Simulación Estadística

Rubén Fernández Casal (ruben.fcasal@udc.es), Ricardo Cao (rcao@udc.es)

Edición: Agosto de 2022. Impresión: 2022-12-14

Índice general

Prólogo

Esta es la segunda edición del libro, **en proceso de elaboración** (se puede acceder a la primera edición a través de este enlace).

Este libro contiene los apuntes de la asignatura de Simulación Estadística del Máster en Técnicas Estadísticas y material de apoyo a la docencia de la asignatura de Técnicas de Simulación y Remuestreo del Grado en Ciencia e Ingeniería de Datos de la UDC.

Este libro ha sido escrito en R-Markdown empleando el paquete `bookdown` y está disponible en el repositorio Github: rubenfcasal/simbook2. Se puede acceder a la versión en línea a través del siguiente enlace:

https://rubenfcasal.github.io/simbook2/index.html.

donde puede descargarse en formato pdf.

Para poder ejecutar los ejemplos mostrados en el libro es recomendable emplear el paquete `simres`, ***no disponible actualmente en CRAN***, aunque se puede instalar la versión de desarrollo en GitHub):

```
# install.packages("remotes")
# remotes::install_github("rubenfcasal/simres")
remotes::install_github("rubenfcasal/simres", INSTALL_opts = "--with-keep.source")
```

Alternativamente se pueden emplear los ficheros de la carpeta *codigo*.

Para instalar los paquetes necesarios se puede emplear los siguientes comandos:

```
pkgs <- c('tictoc', 'boot', 'randtoolbox', 'MASS', 'DEoptim', 'nortest', 'geoR', 'copula',
          'sm', 'car', 'tseries', 'forecast', 'plot3D', 'rgl', 'rngWELL', 'randtoolbox')
install.packages(setdiff(pkgs, installed.packages()[,"Package"]),
                 dependencies = TRUE)

# Si aparecen errores debidos a incompatibilidades entre las versiones de los paquetes,
# probar a ejecutar en lugar de lo anterior:
# install.packages(pkgs, dependencies = TRUE) # Instala todos...
```

Para generar el libro (compilar) serán necesarios paquetes adicionales, para lo que se recomendaría consultar el libro de "Escritura de libros con bookdown" en castellano.

Capítulo 1

Introducción a la simulación

Cuando pensamos en ciencia pensamos en experimentos y en modelos. Se experimenta una y otra vez sobre el fenómeno real que se desea conocer mejor para, con la información así acumulada, construir un modelo teórico, que no es sino una representación simplificada (más o menos acertada) del fenómeno real. Como el modelo se formula en términos matemáticos, en general es susceptible de un estudio analítico del que poder sacar conclusiones.

La simulación ofrece una alternativa a esa última fase del proceso, y sustituye (en parte o completamente) el estudio analítico por más experimentación, pero esta vez sobre el propio modelo en lugar de sobre la realidad.

Así, se puede definir la *simulación* como una técnica que consiste en realizar experimentos sobre el modelo de un sistema (experimentos de muestreo si la simulación incorpora aleatoriedad), con el objetivo de recopilar información bajo determinadas condiciones.

1.1 Conceptos básicos

La experimentación directa sobre la realidad puede tener muchos inconvenientes, entre otros:

- Coste elevado: por ejemplo cuando las pruebas son destructivas o si es necesario esperar mucho tiempo para observar los resultados.
- Puede no ser ética: por ejemplo la experimentación sobre seres humanos o la dispersión de un contaminante.
- Puede resultar imposible: por ejemplo cuando se trata de un acontecimiento futuro o una alternativa en el pasado.

Además la realidad puede ser demasiado compleja como para ser estudiada directamente y resultar preferible trabajar con un modelo del sistema real. Un modelo no es más que un conjunto de variables junto con ecuaciones matemáticas que las relacionan y restricciones sobre dichas variables. Habría dos tipos de modelos:

- Modelos deterministas: en los que bajo las mismas condiciones (fijados los valores de las variables explicativas) se obtienen siempre los mismos resultados.
- Modelos estocásticos (con componente aleatoria): tienen en cuenta la incertidumbre asociada al modelo. Tradicionalmente se supone que esta incertidumbre es debida a que no se dispone de toda la información sobre las variables que influyen en el fenómeno en estudio (puede ser debida simplemente a que haya errores de medida), lo que se conoce como *aleatoriedad aparente*:

  > "Nothing in Nature is random… a thing appears random only through the incompleteness of our knowledge."
  >
  > — Spinoza, Baruch (Ethics, 1677)

aunque hoy en día gana peso la idea de la física cuántica de que en el fondo hay una *aleatoriedad intrínseca*[1].

La modelización es una etapa presente en la mayor parte de los trabajos de investigación, especialmente en las ciencias experimentales. El modelo debería considerar las variables más relevantes para explicar el fenómeno en estudio y las principales relaciones entre ellas. La inferencia estadística proporciona herramientas para estimar los parámetros y contrastar la validez de un modelo estocástico a partir de los datos observados.

La idea es emplear el modelo, asumiendo que es válido, para resolver el problema de interés. Si se puede obtener la solución de forma analítica, esta suele ser exacta (aunque en ocasiones solo se dispone de soluciones aproximadas, basadas en resultados asintóticos, o que dependen de suposiciones que pueden ser cuestionables) y a menudo la resolución también es rápida. Cuando la solución no se puede obtener de modo analítico (o si la aproximación disponible no es adecuada) se puede recurrir a la simulación. De esta forma se pueden obtener resultados para un conjunto más amplio de modelos, que pueden ser mucho más complejos.

Nos centraremos en el caso de la *simulación estocástica*: las conclusiones se obtienen generando repetidamente simulaciones del modelo aleatorio. Muchas veces se emplea la denominación de *método Monte Carlo*[2] como sinónimo de simulación estocástica, pero realmente se trata de métodos especializados que emplean simulación para resolver problemas que pueden no estar relacionados con un modelo estocástico de un sistema real. Por ejemplo, en el Capítulo 7 se tratarán métodos de integración y optimización Monte Carlo.

1.1.1 Ejemplo

Supongamos que nos regalan un álbum con $n = 75$ cromos, que se venden sobres con $m = 6$ cromos por 0.8€, y que estamos interesados en el número de sobres que hay que comprar para completar la colección, por ejemplo en su valor medio.

