Tarea3-GLM Sofia Gerard y Román Vélez

```
library(tidyverse)
```

```
── Attaching core tidyverse packages ──────────────────────── tidyverse 2.0.0 ──
✔ dplyr     1.1.2     ✔ readr     2.1.4
✔ forcats   1.0.0     ✔ stringr   1.5.0
✔ ggplot2   3.4.4     ✔ tibble    3.2.1
✔ lubridate 1.9.2     ✔ tidyr     1.3.0
✔ purrr     1.0.2
── Conflicts ────────────────────────────────────────── tidyverse_conflicts() ──
✖ dplyr::filter() masks stats::filter()
✖ dplyr::lag()    masks stats::lag()
ℹ Use the conflicted package (<http://conflicted.r-lib.org/>) to force all conflicts to
become errors
```

```
library(ggplot2)
library(purrr)
library(dplyr)
library(knitr)
```

Ejercicio 1

```
# Calcula el valor exacto de la integral
valor_exacto <- integrate(sin, lower = 0, upper = pi/3)$value

# Estimación de Monte Carlo
# Tamaño máximo de la muestra
n <- 20000

# Función para integrar, en este caso sin(t)
h <- function(t) sin(t)

# Genera muestras en el intervalo [0, pi/3]
muestras <- runif(n, 0, pi/3)

# Evalúa la función seno en los puntos muestreados
evaluaciones <- h(muestras)

# Estima la integral como el promedio de las evaluaciones muestreadas
# multiplicado por el rango de integración
estimacion_monte_carlo <- (pi/3) * mean(evaluaciones)

# Imprime los resultados
cat("Valor exacto de la integral:", valor_exacto, "\n")
```

```
Valor exacto de la integral: 0.5
```

```r
cat("Estimación de Monte Carlo con", n, "muestras:", estimacion_monte_carlo)
```

```
Estimación de Monte Carlo con 20000 muestras: 0.4965773
```

Ejercicio 2

```r
# Función para calcular el estimador de Monte Carlo de la CDF de una Beta(3, 3)
estimar_cdf_beta_monte_carlo <- function(x, n = 20000) {
  # Genera muestras de una Beta(3, 3)
  muestras_beta <- rbeta(n, 3, 3)

  # Inicializar un vector para almacenar las estimaciones de la CDF para cada x
  estimaciones_cdf <- numeric(length(x))

  # Calcular las estimaciones de la CDF para cada valor de x
  for(i in 1:length(x)) {
    estimaciones_cdf[i] <- mean(muestras_beta <= x[i])
  }

  return(estimaciones_cdf)
  }

# Usar la función para estimar la CDF para un rango de valores de x
x_valores <- seq(0.1, 0.9, by = 0.1)
estimaciones <- estimar_cdf_beta_monte_carlo(x_valores)

# Imprimir las estimaciones
cat("Estimaciones de Monte Carlo de la CDF para valores de x:", "\n")
```

```
Estimaciones de Monte Carlo de la CDF para valores de x:
```

```r
print(data.frame(x = x_valores, Estimaciones_CDF = estimaciones))
```

```
    x Estimaciones_CDF
1 0.1          0.00845
2 0.2          0.05685
3 0.3          0.16125
4 0.4          0.31315
5 0.5          0.49170
6 0.6          0.67555
7 0.7          0.83305
8 0.8          0.94005
9 0.9          0.99085
```

```r
# Calcular los valores exactos con pbeta
valores_exactos <- pbeta(x_valores, 3, 3)

# Resultados
resultados_df <- data.frame(
  x = x_valores,
  Estimaciones_Monte_Carlo = estimaciones,
  Valores_Exactos_CDF = valores_exactos
)

# Imprimir el DataFrame
print(resultados_df)
```

```
    x Estimaciones_Monte_Carlo Valores_Exactos_CDF
1 0.1                  0.00845             0.00856
2 0.2                  0.05685             0.05792
3 0.3                  0.16125             0.16308
4 0.4                  0.31315             0.31744
5 0.5                  0.49170             0.50000
6 0.6                  0.67555             0.68256
7 0.7                  0.83305             0.83692
8 0.8                  0.94005             0.94208
9 0.9                  0.99085             0.99144
```

