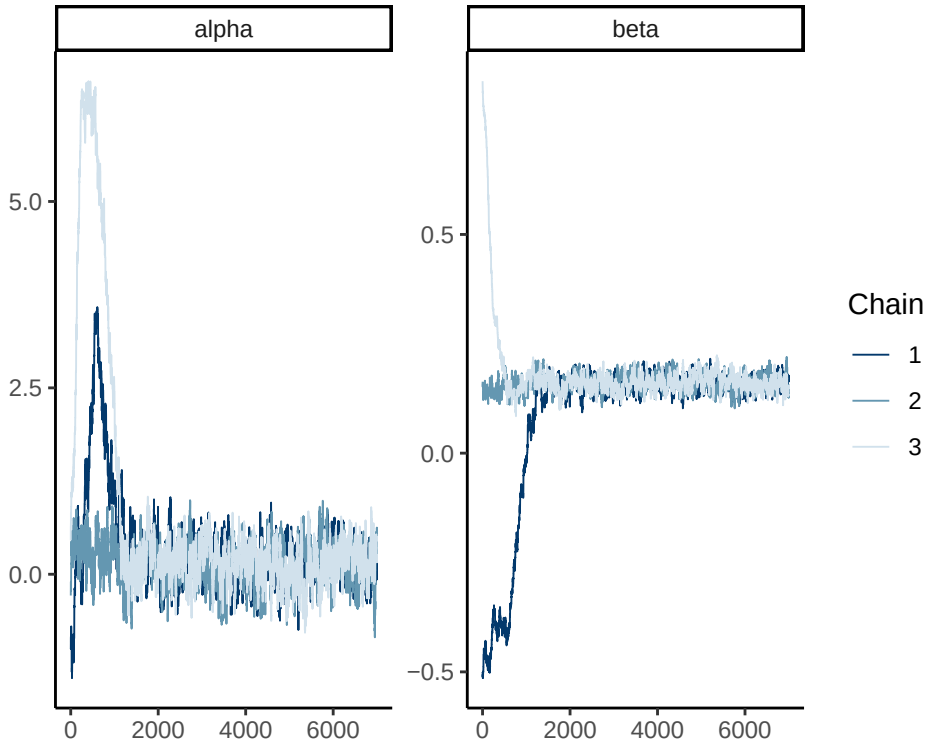
[Uso interactivo, vaya a R]

```
# Corremos 3 cadenas de igual largo comenzando en puntos al azar
iter_warmup <- 2000
iter_sampling <- 5000
n <- iter_warmup + iter_sampling
f <- optimal_factor * 0.05

c1 <- mcmc_fire(start = runif(2, -1, 1), n = n, factor = f)
c2 <- mcmc_fire(start = runif(2, -1, 1), n = n, factor = f)
c3 <- mcmc_fire(start = runif(2, -1, 1), n = n, factor = f)

# Las Combinamos en un array 3D, ya que son 3 matrices
arr <- abind::abind(list(c1, c2, c3), along = 3)
dimnames(arr) <- list(
  iteration = 1:n,
  variable = c("alpha", "beta"),
  chain = 1:3
)

# El paquete posterior requiere que las dimensiones sean iteration, chain,
# variable, así que reordenamos:
arr <- aperm(arr, c(1, 3, 2))

# Converimos draws_array object, clase del paquete posterior
draws <- as_draws_array(arr)
summarize_draws(draws)
```

```
# A tibble: 2 x 10
  variable  mean median    sd    mad      q5   q95  rhat ess_bulk ess_tail
  <chr>    <dbl>  <dbl> <dbl>  <dbl>   <dbl> <dbl> <dbl>    <dbl>    <dbl>
```

```
1 alpha    0.434  0.198 1.11  0.324  -0.329  2.47   1.06     36.4     17.5
2 beta     0.141  0.158 0.123 0.0222  0.0486 0.197  1.09     24.7     11.1
```

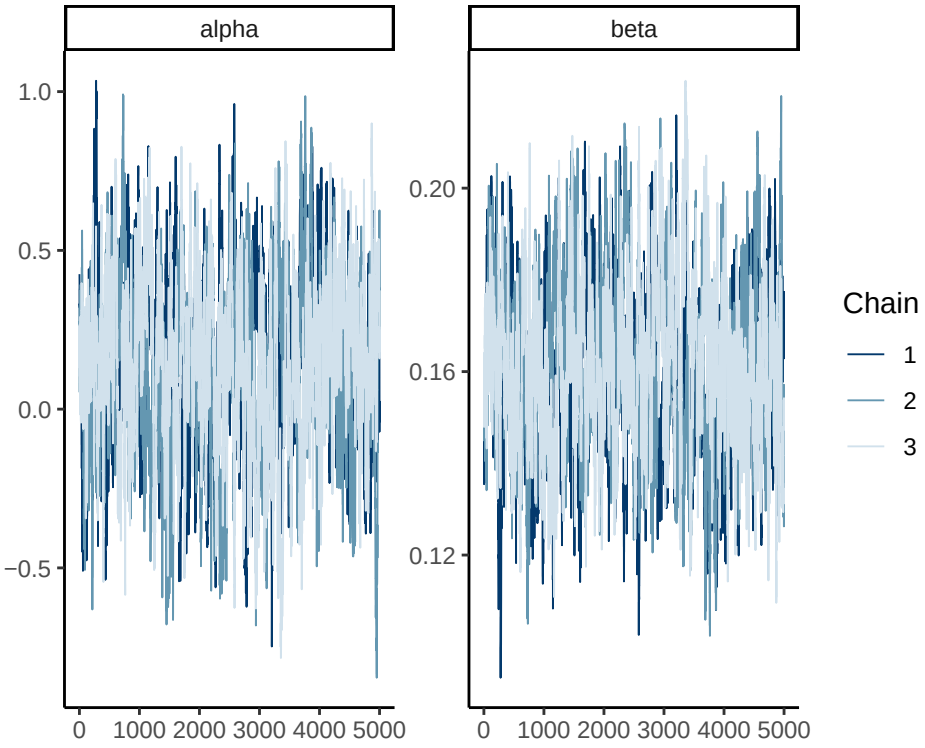
```
mcmc_trace(draws) # de bayesplot
```




```
# Quitamos el warmup, porque aún no había llegado al set típico (nos quedamos
# con las últimas iter_sampling de cada cadena)
iters_use <- (iter_warmup + 1) : n
draws_crop <- draws[iters_use, , ]
summarize_draws(draws_crop)
```

```
# A tibble: 2 x 10
  variable  mean median     sd    mad     q5   q95  rhat ess_bulk ess_tail
  <chr>    <dbl>  <dbl>  <dbl>  <dbl>  <dbl> <dbl> <dbl>    <dbl>    <dbl>
1 alpha    0.134  0.144 0.280  0.284  -0.344 0.577  1.01     270.     615.
2 beta     0.161  0.161 0.0191 0.0199  0.130 0.193  1.01     287.     688.
```

```
mcmc_trace(draws_crop)
```