Podemos aproximar la distribución del número de sobres para completar la colección a partir de $nsim = 1000$ simulaciones de coleccionistas de cromos:

```
# Parámetros
n <- 75 # Número total de cromos
m <- 6  # Número de cromos en cada sobre
repe <- TRUE # Repetición de cromos en cada sobre
# Número de simulaciones
nsim <- 1000
# Resultados simulación
nsobres <- numeric(nsim)
# evol <- vector("list", nsim)
# Fijar semilla
set.seed(1)
# Bucle simulación
for (isim in 1:nsim) {
  # seed <- .Random.seed    # .Random.seed <- seed
  album <- logical(n)
  i <- 0 # Número de sobres
  while(sum(album) < n) {
    i <- i + 1
```

[1] Como ejemplo, en física cuántica, la ecuación de Schrödinger es un modelo determinista que describe la evolución en el tiempo de la función de onda de un sistema. Sin embargo, como las funciones de onda pueden cambiar de forma aleatoria al realizar una medición, se emplea la regla de Born para modelar las probabilidades de las distintas posibilidades (algo que inicialmente generó rechazo, dió lugar a la famosa frase de Einstein "Dios no juega a los dados", pero experimentos posteriores parecen confirmar). Por tanto en la práctica se emplea un modelo estocástico.

[2] Estos métodos surgieron a finales de la década de 1940 como resultado del trabajo realizado por Stanislaw Ulam y John von Neumann en el proyecto Manhattan para el desarrollo de la bomba atómica. Al parecer, como se trataba de una investigación secreta, Nicholas Metropolis sugirió emplear el nombre clave de "Monte-Carlo" en referencia al casino de Monte Carlo de Mónaco.

```
    album[sample(n,m, replace = repe)] <- TRUE
  }
  nsobres[isim] <- i
}
```

Distribución del número de sobres para completar la colección (aproximada por simulación):

```
hist(nsobres, breaks = "FD", freq = FALSE,
     main = "", xlab = "Número de sobres")
lines(density(nsobres))
```

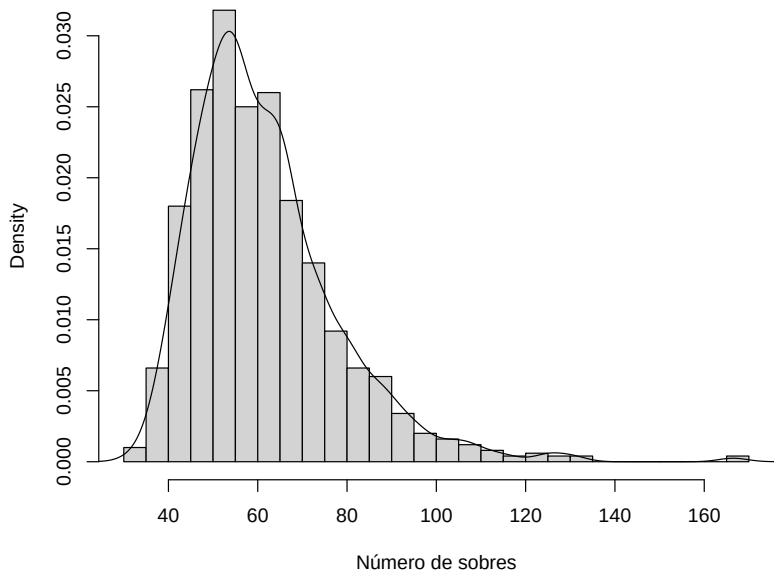


Figura 1.1: Aproximación por simulación de la distribución del número de sobres para completar la colección.

Aproximación por simulación del número medio de sobres para completar la colección:

```
sol <- mean(nsobres)
sol
```

```
## [1] 61.775
```

Número mínimo de sobres para asegurar de que se completa la colección con una probabilidad del 95%:

```
nmin <- quantile(nsobres, probs = 0.95)
ceiling(nmin)
```

```
## 95%
##  92
```

```
# Reserva de dinero para poder completar la colección el 95% de las veces:
ceiling(nmin)*0.8
```

```
##  95%
## 73.6
```

```
hist(nsobres, breaks = "FD", freq = FALSE,
     main = "", xlab = "Número de sobres")
lines(density(nsobres))
abline(v = sol)
abline(v = nmin, lty = 2)
```

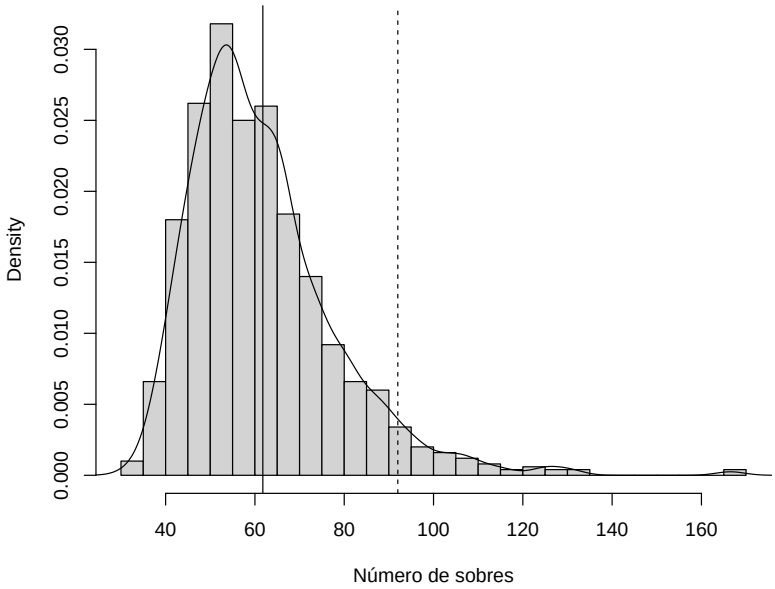


Figura 1.2: Aproximaciones por simulación de la distribución del número de sobres para completar la colección, de su valor esperado (línea vertical continua) y del cuantil 0.95 (línea vertical discontinua).