Ejercicio 3:

```r
# Estimación de Monte Carlo
N <- 10000
err_estim_max <- 0.001

# Define la función a integrar
h <- function(x) {
  exp(-x) / (1 + x^2)
}

# Genera muestras aleatorias entre 0 y 1
x <- runif(N)

# Evalúa la función en las muestras aleatorias
evaluaciones <- h(x)

# Calcula la estimación de Monte Carlo como el promedio de las evaluaciones
estimacion_monte_carlo <- mean(evaluaciones)

# Calcula el error estándar de la estimación
sem <- sd(evaluaciones) / sqrt(N)

# Calcula el tamaño de muestra necesario para lograr el error de estimación máximo desead
```

```
# Usando la fórmula para el intervalo de confianza de 95%
Z <- qnorm(0.975) # IC 95%
n_required <- ceiling((Z * sem / err_estim_max)^2)

# Imprime los resultados
cat("Estimación de Monte Carlo:", estimacion_monte_carlo, "\n")
```

```
Estimación de Monte Carlo: 0.5235905
```

```
cat("Error estándar de la estimación:", sem, "\n")
```

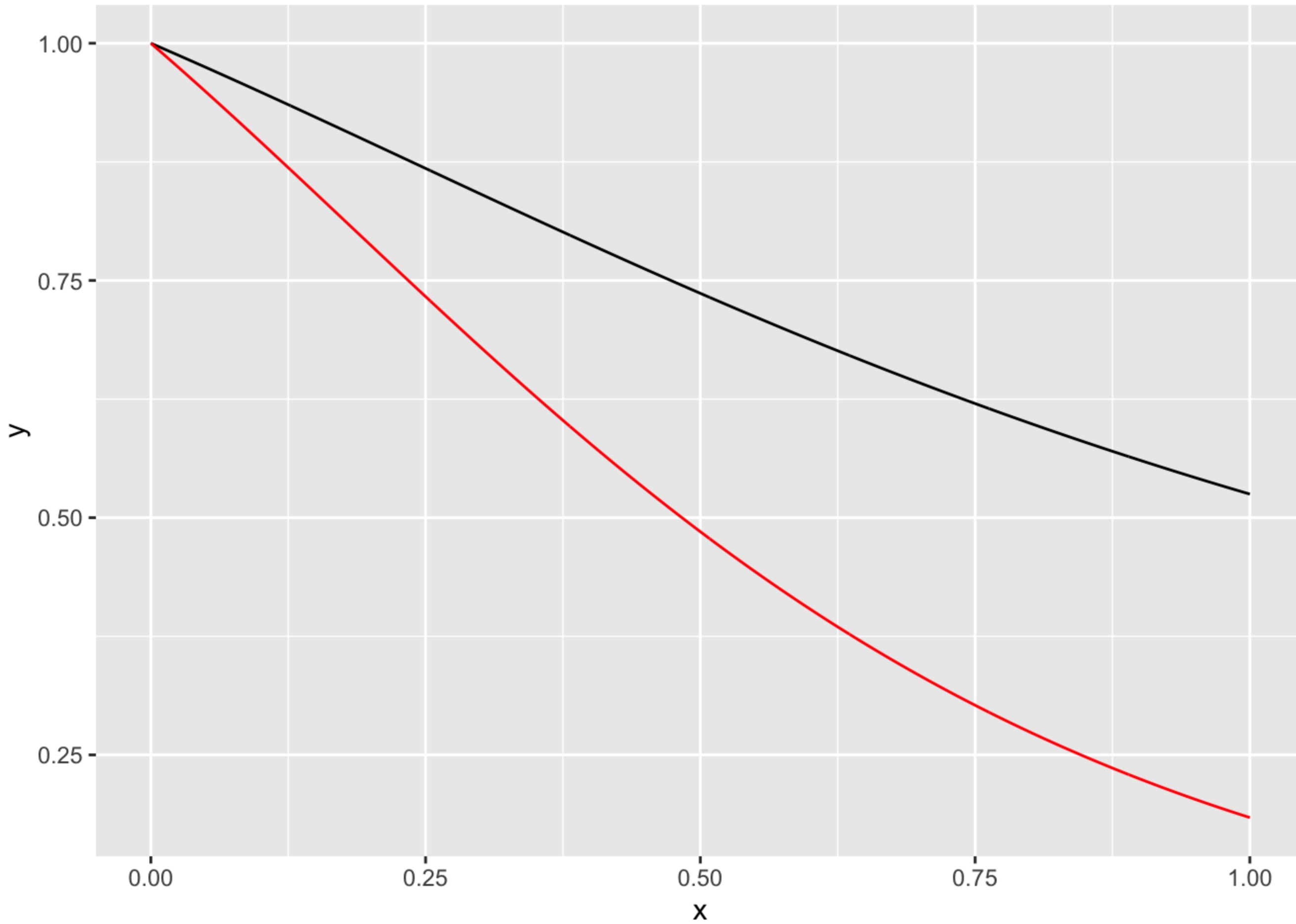
```
Error estándar de la estimación: 0.002442705
```

```
cat("Tamaño de la muestra necesario para un error máximo de ±0.001:", n_required, "\n")
```

```
Tamaño de la muestra necesario para un error máximo de ±0.001: 23
```