0.3.4 $\hat{R}$ y Número Efectivo de Muestras (ESS)

- El $\hat{R}$ (factor de reducción potencial de escala) verifica que las cadenas se hayan mezclado bien:
  - $\hat{R} \to 1$ si hay convergencia.
  - Umbral recomendado: $\hat{R} < 1.01$.
- El número efectivo de muestras (ESS) indica a cuántas muestras independientes equivale tu muestra:
  - Considera la autocorrelación temporal en las cadenas.
  - Se recomienda ESS > 400 para estimaciones confiables (pero depende del contexto).

0.3.4.1 Definiciones:

Varianza entre cadenas (B de between):

$$B = \frac{N}{M-1} \sum_{m=1}^{M} \left(\bar{\theta}_m - \bar{\theta}\right)^2,$$

con N = número de muestras por cadena,
M = número de cadenas,
$\bar{\theta}_m$ = media por cadena, y
$\bar{\theta}$ = media total.

Varianza intra-cadenas (W de within):

$$W = \frac{1}{M} \sum_{m=1}^{M} s_m^2,$$

siendo s_m^2 la varianza de la cadena m.

Varianza posterior marginal estimada:

$$\widehat{\text{var}}^+(\theta) = \frac{N-1}{N} W + \frac{1}{N} B$$

Factor de reducción potencial de escala:

$$\hat{R} = \sqrt{\frac{\widehat{\text{var}}^+(\theta)}{W}}$$

Número Efectivo de Muestras:

$$\text{ESS} = \frac{MN}{\hat{\tau}}$$

donde $\hat{\tau}$ es una estimación de la autocorrelación temporal integrada en muchos lags.

Detalles en

Aki Vehtari, Andrew Gelman, Daniel Simpson, Bob Carpenter, and Paul-Christian Bürkner. 2021. Rank-Normalization, Folding, and Localization: An Improved $\hat{R}$ for Assessing Convergence of MCMC. Bayesian Analysis 16, Number 2, pp. 667–718.

Galería de MCMC

0.4 MCMC eficiente: Montecarlo Hamiltoniano (HMC)

- Propuestas inteligentes: utilizando la curvatura de la densidad posterior (gradiente) logra dar pasos grandes manteniéndose en la región de alta probabilidad (*set típico*).

- Para generar estas propuetas utiliza una analogía física: simulando el moviemiento de un cuerpo que se mueve en la superficie (N-dimensional) de la -log posterior siguiendo una dinámica Hamiltoniana.
- Escala muy bien con la dimensionalidad (nro. de parámetros).
- Como requiere calcular el gradiente, no se puede usar para estimar directamente parámetros discretos (hay que marginalizarlos).

0.5 Stan: HMC-NUTS y un poco más

C++ Stan {R, Python, Julia, Unix}

Algoritmos en Stan

Guía de usuarios

0.5.1 Programando un modelo en Stan

- Un programa de Stan define una log densidad que deseamos caracterizar.
- En la mayor parte de los casos se tratará de una densidad posterior.
- En este curso usaremos Stan para muestrear (HMC-NUTS), pero también ofrece otros métodos menos precisos pero más rápidos que valen la pena cuando tenemos sets de datos muy grandes o funciones de verosimilitud muy costosas.

0.5.2 El código

Stan divide el código en 6 secciones, pero rara vez usaremos todas

```
functions {}
data {}
transformed data {}
parameters {}
transformed parameters {}
model {}
generated quantities {}
```

Ejemplo

```
// (Esto se guarda en un archivo .stan)
data {
  int N;
  array[N] int y; // Número de incendios
  vector[N] x; // FWI
}

// Parámetros a muestrear
parameters {
  real alpha;
  real beta;
}

// Calculamos las cantidades derivadas
transformed parameters {
  vector[N] lambda = exp(alpha + beta * x);
}

// Acá definimos la log densidad posterior (o la que sea)
model {
  // Densidad previa
  alpha ~ normal(0, 1);
  beta ~ normal(0, 0.1);

  // Verosimilitud
  y ~ poisson(lambda);
}
```

Formas de definir la log densidad posterior

De forma expresiva, con *sampling statements*:

```
model {
  // Densidad previa
  alpha ~ normal(0, 1);
  beta ~ normal(0, 0.1);

  // Verosimilitud
  y ~ poisson(lambda);

  /*
    Esto equivale a decir "sumá a la log densidad posterior la
    log densidad de alpha en una normal con media 0 y desvío 1, la
    log densidad de beta en una normal con media 0 y desvío 0.1, y la
    log verosimilitud (conjunta) de observar 'y' (es un vector) como realización
```

```
    de una Poisson cuyas medias son lambda (vector también).
  */
}
```

De forma explícita, sumando términos a `target`:

```
model{
  /*
    Stan guarda la log densidad objetivo en un objeto llamado 'target'.
    Cada vez que se evalúa la log posterior, target comienza valiendo 0.
    Esta parte de la función se encarga de sumarle los términos correspondientes.
  */

  // Sumar log densidad previa
  target += normal_lpdf(alpha | 0, 1);
  target += normal_lpdf(beta | 0, 0.1);

  // Sumar log verosimilitud
  target += poisson_lpmf(y | lambda);

  // lpdf: Log Probability Density Function
  // lpmf: Log Probability Mass Function
}
```

Y se pueden mezclar

```
model{
  /*
    Podemos mezclar ambas formas, pero siempre hay que usar una sola por
    parámetro. (Si no, estará duplicada, lo cual es computacionalmente válido
    pero sería sospechoso que tenga sentido.)
  */

  // Vale hacer
  alpha ~ normal(0, 1);
  target += normal_lpdf(beta | 0, 0.1);
  y ~ poisson(lambda);

  /*
    Las funciones de densidad o verosimilitudes suelen tener constantes
    normalizadoras. Usando target +=, Stan las conserva, pero usando los
    sampling statements (~), los quita de la función, ahorrando un poco de
    cómputo. En algunos casos necesitamos conservar esas constantes para
    hacer una cuenta, y ahí conviene usar target +=.
  */
}
```