Por supuesto, la distribución del gasto necesario para completar la colección es esta misma reescalada.

```
res <- simres::mc.plot(nsobres*0.8)
```

Aproximación del gasto medio:

```
res$approx  # sol*0.8
```

```
## [1] 49.42
```

En el Ejercicio 1.5 se propone modificar este código para obtener información adicional sobre la evolución del número de cromos distintos dependiendo de los sobres comprados por un coleccionista.

1.1.2 Ventajas e inconvenientes de la simulación

Ventajas (Shannon, 1975):

- Cuando la resolución analítica no puede llevarse a cabo.
- Cuando existen medios de resolver analíticamente el problema pero dicha resolución es complicada y costosa (o solo proporciona una solución aproximada).
- Si se desea experimentar antes de que exista el sistema (pruebas para la construcción de un sistema).
- Cuando es imposible experimentar sobre el sistema real por ser dicha experimentación destructiva.

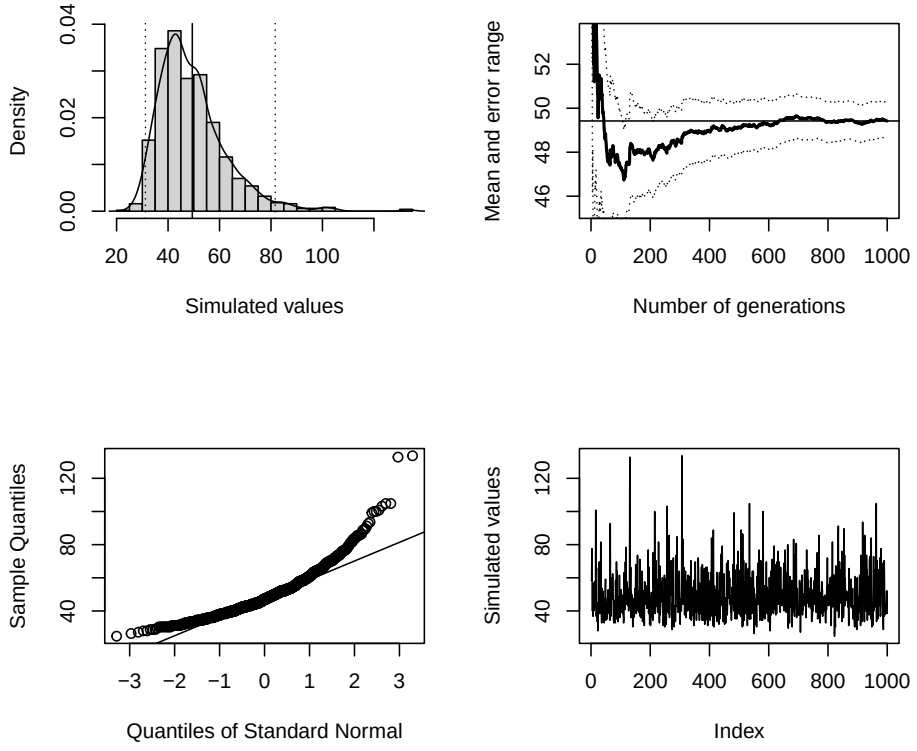


Figura 1.3: Gráficos exploratorios de las simulaciones del gasto para completar la colección obtenidos con la función ‘simres::mc.plot()‘.

- En ocasiones en las que la experimentación sobre el sistema es posible pero no ética.
- En sistemas que evolucionan muy lentamente en el tiempo.

El principal incoveniente puede ser el tiempo de computación necesario, aunque gracias a la gran potencia de cálculo de los computadores actuales, se puede obtener rápidamente una solución aproximada en la mayor parte de los problemas susceptibles de ser modelizados. Además siempre están presentes los posibles problemas debidos a emplear un modelo:

- La construcción de un buen modelo puede ser una tarea muy costosa (compleja, laboriosa y requerir mucho tiempo; e.g. modelos climáticos).
- Frecuentemente el modelo omite variables o relaciones importantes entre ellas (los resultados pueden no ser válidos para el sistema real).
- Resulta difícil conocer la precisión del modelo formulado.

Otro problema de la simulación es que se obtienen resultados para unos valores concretos de los parámetros del modelo, por lo que en principio resultaría complicado extrapolar las conclusiones a otras situaciones.

1.1.3 Aplicaciones de la simulación

La simulación resulta de utilidad en multitud de contextos diferentes. Los principales campos de aplicación son:

- Estadística:

  - Muestreo, remuestreo…
  - Aproximación de distribuciones (de estadísticos, estimadores…)
  - Realización de contrastes, intervalos de confianza…
  - Comparación de estimadores, contrastes…
  - Validación teoría (distribución asintótica…)
  - Inferencia Bayesiana
- Optimización: Algoritmos genéticos, temple simulado…
- Análisis numérico: Aproximación de integrales, resolución de ecuaciones…
- Computación: Diseño, verificación y validación de algoritmos…
- Criptografía: Protocolos de comunicación segura…
- Física: Simulación de fenómenos naturales…

En los capítulos 7.4 y 7 nos centraremos en algunas de las aplicaciones de utilidad en Estadística.

1.2 Tipos de números aleatorios

El primer requisito para poder realizar simulación estocástica sería disponer de números aleatorios. Se distingue entre tres tipos de secuencias:

- *números aleatorios puros* (*true random*): se caracteriza porque no existe ninguna regla o plan que nos permita conocer sus valores.
- *números pseudo-aleatorios*: simulan realizaciones de una variable aleatoria (uniforme),
- *números cuasi-aleatorios*: secuencias deterministas con una distribución más regular en el rango considerado.

1.2.1 Números aleatorios puros

Normalmente son obtenidos por procesos físicos (loterías, ruletas, ruidos…) y, hasta hace una décadas, se almacenaban en *tablas de dígitos aleatorios*. Por ejemplo, en 1955 la Corporación RAND publicó el libro *A Million Random Digits with 100,000 Normal Deviates* que contenía números aleatorios generados mediante una ruleta electrónica conectada a una computadora (ver Figura 1.4).

73735	45963	78134	63873
02965	58303	90708	20025
98859	23851	27965	62394
33666	62570	64775	78428
81666	26440	20422	05720
15838	47174	76866	14330
89793	34378	08730	56522
78155	22466	81978	57323
16381	66207	11698	99314
75002	80827	53867	37797
99982	27601	62686	44711
84543	87442	50033	14021
77757	54043	46176	42391
80871	32792	87989	72248
30500	28220	12444	71840

Figura 1.4: Líneas 10580-10594, columnas 21-40, del libro *A Million Random Digits with 100,000 Normal Deviates*.