Ejercicio 3: solución alternativa

```
N <- 1000
err_estim_max <- 0.001
h <- function(x){
  return(exp(-x)/(1+x^2))
}
x <- seq(0,1, length.out = N)
Int <- cumsum(h(x))/(1:N)
var_n <- sqrt((x-Int)^2)/(1:N)

# find the first value that fulfils err_estim_max
min_num_samples <- which(var_n < err_estim_max)[1]
plt_df_p3 <- data.frame(x = x,
                        y = Int,
                        fcn=h(x))
ggplot(plt_df_p3, aes(x=x, y=y)) +
  geom_line() +
  geom_line(aes(x=x, y=fcn), color="red")
```




```
print("tamaño de muestra necesario:")
```

```
[1] "tamaño de muestra necesario:"
```

```
print(min_num_samples)
```

```
[1] 396
```

Ejercicio 4

```
# Importance Sampling: en lugar de muestrear de manera uniforme de toda la distribución d

##### Escrito a mano
```

Ejercicio 5

```
# Funciones de importancia f1 y f2
 # f1 = exponencial más sencilla
 # f2 = cauchy porque tiene colas más pesadas
```

```r
# Función objetivo g(x)
g <- function(x) {
  x^2 / sqrt(2 * pi) * exp(-x^2 / 2)
}

# Definir las funciones de importancia f1 y f2
f1 <- function(x) {
  exp(-x)
}

f2 <- function(x) {
  1 / (pi * (1 + x^2))
}

n_samples <- 100000

# Monte Carlo
monte_carlo <- function(n, f_importance, distribution_sampler) {
  # Generar n muestras de la distribución de importancia
  samples <- distribution_sampler(n)

 # El soporte es x > 1
  valid_samples <- samples[samples > 1]

  # Calcular los pesos para cada muestra válida.
  weights <- g(valid_samples) / f_importance(valid_samples)

  # Calcular la media de los pesos (estimación de la integral)
  estimated_integral <- mean(weights)

  # Calcular la varianza de los pesos
  variance_of_weights <- var(weights)

  # Lista con la estimación y la varianza
  list(estimated_integral = estimated_integral, variance = variance_of_weights)
}

set.seed(1994)

# Simular y calcular la varianza para f1
results_f1 <- monte_carlo(n_samples, f1, function(n) rexp(n, rate=1))

# Simular y calcular la varianza para f2
results_f2 <- monte_carlo(n_samples, f2, function(n) rcauchy(n, location=0, scale=1))

# Mostrar los resultados
print(results_f1)
```

```
$estimated_integral
[1] 1.08806

$variance
[1] 0.1818549
```

```
print(results_f2)
```

```
$estimated_integral
[1] 1.59751

$variance
[1] 1.750909
```