Analogía con R

```
# Una función que evalúa la log posterior de nuestro modelo con un estilo
# similar a Stan
post_fire_like_stan <- function(alpha, beta) {
  # Datos
  y <- datos$fires
  x <- datos$fwi

  # Parámetros
  # alpha y beta, pero en R no hay que declarar variables :) ... o :(, no sé

  # Parámetros transformados
  lambda <- exp(alpha + beta * x)

  # Modelo (definimos log densidad posterior)

  target <- 0 # iniciamos la log posterior en cero

  # Log densidad de las previas
  target <- target + dnorm(alpha, mean = 0, sd = 1, log = T)
  target <- target + dnorm(beta, mean = 0, sd = 0.1, log = T)
  # Es una analogía con el operador += de Stan

  # Log verosimilitud
  log_like_pointwise <- dpois(y, lambda, log = T) # dato por dato
  log_like_joint <- sum(log_like_pointwise)       # conjunta
  target <- target + log_like_joint

  return(target)
}
```

0.5.3 Interfaz entre R y Stan: cmdstanr

```
# El código de Stan, como escribimos arriba, debe estar guardado en un archivo
# .stan.

# Compilamos el modelo
model <- cmdstan_model(here::here("modelos", "nfuegos.stan"))
```

```
Warning in readLines(stan_file): incomplete final line found on
'/home/ivan/Insync/Curso Modelos y Datos - CRUB -
Gure25/curso-bayes-25/modelos/nfuegos.stan'
```

```r
# Podemos revisar si lo escribimos bien
model$check_syntax()
```

```
Stan program is syntactically correct
```

```r
# Creamos una lista nombrada para pasarle los datos. Los nombres de cada
# elemento de la lista tienen que ser exactamente los que definimos en
# la sección data {}
stan_data <- list(
  N = nrow(datos),
  y = datos$fires,
  x = datos$fwi
)

# Y muestreamos la posterior
fit_mcmc <- model$sample(
  data = stan_data,
  chains = 4,
  parallel_chains = 4,
  iter_warmup = 1000,
  iter_sampling = 1000
)
```

```
Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 1 Iteration:  100 / 2000 [  5%]  (Warmup)
Chain 1 Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain 1 Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain 1 Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain 1 Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain 1 Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain 1 Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain 1 Iteration:  800 / 2000 [ 40%]  (Warmup)
Chain 1 Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain 1 Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain 1 Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain 1 Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain 1 Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain 1 Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain 1 Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain 1 Iteration: 1800 / 2000 [ 90%]  (Sampling)
Chain 1 Iteration: 1900 / 2000 [ 95%]  (Sampling)
```

```
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 2 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 2 Iteration:  100 / 2000 [  5%]  (Warmup)
Chain 2 Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain 2 Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain 2 Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain 2 Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain 2 Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain 2 Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain 2 Iteration:  800 / 2000 [ 40%]  (Warmup)
Chain 2 Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain 2 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 2 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 2 Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain 2 Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain 2 Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain 2 Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain 2 Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain 2 Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain 2 Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain 2 Iteration: 1800 / 2000 [ 90%]  (Sampling)
Chain 2 Iteration: 1900 / 2000 [ 95%]  (Sampling)
Chain 2 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 3 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 3 Iteration:  100 / 2000 [  5%]  (Warmup)
Chain 3 Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain 3 Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain 3 Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain 3 Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain 3 Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain 3 Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain 3 Iteration:  800 / 2000 [ 40%]  (Warmup)
Chain 3 Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain 3 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 3 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 3 Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain 3 Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain 3 Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain 3 Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain 3 Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain 3 Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain 3 Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain 3 Iteration: 1800 / 2000 [ 90%]  (Sampling)
Chain 3 Iteration: 1900 / 2000 [ 95%]  (Sampling)
Chain 3 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 4 Iteration:    1 / 2000 [  0%]  (Warmup)
```

```
Chain 4 Iteration:  100 / 2000 [  5%]  (Warmup)
Chain 4 Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain 4 Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain 4 Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain 4 Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain 4 Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain 4 Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain 4 Iteration:  800 / 2000 [ 40%]  (Warmup)
Chain 4 Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain 4 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 4 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 4 Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain 4 Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain 4 Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain 4 Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain 4 Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain 4 Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain 4 Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain 4 Iteration: 1800 / 2000 [ 90%]  (Sampling)
Chain 4 Iteration: 1900 / 2000 [ 95%]  (Sampling)
Chain 4 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 1 finished in 0.1 seconds.
Chain 2 finished in 0.1 seconds.
Chain 3 finished in 0.1 seconds.
Chain 4 finished in 0.1 seconds.

All 4 chains finished successfully.
Mean chain execution time: 0.1 seconds.
Total execution time: 0.3 seconds.
```

```
# Corremos los diagnósticos del muestreo. Si hay problemas, por defecto
# aparecen warnings cuando sample() termina de correr, pero así podemos verlos
# de nuevo.
fit_mcmc$cmdstan_diagnose()
```

```
Checking sampler transitions treedepth.
Treedepth satisfactory for all transitions.

Checking sampler transitions for divergences.
No divergent transitions found.

Checking E-BFMI - sampler transitions HMC potential energy.
E-BFMI satisfactory.

Rank-normalized split effective sample size satisfactory for all parameters.
```

```
Rank-normalized split R-hat values satisfactory for all parameters.

Processing complete, no problems detected.
```

```
# Resumimos las marginales usando el paquete 'posterior' summaries
summ <- fit_mcmc$summary()
print(summ)
```

```
# A tibble: 27 x 10
   variable     mean  median     sd    mad      q5     q95  rhat ess_bulk
   <chr>       <dbl>   <dbl>  <dbl>  <dbl>   <dbl>   <dbl> <dbl>    <dbl>
 1 lp__       340.    340.    1.00   0.744  338.    341.     1.01    1016.
 2 alpha        0.135   0.141 0.281  0.288   -0.322   0.582  1.01     626.
 3 beta         0.161   0.161 0.0192 0.0195   0.130   0.192  1.01     610.
 4 lambda[1]   12.3    12.3   0.799  0.818   11.0    13.6    1.00    3717.
 5 lambda[2]    5.22    5.20  0.592  0.602    4.29    6.21   1.01     833.
 6 lambda[3]    4.27    4.24  0.572  0.581    3.38    5.23   1.01     745.
 7 lambda[4]   13.6    13.6   0.925  0.919   12.2    15.2    1.00    3066.
 8 lambda[5]    7.36    7.34  0.610  0.624    6.38    8.36   1.00    1284.
 9 lambda[6]    7.61    7.59  0.613  0.624    6.62    8.61   1.00    1359.
10 lambda[7]    6.31    6.29  0.603  0.614    5.35    7.30   1.01    1015.
# i 17 more rows
# i 1 more variable: ess_tail <dbl>
```

```
# Cuál es el peor neff y el peor rhat?
min(c(min(summ$ess_bulk), min(summ$ess_tail)))
```

```
[1] 609.9725
```