El procedimiento que se utilizaba para seleccionar de una tabla, de forma manual, números aleatorios en un rango de 1 a m era el siguiente:

- Se selecciona al azar un punto de inicio en la tabla y la dirección que se seguirá.
- Se agrupan los dígitos de forma que "cubran" el valor de m.
- Se va avanzado en la dirección elegida, seleccionando los valores menores o iguales que m y descartando el resto.

Hoy en día están disponibles generadores de números aleatorios "online", por ejemplo:

- RANDOM.ORG: ruido atmosférico (ver paquete `random` en R).
- HotBits: desintegración radiactiva.

Aunque para un uso profesional es recomendable emplear generadores implementados mediante hardware:

- Intel Digital Random Number Generator
- An Overview of Hardware based True Random Number Generators

Sus principales aplicaciones hoy en día son en criptografía y juegos de azar, donde resulta especialmente importante su impredecibilidad.

El uso de números aleatorios puros presenta dos grandes incovenientes. El principal para su aplicación en el campo de la Estadística (y en otros casos) es que los valores generados deberían ser independientes e idénticamente distribuidos con distribución conocida, algo que resulta difícil (o imposible) de garantizar. Siempre está presente la posible aparición de sesgos, principalmente debidos a fallos del sistema o interferencias. Por ejemplo, en el caso de la máquina RAND, fallos mecánicos en el sistema de grabación de los datos causaron problemas de aleatoriedad (Hacking, 1965, p. 129).

El otro inconveniente estaría relacionado con su reproducibilidad, por lo que habría que almacenarlos en tablas si se quieren volver a reproducir los resultados. A partir de la década de 1960, al disponer de computadoras de mayor velocidad, empezó a resultar más eficiente generar valores mediante software en lugar de leerlos de tablas.

1.2.2 Números cuasi-aleatorios

Algunos problemas, como la integración numérica (en el Capítulo 7 se tratarán métodos de integración Monte Carlo), no dependen realmente de la aleatoriedad de la secuencia. Para evitar generaciones poco probables, se puede recurrir a secuencias cuasi-aleatorias, también denominadas *sucesiones de baja discrepancia* (hablaríamos entonces de métodos cuasi-Monte Carlo). La idea sería que la proporción de valores en una región cualquiera sea siempre aproximadamente proporcional a la medida de la región (como sucedería en media con la distribución uniforme, aunque no necesariamente para una realización concreta).

Por ejemplo, el paquete `randtoolbox` de R implementa métodos para la generación de secuencias cuasi-aleatorias (ver Figura 1.5).

```
library(randtoolbox)
n <- 2000
par.old <- par( mfrow=c(1,3))
plot(halton(n, dim = 2), xlab = 'x1', ylab = 'x2')
plot(sobol(n, dim = 2), xlab = 'x1', ylab = 'x2')
plot(torus(n, dim = 2), xlab = 'x1', ylab = 'x2')
```

```
par(par.old)
```

En este libro sólo consideraremos los números pseudoaleatorios y por comodidad se eliminará el prefijo "pseudo" en algunos casos.

1.2.3 Números pseudo-aleatorios

La mayoría de los métodos de simulación se basan en la posibilidad de generar números pseudoaleatorios que imiten las propiedades de valores independientes de la distribución $\mathcal{U}(0,1)$, es decir, que

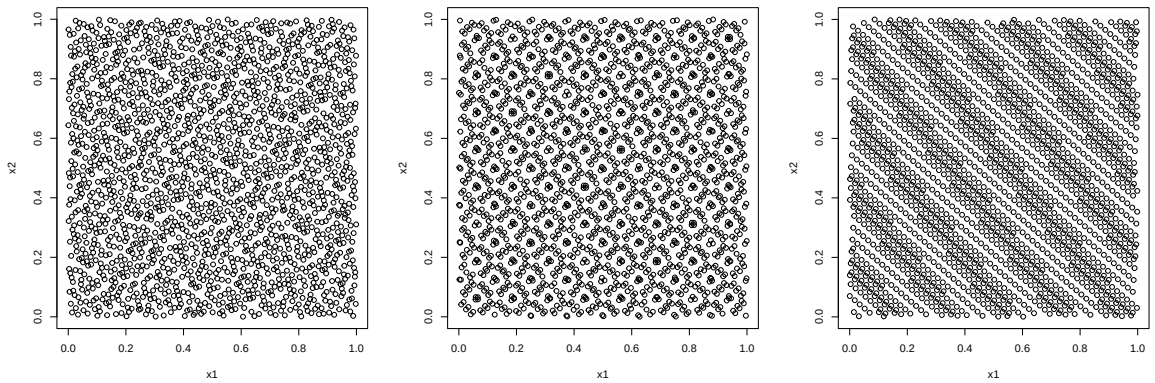


Figura 1.5: Secuencias cuasi-aleatorias bidimensionales obtenidas con los métodos de Halton (izquierda), Sobol (centro) y Torus (derecha).

imiten las propiedades de una muestra aleatoria simple[3] de esta distribución.

El procedimiento habitual para obtener estas secuencias es emplear un algoritmo recursivo denominado *generador*:

$$x_i = f\left(x_{i-1}, x_{i-2}, \cdots, x_{i-k}\right)$$

donde:

- k es el orden del generador.
- $(x_0, x_1, \cdots, x_{k-1})$ es la *semilla* (estado inicial).

El *periodo* o *longitud del ciclo* es la longitud de la secuencia antes de que vuelva a repetirse. Lo denotaremos por p.

Los números de la sucesión son predecibles, conociendo el algoritmo y la semilla. Sin embargo, si no se conociesen, *no se debería poder distinguir* una serie de números pseudoaleatorios *de una sucesión de números verdaderamente aleatoria* (utilizando recursos computacionales razonables). En caso contrario esta predecibilidad puede dar lugar a serios problemas (e.g. http://eprint.iacr.org/2007/419).

Como regla general, por lo menos mientras se está desarrollando un programa, interesa *fijar la semilla de aleatorización*.

- Permite la reproducibilidad de los resultados.
- Facilita la depuración del código.