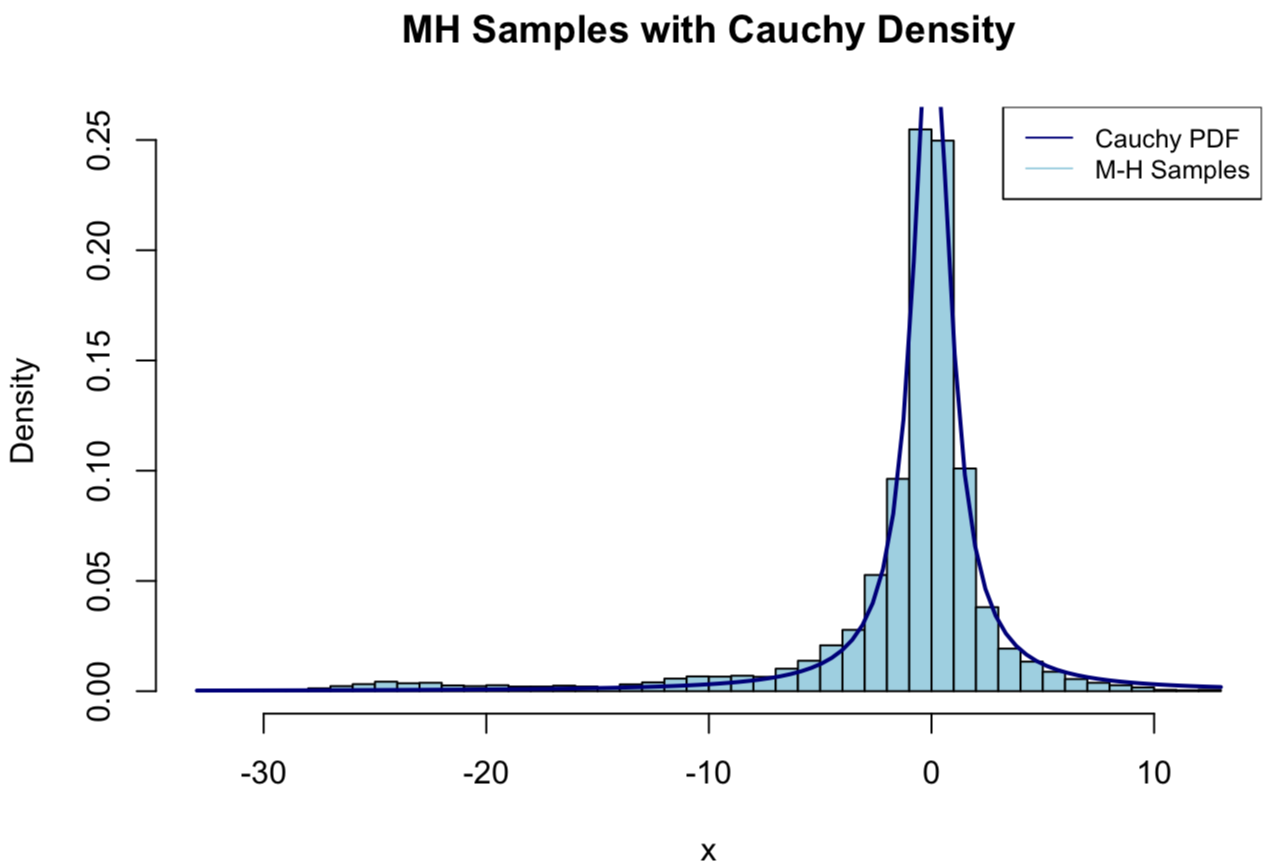
Ejercicio 6

```
# Usar el algoritmo de Metropolis-Hastings para generar variables aleatorias de una densi

# Definir la función de densidad de probabilidad (PDF) de Cauchy estándar
cauchy_density <- function(x) {
  1 / (pi * (1 + x^2))
}

# Configurar el algoritmo
n_iterations <- 11000  # Total de iteraciones incluyendo burn-in
x <- numeric(n_iterations)  # Vector para almacenar las muestras
x[1] <- 0  # Valor inicial arbitrario

# Implementar el algoritmo de Metropolis-Hastings
set.seed(1994)  # Establecer una semilla para reproducibilidad
for (i in 2:n_iterations) {
  current_x <- x[i - 1]  # El valor actual de la cadena
  proposed_x <- rnorm(1, mean = current_x, sd = 1)  # Generar un candidato desde una norm

  # Calcular la razón de aceptación
  acceptance_ratio <- cauchy_density(proposed_x) / cauchy_density(current_x)

  # Decidir si se acepta el candidato
  if (runif(1) < acceptance_ratio) {
    x[i] <- proposed_x  # Aceptar el candidato
  } else {
    x[i] <- current_x  # Rechazar el candidato y mantener el actual
  }
}

# Descartar los primeros 1000 valores (burn-in)
x <- x[-(1:1000)]  # Eliminar el burn-in para quedarnos solo con las muestras válidas
```

```r
# Calcular los deciles de la muestra
sample_deciles <- quantile(x, probs = seq(0.1, 0.9, by = 0.1))

# Calcular los deciles teóricos de la distribución de Cauchy estándar
theoretical_deciles <- qcauchy(seq(0.1, 0.9, by = 0.1), location = 0, scale = 1)

# Comparar los deciles de la muestra con los deciles teóricos
deciles_comparison <- data.frame(
  Sample = sample_deciles,
  Theoretical = theoretical_deciles
)
print(deciles_comparison)
```

```
        Sample Theoretical
10% -5.1207899  -3.0776835
20% -2.0363967  -1.3763819
30% -0.9946687  -0.7265425
40% -0.5157644  -0.3249197
50% -0.1629317   0.0000000
60%  0.1518595   0.3249197
70%  0.4628965   0.7265425
80%  0.9838074   1.3763819
90%  1.8940462   3.0776835
```