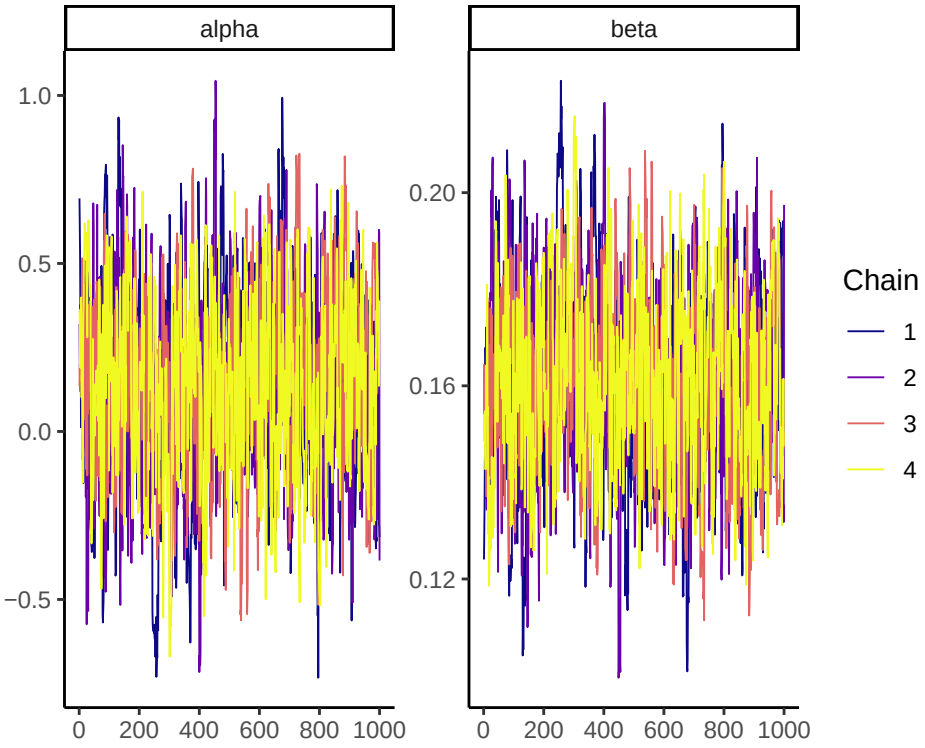
```
max(summ$rhat)
```

```
[1] 1.010698
```

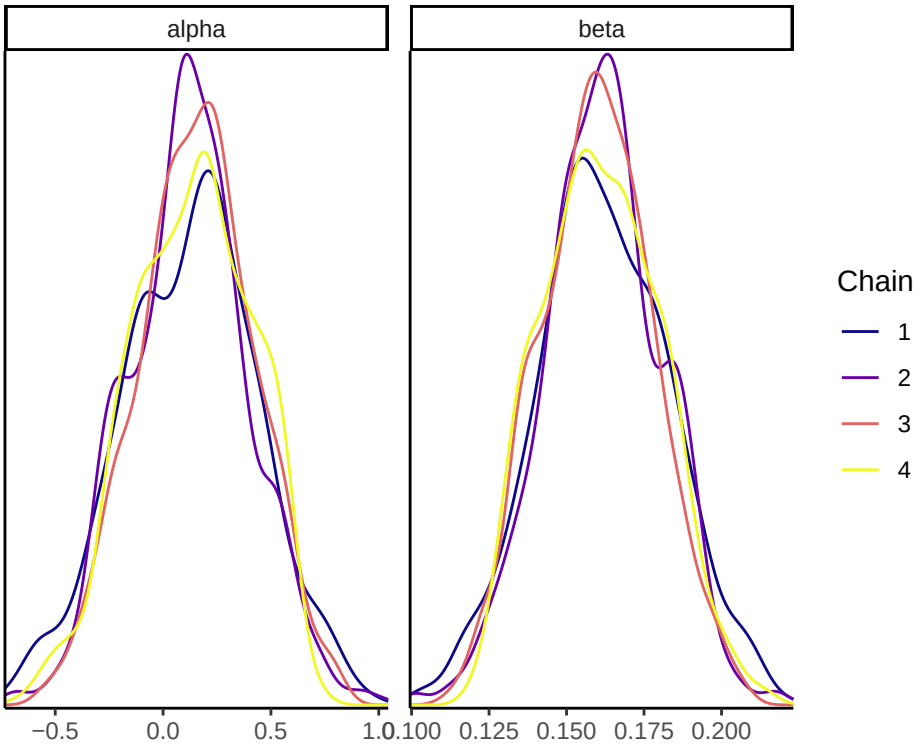
Diagnósticos del MCMC

Algunas herramientas de visualización usando bayesplot

```
# Seteamos un lindo color para bayesplot
color_scheme_set("viridisC")

# Extraemos las muestras y las formateamos como draws_array, una clase del
# paquete 'posterior'
draws <- fit_mcmc$draws(format = "draws_array")

# Gráfico de traza, para visualizar convergencia
mcmc_trace(draws, pars = c("alpha", "beta"))
```




```
# Gráfico de densidad por cadena
mcmc_dens_overlay(draws, pars = c("alpha", "beta"))
```