Todo generador de números pseudoaleatorios mínimamente aceptable debe comportarse como si proporcionase muestras genuinas de datos independientes de una $\mathcal{U}(0,1)$. Otras propiedades de interés son:

- Reproducibilidad a partir de la semilla.
- Periodo suficientemente largo.
- Eficiencia (rapidez y requerimientos de memoria).
- Portabilidad.
- Generación de sub-secuencias (computación en paralelo).

[3] Aunque hay que distinguir entre secuencia y muestra. En un problema de inferencia, en principio estamos interesados en una característica desconocida de la población. En cambio, en un problema de simulación "la población" es el modelo y lo conocemos por completo (no obstante el problema de simulación puede surgir como solución de un problema de inferencia).

- Parsimonia.

Es importante asegurarse de que el generador empleado es adecuado:

> "Random numbers should not be generated with a method chosen at random."
>
> — Knuth, D.E. (TAOCP, 2002)

Se dispone de una gran cantidad de algoritmos. Los primeros intentos (cuadrados medios, método de Lehmer…) resultaron infructuosos, pero al poco tiempo ya se propusieron métodos que podían ser ampliamente utilizados (estableciendo adecuadamente sus parámetros). Entre ellas podríamos destacar:

- Generadores congruenciales.
- Registros desfasados.
- Combinaciones de distintos algoritmos.

La recomendación sería emplear un algoritmo conocido y que haya sido estudiado en profundidad (por ejemplo el generador *Mersenne-Twister* empleado por defecto en R, propuesto por Matsumoto y Nishimura, 1998). Además, sería recomendable utilizar alguna de las implementaciones disponibles en múltiples librerías, por ejemplo:

- GNU Scientific Library (GSL): http://www.gnu.org/software/gsl/manual
- StatLib: http://lib.stat.cmu.edu
- Numerical recipes: http://www.nrbook.com/nr3
- UNU.RAN (paquete `Runuran`): http://statmath.wu.ac.at/unuran

En este libro nos centraremos en los generadores congruenciales, descritos en la Sección 2.1. Estos métodos son muy simples, aunque con las opciones adecuadas podrían ser utilizados en pequeños estudios de simulación. Sin embargo, su principal interés es que constituyen la base de los generadores avanzados habitualmente considerados.

1.3 Números aleatorios en R

La generación de números pseudoaleatorios en R es una de las mejores disponibles en paquetes estadísticos. Entre las herramientas implementadas en el paquete base de R podemos destacar:

- `set.seed(entero)`: permite establecer la semilla (y el generador).
- `RNGkind()`: selecciona el generador.
- `rdistribución(n,...)`: genera valores aleatorios de la correspondiente distribución. Por ejemplo, `runif(n, min = 0, max = 1)`, generaría `n` valores de una uniforme. Se puede acceder al listado completo de las funciones disponibles en el paquete `stats` mediante el comando `?distributions`.
- `sample()`: genera muestras aleatorias de variables discretas y permutaciones (se tratará en el Capítulo 5).
- `simulate()`: genera realizaciones de la respuesta de un modelo ajustado.

Además están disponibles otros paquetes que implementan distribuciones adicionales (ver CRAN Task View: Probability Distributions). Entre ellos podríamos destacar los paquetes `distr` (clases S4; con extensiones en otros paquetes) y `distr6` (clases R6).

La semilla se almacena en `.Random.seed`:

- Inicialmente no existe. La recomendación es establecerla con `set.seed()`, en caso contrario se generará a partir del reloj del sistema[4] cuando se necesite.

[4] y del identificador asignado por el sistema operativo al proceso.

- Se almacena como un objeto oculto en el entorno de trabajo (o entorno global `.GlobalEnv`). Con las opciones por defecto de R, si al terminar una sesión almacenamos el entorno (en un fichero *.RData*), al iniciar una nueva sesión se restaurará también la semilla (y se podría continuar con las simulaciones).

- Es un vector de enteros cuya estructura depende del tipo de generador (en la Sección 2.2 se dan algunos detalles sobre la configuración por defecto), por lo que no debería ser modificado manualmente.
  Puede ser recomendable almacenar (el objeto completo) antes de generar simulaciones, e.g. `seed <- .Random.seed`. Esto permite reproducir los resultados y facilita la depuración de posibles errores.

En la mayoría de los ejemplos de este libro se generan todos los valores de una vez, se guardan y se procesan vectorialmente (normalmente empleando la función `apply`). En problemas mas complejos, en los que no es necesario almacenar todas las simulaciones, puede ser preferible emplear un bucle para generar y procesar cada simulación iterativamente. Por ejemplo podríamos emplear el siguiente esquema:

```
# Fijar semilla
set.seed(1)
for (isim in 1:nsim) {
  seed <- .Random.seed
  # Si se produce un error, podremos depurarlo ejecutando:
  #  .Random.seed <- seed
  ...
  # Generar valores pseudoaleatorios
  ...
}
```

o alternativamente fijar la semilla en cada iteración, por ejemplo:

```
for (isim in 1:nsim) {
  set.seed(isim)
  ...
  # Generar valores pseudoaleatorios
  ...
}
```

1.3.1 Opciones

Normalmente no nos va a interesar cambiar las opciones por defecto de R para la generación de números pseudoaleatorios. Para establecer estas opciones podemos emplear los argumentos `kind = NULL`, `normal.kind = NULL` y `sample.kind = NULL` en las funciones `RNGkind()` o `set.seed()`. A continuación se muestran las distintas opciones (resaltando en negrita los valores por defecto):

- `kind` especifica el generador pseudoaleatorio (uniforme):
  - "Wichmann-Hill": Ciclo 6.9536×10^{12}
  - "Marsaglia-Multicarry": Ciclo mayor de 2^{60}
  - "Super-Duper": Ciclo aprox. 4.6×10^{18} (S-PLUS)
  - **"Mersenne-Twister"**: Ciclo $2^{19937} - 1$ y equidistribution en 623 dimensiones.
  - "Knuth-TAOCP-2002": Ciclo aprox. 2^{129}.
  - "Knuth-TAOCP"
  - "user-supplied": permite emplear generadores adicionales.

- `normal.kind` selecciona el método de generación de normales (se tratará más adelante):

"Kinderman-Ramage", "Buggy Kinderman-Ramage", "Ahrens-Dieter", "Box-Muller", **"Inversion"** , o "user-supplied".