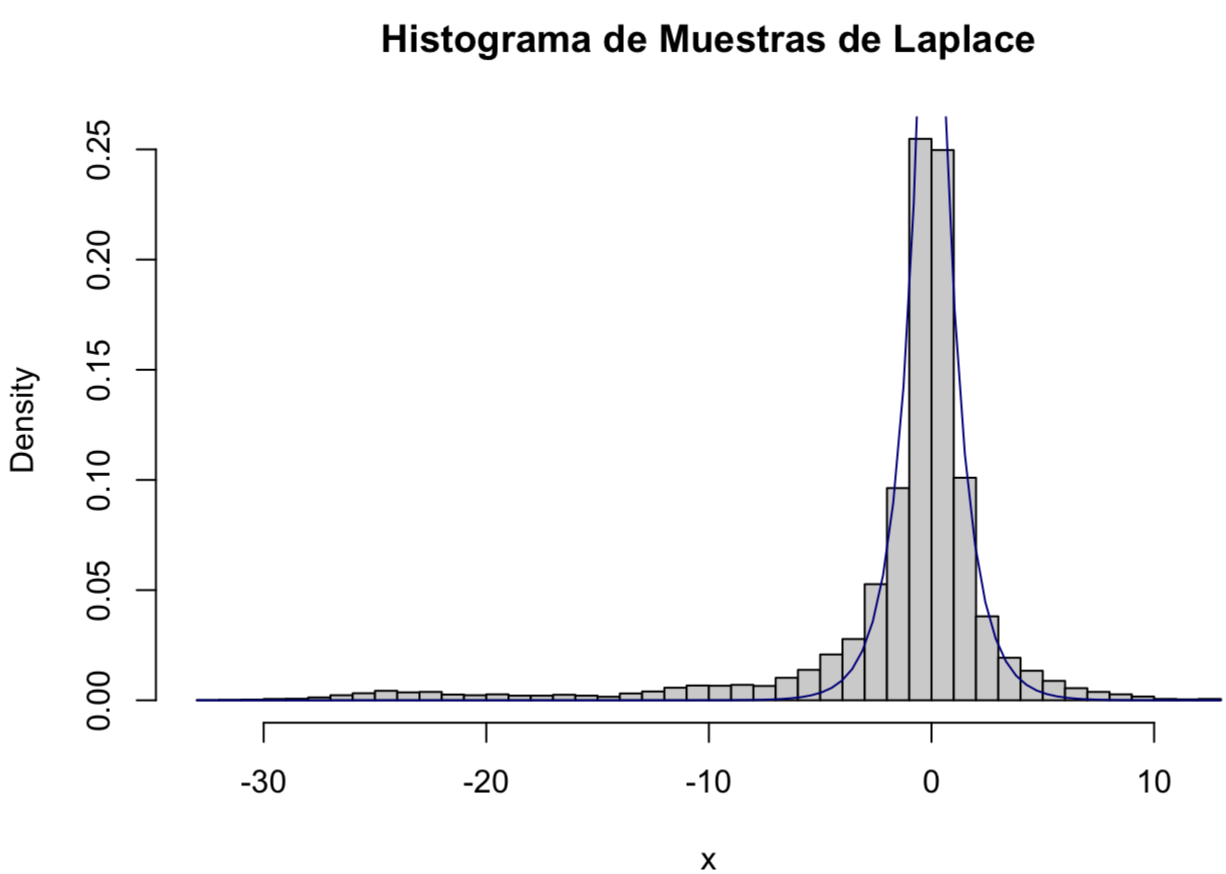
```r
# Crear un histograma para comparar
hist(x, breaks = 50, probability = TRUE, col = "lightblue", main = "MH Samples with Cauch
curve(dcauchy(x), col = "darkblue", add = TRUE, lwd = 2)
legend("topright", legend = c("Cauchy PDF", "M-H Samples"),
       col = c("darkblue", "lightblue"), lty = 1:1, cex = 0.8)
```



Ejercicio 7

```
# Definir la densidad de la distribución de Laplace
laplace_density <- function(x) {
  (1/2) * exp(-abs(x))
}

# Función para el muestreo de Metropolis-Hastings
metropolis_sampling <- function(varianza, n_iterations, init_value = 0) {
  # Inicializar el vector para almacenar las muestras
  x <- numeric(n_iterations)
  x[1] <- init_value  # Valor inicial de la cadena
  n_accepted <- 0  # Contador de aceptaciones


  for (i in 2:n_iterations) {
    current_x <- x[i - 1]
    proposed_x <- rnorm(1, mean = current_x, sd = sqrt(varianza))

    # Calcular la razón de aceptación
    acceptance_ratio <- laplace_density(proposed_x) / laplace_density(current_x)
```

```
    # AR
    if (runif(1) < acceptance_ratio) {
      x[i] <- proposed_x  # Aceptar el candidato
      n_accepted <- n_accepted + 1
    } else {
      x[i] <- current_x  # Rechazar el candidato y mantener el actual
    }
  }

  # Calcular la tasa de aceptación
  acceptance_rate <- n_accepted / (n_iterations - 1)

  return(list(samples = x, acceptance_rate = acceptance_rate))
}

# Histograma de las muestras
hist(x, probability = TRUE, breaks = 40, main = "Histograma de Muestras de Laplace")
curve(laplace_density, col = "darkblue", add = TRUE)
```




```
set.seed(1994)

# Número de iteraciones
n_iterations <- 10000
```