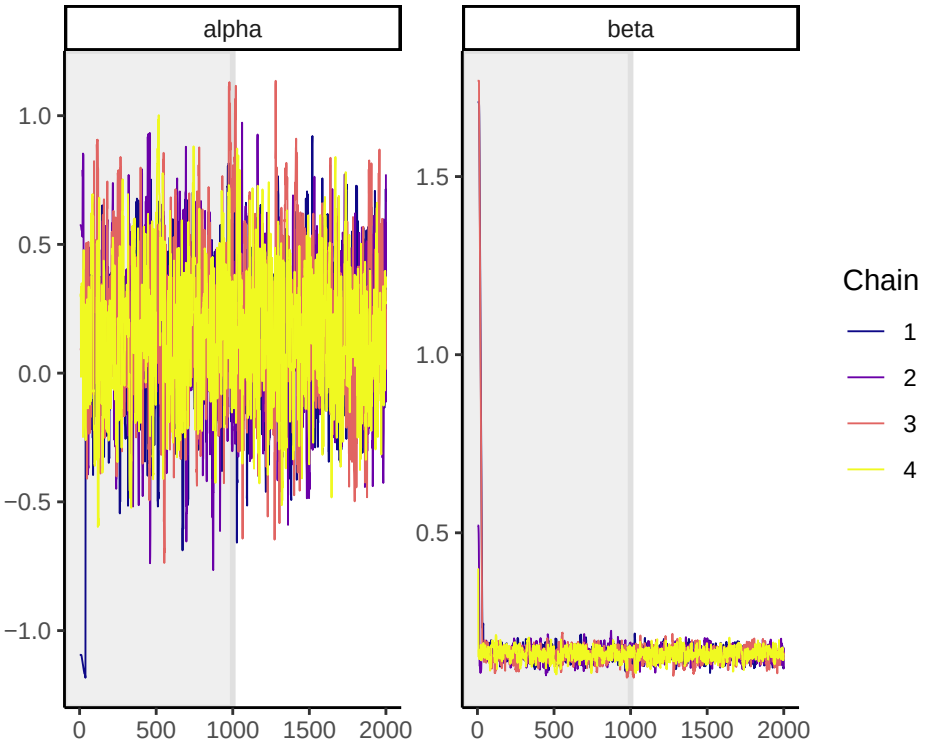

```
# Sólo para ilustrar, reajustamos el modelo pero guardando las iteraciones
# del warmup.
fit_mcmc_all <- model$sample(
  data = stan_data,
  chains = 4,
  parallel_chains = 4,
  iter_warmup = 1000,
  iter_sampling = 1000,
  save_warmup = TRUE   # por defecto es FALSE
)
```

```
Running MCMC with 4 parallel chains...

Chain 1 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 1 Iteration:  100 / 2000 [  5%]  (Warmup)
Chain 1 Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain 1 Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain 1 Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain 1 Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain 1 Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain 1 Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain 1 Iteration:  800 / 2000 [ 40%]  (Warmup)
```

```
Chain 1 Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain 1 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 1 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 1 Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain 1 Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain 1 Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain 1 Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain 1 Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain 1 Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain 1 Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain 1 Iteration: 1800 / 2000 [ 90%]  (Sampling)
Chain 1 Iteration: 1900 / 2000 [ 95%]  (Sampling)
Chain 1 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 2 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 2 Iteration:  100 / 2000 [  5%]  (Warmup)
Chain 2 Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain 2 Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain 2 Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain 2 Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain 2 Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain 2 Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain 2 Iteration:  800 / 2000 [ 40%]  (Warmup)
Chain 2 Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain 2 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 2 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 2 Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain 2 Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain 2 Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain 2 Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain 2 Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain 2 Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain 2 Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain 2 Iteration: 1800 / 2000 [ 90%]  (Sampling)
Chain 2 Iteration: 1900 / 2000 [ 95%]  (Sampling)
Chain 2 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 3 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 3 Iteration:  100 / 2000 [  5%]  (Warmup)
Chain 3 Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain 3 Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain 3 Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain 3 Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain 3 Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain 3 Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain 3 Iteration:  800 / 2000 [ 40%]  (Warmup)
Chain 3 Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain 3 Iteration: 1000 / 2000 [ 50%]  (Warmup)
```