- `sample.kind` selecciona el método de generación de uniformes discretas (el empleado por la función `sample()`, que cambió ligeramente[5] a partir de la versión 3.6.0 de R): "Rounding" (versión anterior a 3.6.0) o **"Rejection"**.

Estas opciones están codificadas (con índices comenzando en 0) en el primer componente de la semilla:

```
set.seed(1)
.Random.seed[1]
```

```
## [1] 10403
```

Los dos últimos dígitos se corresponden con el generador, las centenas con el método de generación de normales y las decenas de millar con el método uniforme discreto.

1.3.2 Paquetes de R

Otros paquetes de R que pueden ser de interés:

- `setRNG` contiene herramientas que facilitan operar con la semilla (dentro de funciones,...).
- `random` permite la descarga de números "true random" desde RANDOM.ORG.
- `randtoolbox` implementa generadores más recientes (`rngWELL`) y generación de secuencias cuasi-aleatorias.
- `RDieHarder` implementa diversos contrastes para el análisis de la calidad de un generador y varios generadores.
- `Runuran` interfaz para la librería UNU.RAN para la generación (automática) de variables aleatorias no uniformes (ver Hörmann et al., 2004).
- `rsprng`, `rstream` y `rlecuyer` implementan la generación de múltiples secuencias (para programación paralela).
- `gls`, `rngwell19937`, `randaes`, `SuppDists`, `lhs`, `mc2d`, `fOptions`, ...

1.3.3 Tiempo de CPU

La velocidad del generador suele ser una característica importante (también medir los tiempos, de cada iteración y de cada procedimento, en estudios de simulación). Para evaluar el rendimiento están disponibles en R distintas herramientas:

- `proc.time()`: permite obtener tiempo de computación real y de CPU.

  ```
  tini <- proc.time()
  # Código a evaluar
  tiempo <- proc.time() - tini
  ```

- `system.time(expresión)`: muestra el tiempo de computación (real y de CPU) de expresión.

Por ejemplo, podríamos emplear las siguientes funciones para ir midiendo los tiempos de CPU durante una simulación:

```
CPUtimeini <- function() {
  .tiempo.ini <<- proc.time()
  .tiempo.last <<- .tiempo.ini
}

CPUtimeprint <- function() {
  tmp <- proc.time()
```

[5] Para evitar problemas de redondeo con tamaños extremadamente grandes; ver bug PR#17494.

```
  cat("Tiempo última operación:\n")
  print(tmp-.tiempo.last)
  cat("Tiempo total operación:\n")
  print(tmp-.tiempo.ini)
  .tiempo.last <<- tmp
}
```

Llamando a `CPUtimeini()` donde se quiere empezar a contar, y a `CPUtimeprint()` para imprimir el tiempo total y el tiempo desde la última llamada a una de estas funciones. Ejemplo:

```
funtest <- function(n) mad(runif(n))
CPUtimeini()
result1 <- funtest(10^6)
CPUtimeprint()
```

```
## Tiempo última operación:
##    user  system elapsed
##    0.27    0.00    0.26
## Tiempo total operación:
##    user  system elapsed
##    0.27    0.00    0.26
```

```
result2 <- funtest(10^3)
CPUtimeprint()
```

```
## Tiempo última operación:
##    user  system elapsed
##       0       0       0
## Tiempo total operación:
##    user  system elapsed
##    0.27    0.00    0.26
```

Hay diversos paquetes que implementan herramientas similares, por ejemplo:

- El paquete `tictoc`:
  - `tic("mensaje")`: inicia el temporizador y almacena el tiempo de inicio junto con el mensaje en una pila.
  - `toc()`: calcula el tiempo transcurrido desde la llamada correspondiente a `tic()`.

```
library(tictoc)
## Timing nested code
tic("outer")
   result1 <- funtest(10^6)
   tic("middle")
      result2 <- funtest(10^3)
      tic("inner")
         result3 <- funtest(10^2)
      toc() # inner
```

```
## inner: 0 sec elapsed
```

```
   toc() # middle
```

```
## middle: 0 sec elapsed
```

```
toc() # outer
```

```
## outer: 0.27 sec elapsed
```

```
## Timing in a loop and analyzing the results later using tic.log().
tic.clearlog()
```

```r
for (i in 1:10)
{
   tic(i)
   result <- funtest(10^4)
   toc(log = TRUE, quiet = TRUE)
}
# log.txt <- tic.log(format = TRUE)
# log.lst <- tic.log(format = FALSE)
log.times <- do.call(rbind.data.frame, tic.log(format = FALSE))
str(log.times)
```

```
## 'data.frame':    10 obs. of  3 variables:
##  $ tic: num  5.64 5.64 5.64 5.64 5.64 5.64 5.64 5.64 5.64 5.64
##  $ toc: num  5.64 5.64 5.64 5.64 5.64 5.64 5.64 5.64 5.64 5.64
##  $ msg: chr  "1" "2" "3" "4" ...
```

```r
tic.clearlog()

# timings <- unlist(lapply(log.lst, function(x) x$toc - x$tic))
log.times$timings <- with(log.times, toc - tic)
summary(log.times$timings)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##       0       0       0       0       0       0
```

- La función `cpu.time()` del paquete `simres`:
  - `cpu.time(restart = TRUE)`: inicia el temporizador y almacena el tiempo de inicio.
  - `cpu.time()`: calcula el tiempo (real y de CPU) total (desde tiempo de inicio) y parcial (desde la última llamada a esta función).

Hay que tener en cuenta que, por construcción, aunque se realicen en la mismas condiciones (en el mismo equipo), los tiempos de CPU en R pueden variar "ligeramente" entre ejecuciones. Si se quieren estudiar tiempos de computación de forma más precisa, se recomendaría promediar los tiempos de varias ejecuciones. Para ello se pueden emplear las herramientas del paquete `microbenchmark`. No obstante, para los fines de este libro no será necesaria tanta precisión.

Finalmente, si los tiempos de computación no fuesen asumibles, para identificar los cuellos de botella y mejorar el código para optimizar la velocidad, podríamos emplear la función `Rprof(fichero)`. Esta función permite evaluar el rendimiento muestreando la pila en intervalos para determinar en que funciones se emplea el tiempo de computación. Después de ejecutar el código, llamando a `Rprof(NULL)` se desactiva el muestreo y con `summaryRprof(fichero)` se muestran los resultados (para analizarlos puede resultar de utilidad el paquete `proftools`).