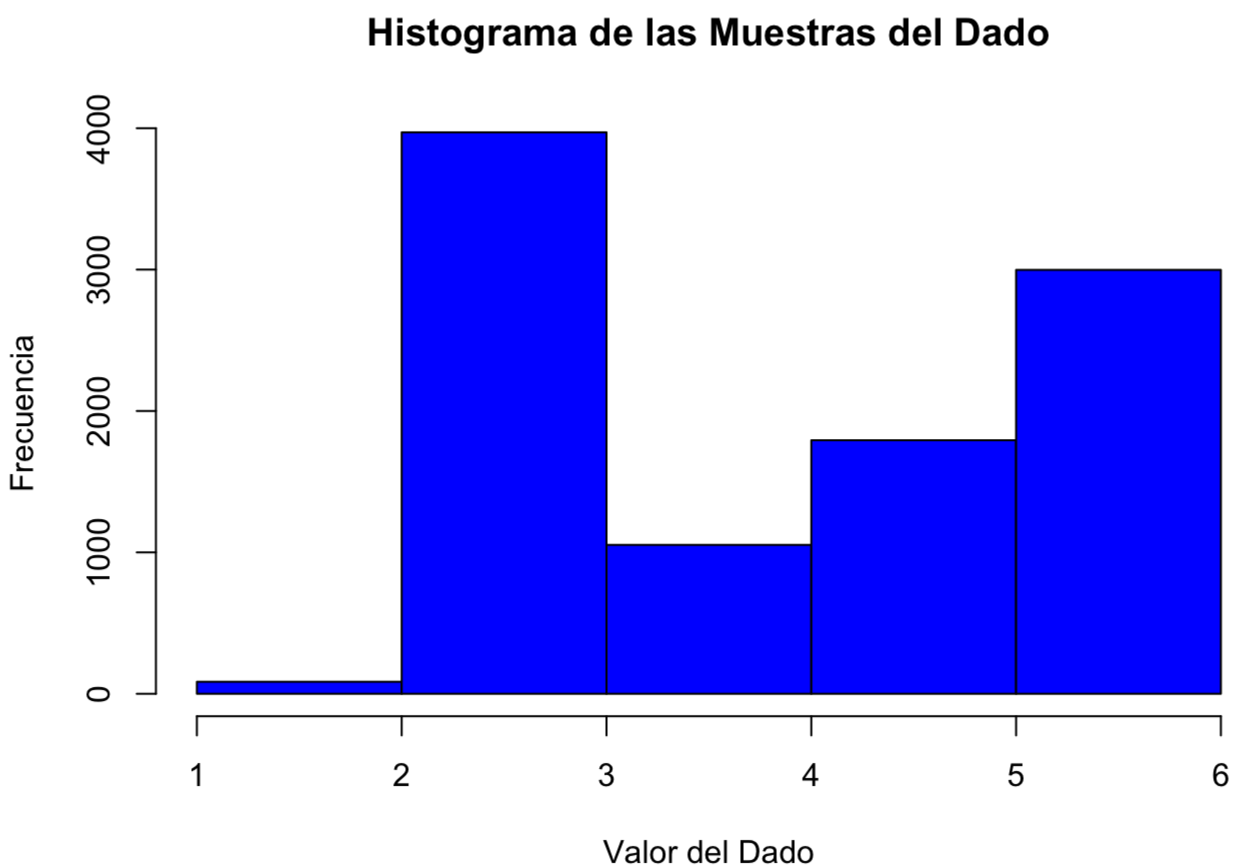
```
# Definir las varianzas y realizar el muestreo
variances <- c(0.1, 1, 10)
results <- lapply(variances, function(v) metropolis_sampling(v, n_iterations))

# Extraer y mostrar las tasas de aceptación
acceptance_rates <- sapply(results, function(res) res$acceptance_rate)
names(acceptance_rates) <- variances
results_table <- data.frame(Varianza = variances, Tasa_de_Aceptacion = acceptance_rates)
kable(results_table, row.names = FALSE)
```