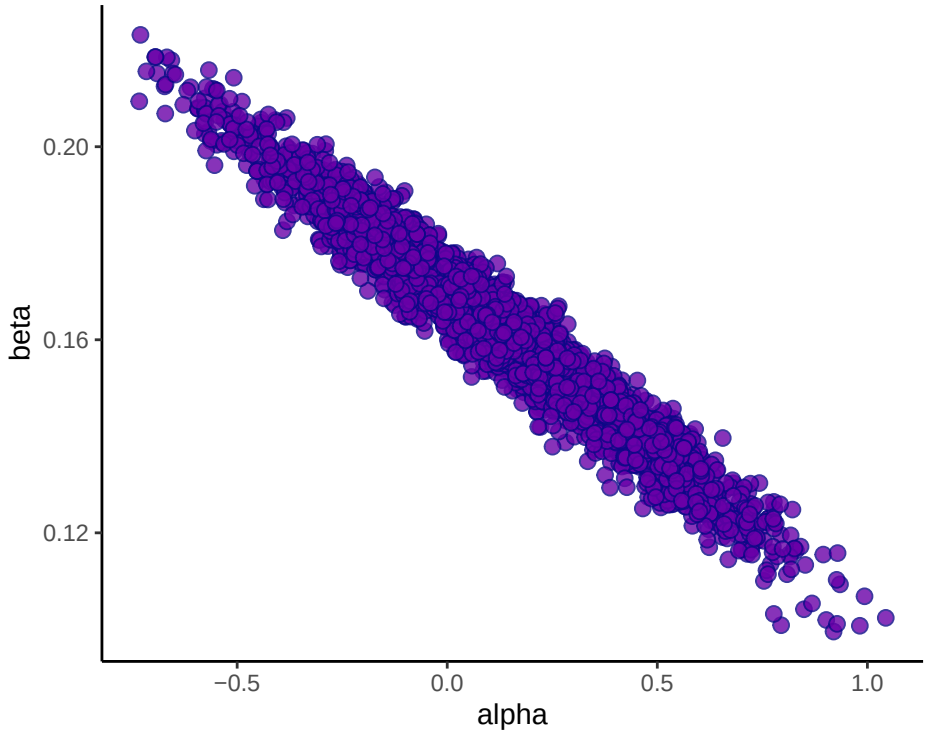
```
Chain 3 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 3 Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain 3 Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain 3 Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain 3 Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain 3 Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain 3 Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain 3 Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain 3 Iteration: 1800 / 2000 [ 90%]  (Sampling)
Chain 3 Iteration: 1900 / 2000 [ 95%]  (Sampling)
Chain 3 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 4 Iteration:    1 / 2000 [  0%]  (Warmup)
Chain 4 Iteration:  100 / 2000 [  5%]  (Warmup)
Chain 4 Iteration:  200 / 2000 [ 10%]  (Warmup)
Chain 4 Iteration:  300 / 2000 [ 15%]  (Warmup)
Chain 4 Iteration:  400 / 2000 [ 20%]  (Warmup)
Chain 4 Iteration:  500 / 2000 [ 25%]  (Warmup)
Chain 4 Iteration:  600 / 2000 [ 30%]  (Warmup)
Chain 4 Iteration:  700 / 2000 [ 35%]  (Warmup)
Chain 4 Iteration:  800 / 2000 [ 40%]  (Warmup)
Chain 4 Iteration:  900 / 2000 [ 45%]  (Warmup)
Chain 4 Iteration: 1000 / 2000 [ 50%]  (Warmup)
Chain 4 Iteration: 1001 / 2000 [ 50%]  (Sampling)
Chain 4 Iteration: 1100 / 2000 [ 55%]  (Sampling)
Chain 4 Iteration: 1200 / 2000 [ 60%]  (Sampling)
Chain 4 Iteration: 1300 / 2000 [ 65%]  (Sampling)
Chain 4 Iteration: 1400 / 2000 [ 70%]  (Sampling)
Chain 4 Iteration: 1500 / 2000 [ 75%]  (Sampling)
Chain 4 Iteration: 1600 / 2000 [ 80%]  (Sampling)
Chain 4 Iteration: 1700 / 2000 [ 85%]  (Sampling)
Chain 4 Iteration: 1800 / 2000 [ 90%]  (Sampling)
Chain 4 Iteration: 1900 / 2000 [ 95%]  (Sampling)
Chain 4 Iteration: 2000 / 2000 [100%]  (Sampling)
Chain 1 finished in 0.1 seconds.
Chain 2 finished in 0.1 seconds.
Chain 3 finished in 0.1 seconds.
Chain 4 finished in 0.1 seconds.

All 4 chains finished successfully.
Mean chain execution time: 0.1 seconds.
Total execution time: 0.2 seconds.
```

```r
draws_all <- fit_mcmc_all$draws(format = "draws_array", inc_warmup = TRUE)
mcmc_trace(draws_all, pars = c("alpha", "beta"), n_warmup = 1000)
```

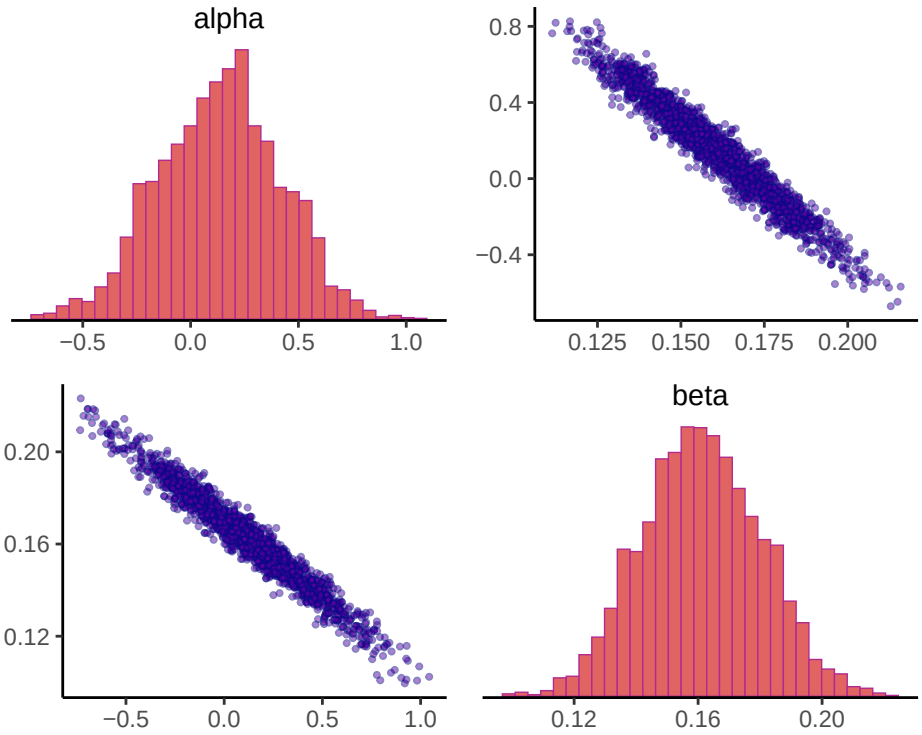


Gráficos de pares

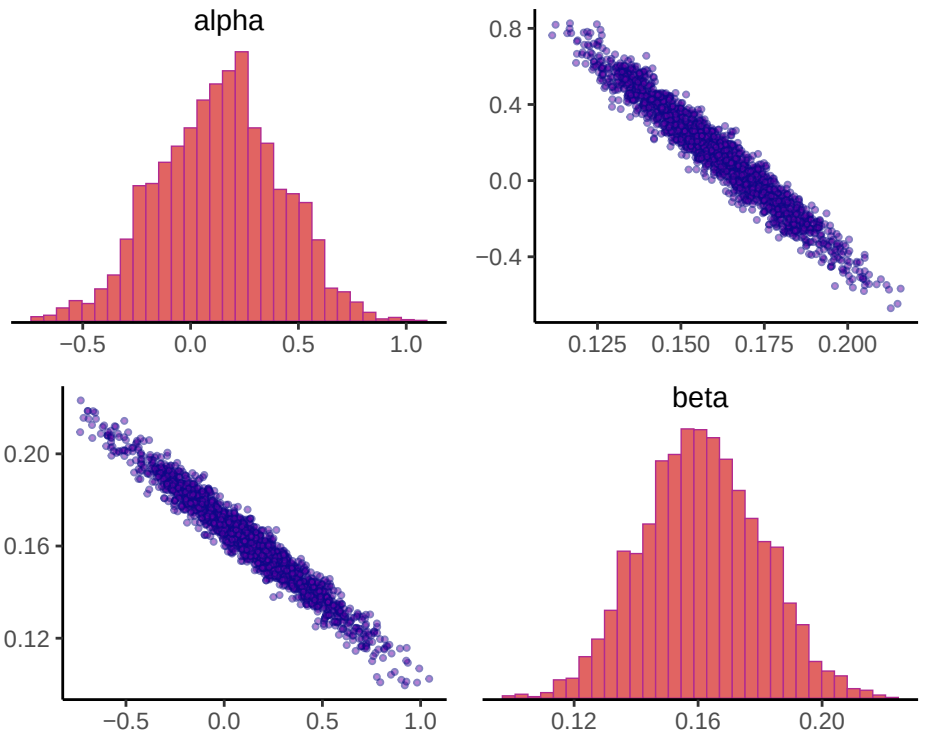
```
# Gráfico de dispersión, para ver un parámetro contra otro
mcmc_scatter(draws, pars = c("alpha", "beta"))
```




```
# Gráfico de pares, para ver de a muchos pares
mcmc_pairs(draws, pars = c("alpha", "beta"),
           off_diag_args = list(size = 0.75, alpha = 0.5))
```




```r
# Avanzado: evaluar dóndo hubo transiciones divergentes
# Extraemos los parámetros del No-U-Turn Sampler (NUTS)
np <- nuts_params(fit_mcmc)
mcmc_pairs(draws, np = np, pars = c("alpha", "beta"),
           off_diag_args = list(size = 0.75, alpha = 0.5))
```




```
# (como no hubo transiciones divergentes, no muestra puntos rojos)
```