1.4 Ejercicios

Ejercicio 1.1

Sea (X, Y) es un vector aleatorio con distribución uniforme en el cuadrado $[-1, 1] \times [-1, 1]$ de área 4.

a) Aproximar mediante simulación $P(X + Y \leq 0)$ y compararla con la probabilidad teórica (obtenida aplicando la regla de Laplace $\frac{\text{área favorable}}{\text{área posible}}$).

b) Aproximar el valor de π mediante simulación a partir de $P(X^2 + Y^2 \leq 1)$.

Ver solución en Sección D.1.1.

Ejercicio 1.2 (Experimento de Bernoulli)

Consideramos el experimento de Bernoulli consistente en el lanzamiento de una moneda.

a) Empleando la función `sample`, obtener 1000 simulaciones del lanzamiento de una moneda `(0 = cruz, 1 = cara)`, suponiendo que no está trucada. Aproximar la probabilidad de cara a partir de las simulaciones.

b) En R pueden generarse valores de la distribución de Bernoulli mediante la función `rbinom(nsim, size=1, prob)`. Generar un gráfico de lineas considerando en el eje X el número de lanzamientos (de 1 a 10000) y en el eje Y la frecuencia relativa del suceso cara (puede ser recomendable emplear la función `cumsum`).

Ver solución en Sección D.1.2.

Ejercicio 1.3 (Simulación de un circuito)

Simular el paso de corriente a través del circuito mostrado en la Figura 1.6, donde se muestran las probabilidades de que pase corriente por cada uno de los interruptores, que se suponen variables aleatorias de Bernoulli independientes.

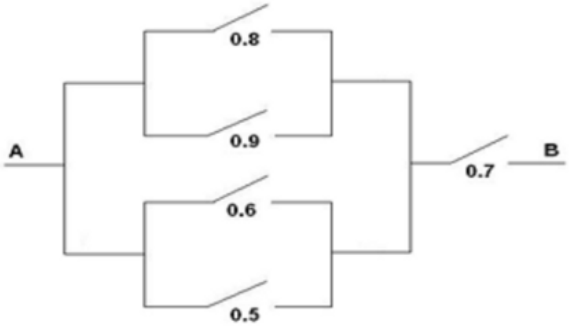


Figura 1.6: Esquema de un circuito eléctrico con interruptores aleatorios.

Nota: R maneja internamente los valores lógicos como 1 (`TRUE`) y 0 (`FALSE`). Recíprocamente, cualquier número puede ser tratado como lógico (al estilo de C). El entero 0 es equivalente a `FALSE` y cualquier entero distinto de 0 a `TRUE`.

Ver solución en Sección D.1.3.

Ejercicio 1.4 (El problema del Caballero de Méré)

En 1651, el Caballero de Méré le planteó a Pascal una pregunta relacionada con las apuestas y los juegos de azar: ¿es ventajoso apostar a que en cuatro lanzamientos de un dado se obtiene al menos un seis? Este problema generó una fructífera correspondencia entre Pascal y Fermat que se considera, simbólicamente, como el nacimiento del Cálculo de Probabilidades.

a) Escribir una función que simule el lanzamiento de n dados. El parámetro de entrada es el número de lanzamientos n, que toma el valor 4 por defecto, y la salida debe ser `TRUE` si se obtiene al menos un 6 y `FALSE` en caso contrario.

b) Utilizar la función anterior para simular $nsim = 10000$ jugadas de este juego y calcular la proporción de veces que se gana la apuesta (obtener al menos un 6 en n lanzamientos), usando $n = 4$. Comparar el resultado con la probabilidad teórica $1 - (5/6)^n$.

Ver solución en Sección D.1.4.

Ejercicio 1.5 (variación del problema del coleccionista (cadena de Markov))

Continuando con el ejemplo de la Sección 1.1.1 (álbum con $n = 75$ cromos y sobres con $m = 6$). A partir de $nsim = 2000$ simulaciones de coleccionistas de cromos, aproximar por simulación la evolución del proceso de compra de un coleccionista (número de cromos distintos dependiendo de los sobres comprados).

Ver solución en Sección D.1.5.

Capítulo 2

Generación de números pseudoaleatorios

Como ya se comentó, los distintos métodos de simulación requieren disponer de secuencias de números pseudoaleatorios que imiten las propiedades de generaciones independientes de una distribución $\mathcal{U}(0,1)$. En primer lugar nos centraremos en el caso de los generadores congruenciales. A pesar de su simplicidad, podrían ser adecuados en muchos casos y constituyen la base de los generadores avanzados habitualmente considerados. Posteriormente se dará una visión de las diferentes herramientas para estudiar la calidad de un generador de números pseudoaleatorios.

2.1 Generadores congruenciales lineales

En los generadores congruenciales lineales se considera una combinación lineal de los últimos k enteros generados y se calcula su resto al dividir por un entero fijo m. En el método congruencial simple (de orden $k = 1$), partiendo de una semilla inicial x_0, el algoritmo secuencial es el siguiente:

$$\begin{aligned} x_i &= (ax_{i-1} + c) \bmod m \\ u_i &= \frac{x_i}{m} \\ i &= 1, 2, ... \end{aligned}$$

donde a (*multiplicador*), c (*incremento*) y m (*módulo*) son enteros positivos[1] fijados de antemano (los parámetros de este generador). Si $c = 0$ el generador se denomina congruencial *multiplicativo* (Lehmer, 1951) y en caso contrario se dice que es *mixto* (Rotenburg, 1960).

Obviamente los parámetros y la semilla determinan los valores generados, que también se pueden obtener de forma no recursiva:

$$x_i = \left(a^i x_0 + c \frac{a^i - 1}{a - 1} \right) \bmod m$$

Este método está implementado[2] en la función `rlcg()` del paquete `simres`, imitando el funcionamiento del generador uniforme de R (ver también `simres::rng()`; fichero *rng.R*):

```
simres::rlcg
```

[1] Se supone además que a, c y x_0 son menores que m, ya que, dadas las propiedades algebraicas de la suma y el producto en el conjunto de clases de resto módulo m (que es un anillo), cualquier otra elección de valores mayores o iguales que m tiene un equivalente verificando esta restricción.