Varianza	Tasa_de_Aceptacion
0.1	0.8874887
1.0	0.7067707
10.0	0.4011401

Ejercicio 8:

```
# distribución propuesta debe basarse en un dado honesto, es decir, cada número del 1 al

# Configurar el algoritmo
n_iterations <- 10000
x <- numeric(n_iterations)
x[1] <- sample(1:6, 1)

# Definir la distribución objetivo
probabilidades_objetivo <- c(0.01, 0.39, 0.11, 0.18, 0.26, 0.05)
dado_objetivo <- function(x) {
   return(probabilidades_objetivo[x])
}

# Metropolis-Hastings
set.seed(1994)
for (i in 2:n_iterations) {
  current_x <- x[i - 1]
  proposed_x <- sample(1:6, 1)

  # Calcular la razón de aceptación
  acceptance_ratio <- dado_objetivo(proposed_x) / dado_objetivo(current_x)

  # Decidir si se acepta el candidato
  if (runif(1) < acceptance_ratio) {
    x[i] <- proposed_x
  } else {
    x[i] <- current_x
  }
```

```
}
burn_in <- 100
muestras_efectivas <- x[(burn_in + 1):n_iterations]

hist(muestras_efectivas, breaks = 6, right = FALSE, main = "Histograma de las Muestras de
     xlab = "Valor del Dado", ylab = "Frecuencia", col = "blue")
```



Ejercicio 9

a. Mostrar que el número de sucesiones binarias de longitud m sin 1's adyacentes es f_{m+2}
   Demostración por inducción:Se cumple para m=1

$$f_{m+2} = f_{1+2} = f_3 = f_2 + f_1 = 1 + 1 = 2\{0, 1\}$$

   Suponer que se cumple para m-1

$$f_{m-1+2} = f_{m+1} = f_m + f_{m-1}$$

   Esto es las combinaciones de 0 y 1, de lomgitud $m - 1$ sin unos adyacentes es f_{m+1}Finalmente tenemos que demostrar que para m se cumple:

$$f_{m+2} = f_{m+1} + f_m$$

Como sabemos que se cumple para f_{m+1}, sabemos que de las las 2^{m-1} combinaciones del paso anterior, hubo $2^{m-1} - f_{m+1}$ que tuvieron 1's adyacentes y que no van a dar nuevas combinaciones en el siguiente digito, y sabemos que de las combinaciones aceptadas para $m-1$, f_{m-1} terminan en uno y en el siguiente dígito van a generar f_{m-1} combinaciones que no se acepten, esto es:

$$2^m = f_{m+2} + [2 * (2^{m-1} - f_{m+1}) + f_{m-1}]$$

$$2^m = f_{m+2} + [2^m - 2f_{m+1}] + f_{m-1}$$

$$2^m - 2^m = f_{m+2} - 2f_{m+1}] + f_{m-1}$$

$$f_{m+2} = f_{m+1} + f_{m+1} - f_{m-1}$$

$$f_{m+2} = f_{m+1} + f_m + f_{m-1} - f_{m-1}$$

$$f_{m+2} = f_{m+1} + f_m$$

b. Sea $p_{k,m}$ el número de sucesiones binarias de longitud m con exactamente k 1's. Mostrar que

$$p_{k,m} = \binom{m-k+1}{k}, k = 0, 1, \ldots, \text{ceiling}(m/2)$$

El problema es equivalente a demostrar:

$$\forall n \in \mathbb{Z} : f_n = \sum_{k=0}^{\lfloor \frac{n-1}{2} \rfloor} \binom{n-k-1}{k}$$

Demostración por inducción:Comprobamos que se cumple para $n = 1, 2$

$$f_1 = 1 = \binom{0}{0} = \binom{1-0-1}{0} = \sum_{k=0}^{\lfloor \frac{1-1}{2} \rfloor} \binom{1-k-1}{k}$$

$$f_2 = 1 + 0 = \binom{1}{0} = \binom{2-0-1}{0} = \sum_{k=0}^{\lfloor \frac{2-1}{2} \rfloor} \binom{1-k-1}{k}$$

si suponemos que n es par, tenemos que suponer que se cumple para $n-1$ y n:

$$f_{n-1} = \sum_{k=0}^{\frac{n}{2}-1} \binom{n-k-1}{k} f_n = \sum_{k=0}^{\frac{n}{2}-1} \binom{n-k-1}{k}$$

Y demostrar que se cumple para:

$$f_{n+1} = \sum_{k=0}^{\frac{n}{2}} \binom{n-k}{k} f_{n+2} = \sum_{k=0}^{\frac{n}{2}} \binom{n-k+1}{k}$$

Primero para impares tenemos:

$$\sum_{k=0}^{\frac{n}{2}} \binom{n-k}{k} = \binom{n}{0} + \sum_{k=1}^{\frac{n}{2}-1} \binom{n-k}{k} + \binom{n-\frac{n}{2}}{\frac{n}{2}}$$

$$= 1 + \sum_{k=1}^{\frac{n}{2}-1} \binom{n-k}{k} + \binom{\frac{n}{2}}{\frac{n}{2}}$$

$$= 1 + \sum_{k=1}^{\frac{n}{2}-1} \binom{n-k}{k} + 1$$

$$= 1 + \sum_{k=1}^{\frac{n}{2}-1} \left(\binom{n-k-1}{k} + \binom{n-k-1}{k-1} \right) + 1$$