Predicciones y enunciados probabilísticos

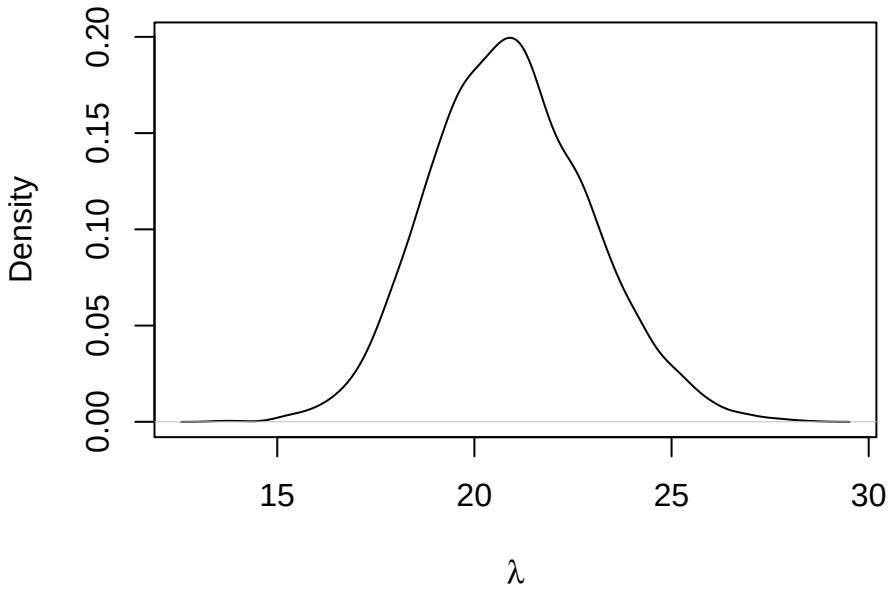
```
# Como vimos antes, podemos calcular probabilides como frecuencias.

# Extraemos las muestras como data.frame
d <- fit_mcmc$draws(variables = c("alpha", "beta"),
                    format = "draws_df")
nsim <- nrow(d)

# Pr(exp(beta) > 1.2)
ebeta <- exp(d$beta)
sum(ebeta > 1.2) / nsim
```

```
[1] 0.14675
```

```
# Pr(lambda > 20 | FWI = 18) // lambda es la media!
FWI <- 18
lambda <- exp(d$alpha + d$beta * FWI)
sum(lambda > 20) / nsim
```

```
[1] 0.66275
```

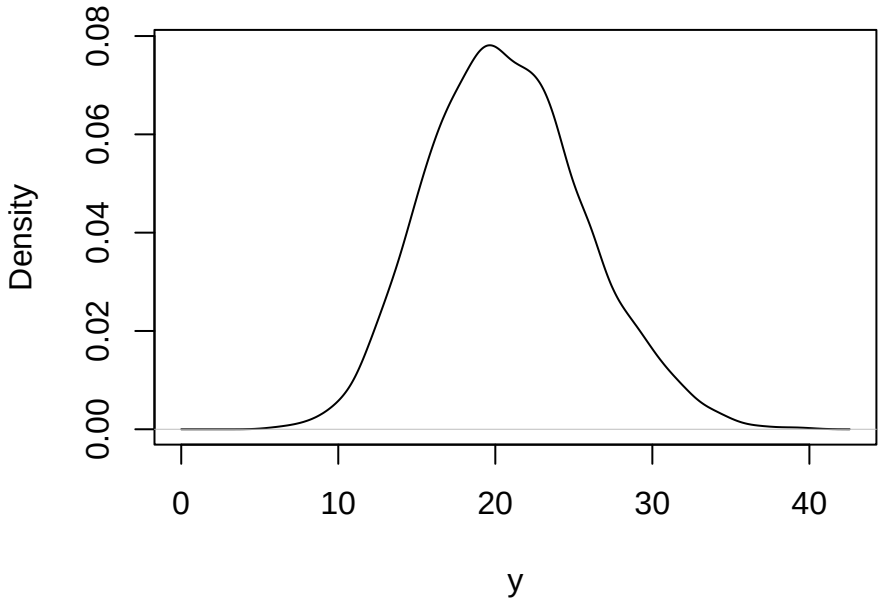
```r
# Distribución posterior de lambda (la media) cuando el FWI es 18:
plot(density(lambda), xlab = expression(lambda), main = NA)
```