$$= 1 + \sum_{k=1}^{\frac{n}{2}-1} \binom{n-k-1}{k} + \sum_{k=1}^{\frac{n}{2}-1} \binom{n-k-1}{k-1} + 1$$

$$= 1 + \sum_{k=1}^{\frac{n}{2}-1} \binom{n-k-1}{k} + \sum_{k=0}^{\frac{n}{2}-2} \binom{n-k-2}{k-1} + 1$$

$$= \binom{n-2}{0} + \sum_{k=1}^{\frac{n}{2}-1} \binom{n-k-1}{k} + \sum_{k=0}^{\frac{n}{2}-2} \binom{n-k-2}{k-1} + \binom{n-(\frac{n}{2}-1)-2}{\frac{n}{2}-1}$$

$$= \sum_{k=0}^{\frac{n}{2}-1} \binom{n-k-1}{k} + \sum_{k=0}^{\frac{n}{2}-1} \binom{n-k-2}{k}$$

$$= f_n + f_{n-1} = f_{n+1}$$

Y para pares:

$$\sum_{k=0}^{\frac{n}{2}} \binom{n-k+1}{k} = \binom{n}{0} + \sum_{k=1}^{\frac{n}{2}} \binom{n-k+1}{k}$$

$$= 1 + \sum_{k=1}^{\frac{n}{2}} \binom{n-k+1}{k}$$

$$= 1 + \sum_{k=1}^{\frac{n}{2}} \left(\binom{n-k}{k} + \binom{n-k}{k-1} \right)$$

$$= 1 + \sum_{k=1}^{\frac{n}{2}} \binom{n-k}{k} + \sum_{k=1}^{\frac{n}{2}} \binom{n-k}{k-1}$$

$$= 1 + \sum_{k=1}^{\frac{n}{2}} \binom{n-k}{k} + \sum_{k=0}^{\frac{n}{2}-1} \binom{n-k-1}{k}$$

$$= \binom{n-2}{0} + \sum_{k=1}^{\frac{n}{2}} \binom{n-k}{k} + \sum_{k=0}^{\frac{n}{2}-1} \binom{n-k-1}{k}$$

$$= \sum_{k=0}^{\frac{n}{2}} \binom{n-k}{k} + \sum_{k=0}^{\frac{n}{2}-1} \binom{n-k-1}{k}$$

$$= f_{n+1} + f_n = f_{n+2}$$

c.

```
# Función para calcular el número esperado de 1's en sucesiones binarias de longitud m
calcularMuM <- function(m) {
  # Inicializamos los dos primeros términos de la secuencia de Fibonacci
  if (m == 1) return(1)
  fib <- numeric(m + 2)
  fib[1] <- 1
  fib[2] <- 1

  # Calculamos los términos de la secuencia de Fibonacci hasta m+2
  for (i in 3:(m + 2)) {
    fib[i] <- fib[i - 1] + fib[i - 2]
  }

  # Calculamos el número esperado de 1's usando la relación con la secuencia de Fibonacci
  mu_m <- fib[m + 1] / fib[m + 2] * m
  return(mu_m)
}

# Calculamos y mostramos el número esperado de 1's para m = 10, 100, y 1000
valores_m <- c(10, 100, 1000)
for (m in valores_m) {
  mu_m <- calcularMuM(m)
  cat(sprintf("Número esperado de 1's para m = %d: %f\n", m, mu_m))
}
```

```
Número esperado de 1's para m = 10: 6.180556
Número esperado de 1's para m = 100: 61.803399
Número esperado de 1's para m = 1000: 618.033989
```

Tarea 3

Sofia Gerard
Román Vélez

Ejercicio 4:

* $\hat{\theta}_{IS}$ estimador de importancia de $\theta = \int g(x)\,dx$

Probar que si $g(x)/f(x)$ es acotada, entonces la varianza del estimador de IS es finita.

$$Var(\hat{\theta}_{IS}) = Var\left(\frac{g(x)}{f(x)}\right) = E\left[\left(\frac{g(x)}{f(x)}\right)^2\right] - \left(E\left[\frac{g(x)}{f(x)}\right]\right)^2$$

Función acotada: $|f(x)| \leq M \longrightarrow |g(x)/f(x)| \leq M$

$$E\left[\left(\frac{g(x)}{f(x)}\right)^2\right] \leq E[M^2] = M^2 \int \underbrace{f(x)\,dx}_{1} = M^2$$

$\therefore$ Si $E\left[\left(\frac{g(x)}{f(x)}\right)^2\right]$ es acotado por M^2, $Var(\hat{\theta}_{IS})$ es finita, ya que la expectativa del cuadrado es finita y el cuadrado de la expectativa es también finito.