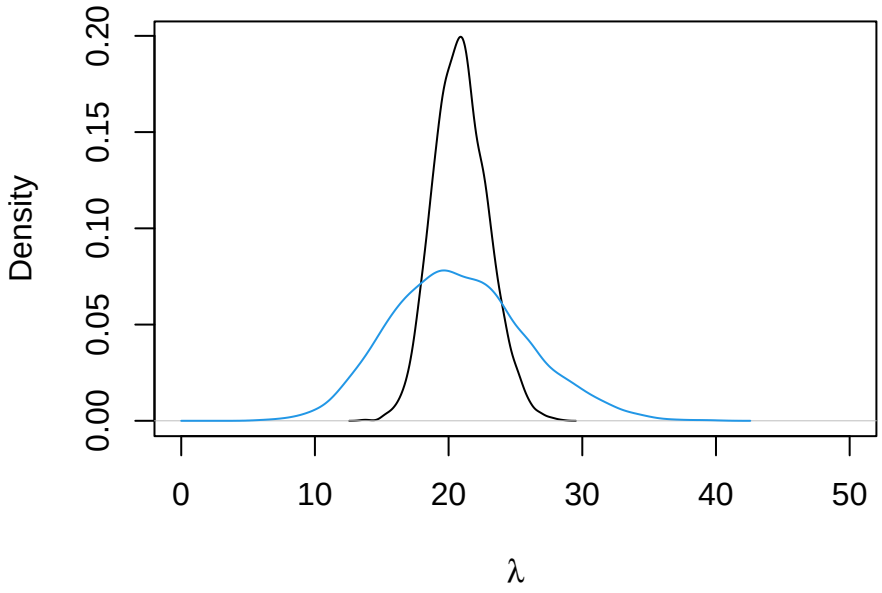

```r
# Pr(y > 20 | FWI = 18) //
ysim <- rpois(nsim, lambda) # simulamos datos :)
sum(ysim > 20) / nsim
```

```
[1] 0.499
```

```r
# Distribución predictiva posterior de y para FWI = 18:
plot(density(ysim, from = 0), xlab = "y", main = NA)
```

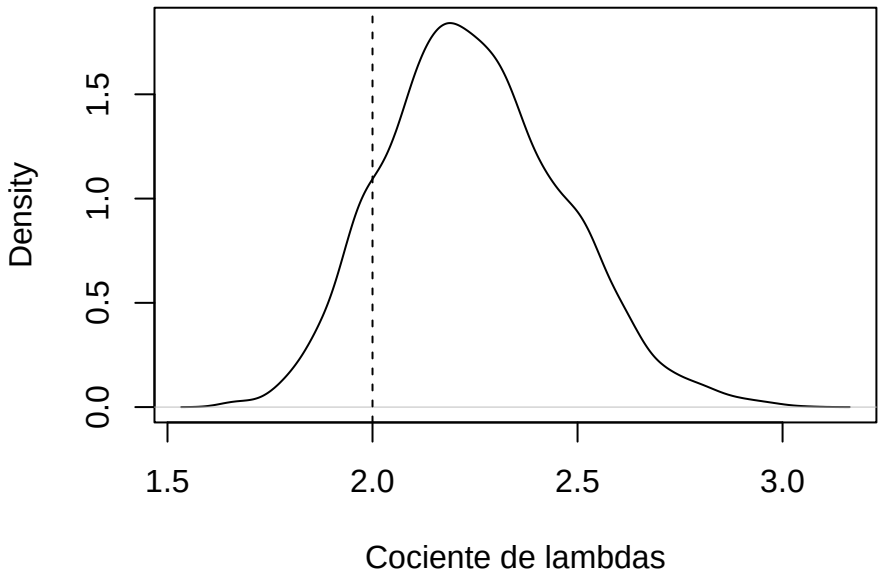



```
# Visualizamos ambos a la vez:
plot(density(lambda), xlab = expression(lambda, y), main = NA, xlim = c(0, 50))
lines(density(ysim, from = 0), xlab = "y", main = NA, col = 4)
```




```
# Y cuentas más arbitrarias aún. Ejemplo: ¿cuán probable es que el número de
# incendios medio con un FWI = 20 sea el doble o mayor que cuando el FWI es 15?
# Esto requiere calcular la distribución posterior de lambda para FWI = 15 y
# FWI = 20, luego calcular el cociente y evaluar qué proporción de muestras
```

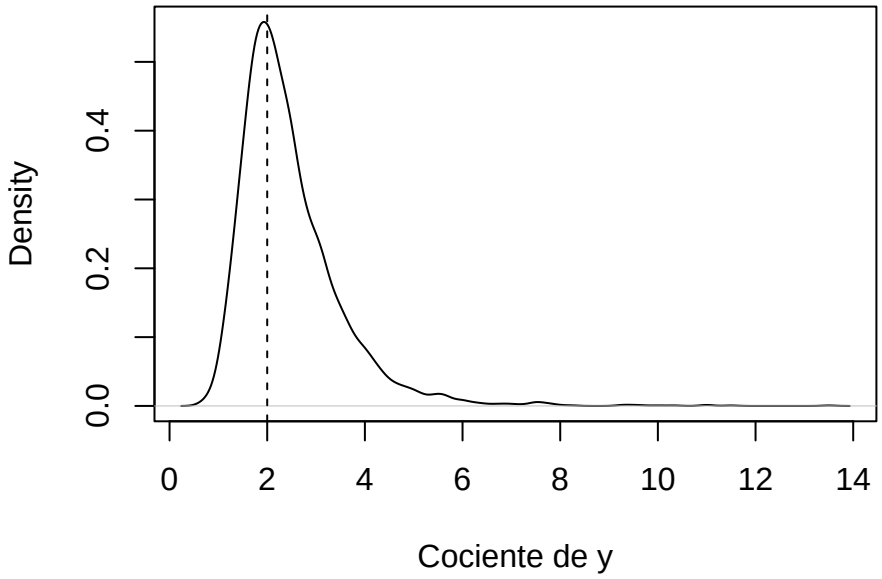
```r
# tienen un cociente >= 2.
lambda_15 <- exp(d$alpha + d$beta * 15)
lambda_20 <- exp(d$alpha + d$beta * 20)
qlambdas <- lambda_20 / lambda_15
plot(density(qlambdas), main = NA, xlab = "Cociente de lambdas")
abline(v = 2, lty = 2)
```




```r
# Pr(qlambdas >= 2)
sum(qlambdas >= 2) / nsim
```

```
[1] 0.871
```

```r
# También podemos reponder lo mismo sobre el número observado de incendios crudo,
# no su media:
y_15 <- rpois(nsim, lambda_15)
y_20 <- rpois(nsim, lambda_20)
qy <- y_20 / y_15
plot(density(qy), main = NA, xlab = "Cociente de y")
abline(v = 2, lty = 2)
```




```
# Pr(qy >= 2)
sum(qy >= 2) / nsim
```

```
[1] 0.65175
```

0.6 Trabajo Práctico 6