[2] Aunque de forma no muy eficiente. Para evitar problemas computacionales, se recomienda realizar el cálculo de los valores empleando el método de Schrage (ver Bratley *et al.*, 1987; L'Ecuyer, 1988).

```
## function(n, seed = as.numeric(Sys.time()), a = 7^5, c = 0, m = 2^31 - 1) {
##   u <- numeric(n)
##   for(i in 1:n) {
##     seed <- (a * seed + c) %% m
##     u[i] <- seed/m # (seed + 1)/(m + 1)
##   }
##   # Almacenar semilla y parámetros
##   assign(".rng", list(seed = seed, type = "lcg",
##           parameters = list(a = a, c = c, m = m)), envir = globalenv())
##   # .rng <<- list(seed = seed, type = "lcg", parameters = list(a = a, c = c, m = m))
##   # Para continuar con semilla y parámetros:
##   #   with(.rng, rlcg(n, seed, parameters$a, parameters$c, parameters$m))
##   # Devolver valores
##   return(u)
## }
## <bytecode: 0x000000003cf11388>
## <environment: namespace:simres>
```

Ejemplos de parámetros:

- $c = 0$, $a = 2^{16} + 3 = 65539$ y $m = 2^{31}$, generador *RANDU* de IBM (**no recomendable**).
- $c = 0$, $a = 7^5 = 16807$ y $m = 2^{31} - 1$ (primo de Mersenne), Park y Miller (1988) *minimal standar*, empleado por las librerías IMSL y NAG.
- $c = 0$, $a = 48271$ y $m = 2^{31} - 1$ actualización del *minimal standar* propuesta por Park, Miller y Stockmeyer (1993).

A pesar de su simplicidad, una adecuada elección de los parámetros permite obtener de manera eficiente secuencias de números "aparentemente" i.i.d. $\mathcal{U}(0, 1)$. Durante los primeros años, el procedimiento habitual consistía en escoger m de forma que se pudiera realizar eficientemente la operación del módulo, aprovechando la arquitectura del ordenador (por ejemplo $m = 2^{31}$ si se emplean enteros con signo de 32 bits). Posteriormente se seleccionaban c y a de forma que el período p fuese lo más largo posible (o suficientemente largo), empleando los resultados mostrados a continuación.

Teorema 2.1 (Hull y Dobell 1962)

Un generador congruencial tiene período máximo ($p = m$) si y solo si:

1. c y m son primos relativos (i.e. $m.c.d.(c, m) = 1$).
2. $a - 1$ es múltiplo de todos los factores primos de m (i.e. $a \equiv 1 \bmod q$, para todo q factor primo de m).
3. Si m es múltiplo de 4, entonces $a - 1$ también lo ha de ser (i.e. $m \equiv 0 \bmod 4 \Rightarrow a \equiv 1 \bmod 4$).

Algunas consecuencias:

- Si m primo, $p = m$ si y solo si $a = 1$.
- Un generador multiplicativo no cumple la condición 1 ($m.c.d.(0, m) = m$).

Teorema 2.2

Un generador multiplicativo tiene período máximo ($p = m - 1$) si:

1. m es primo.
2. a es una raiz primitiva de m (i.e. el menor entero q tal que $a^q = 1 \bmod m$ es $q = m - 1$).

Además de preocuparse de la longitud del ciclo, las secuencias generadas deben aparentar muestras i.i.d. $\mathcal{U}(0,1)$.

Uno de los principales problemas es que los valores generados pueden mostrar una clara estructura reticular. Este es el caso por ejemplo del generador RANDU de IBM muy empleado en la década de los 70 (ver Figura 2.1)[3]. Por ejemplo, el conjunto de datos `randu` contiene 400 tripletas de números sucesivos obtenidos con la implementación de VAX/VMS 1.5 (1977).

```
library(simres)
system.time(u <- rlcg(n = 9999,
          seed = 543210, a = 2^16 + 3, c = 0, m = 2^31))
```

```
##    user  system elapsed
##       0       0       0
```

```
# xyz <- matrix(u, ncol = 3, byrow = TRUE)
xyz <- stats::embed(u, 3)
library(plot3D)
# points3D(xyz[,1], xyz[,2], xyz[,3], colvar = NULL, phi = 60,
#          theta = -50, pch = 21, cex = 0.2)
points3D(xyz[,3], xyz[,2], xyz[,1], colvar = NULL, phi = 60,
         theta = -50, pch = 21, cex = 0.2)
```

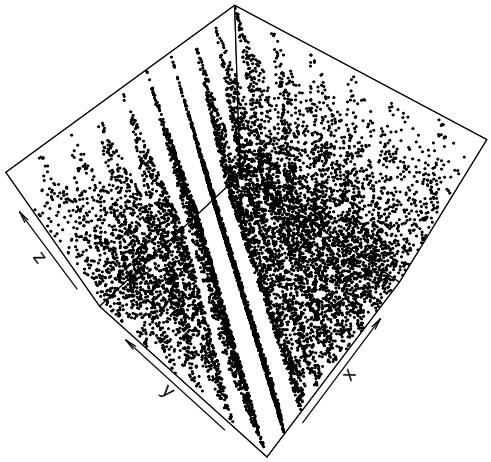


Figura 2.1: Grafico de dispersión de tripletas del generador RANDU de IBM (contenidas en 15 planos).

En general todos los generadores de este tipo van a presentar estructuras reticulares. Marsaglia (1968) demostró que las k-uplas de un generadores multiplicativo están contenidas en a lo sumo $(k!m)^{1/k}$ hiperplanos paralelos (para más detalles sobre la estructura reticular, ver por ejemplo Ripley, 1987, sección 2.7). Por tanto habría que seleccionar adecuadamente m y c (a solo influiría en la pendiente) de forma que la estructura reticular sea imperceptible teniendo en cuenta el número de datos que se

[3]Alternativamente se podría utilizar la función `plot3d` del paquete `rgl`, y rotar la figura (pulsando con el ratón) para ver los hiperplanos: `rgl::plot3d(xyz)`

pretende generar (por ejemplo de forma que la distancia mínima entre los puntos sea próxima a la esperada en teoría).

Se han propuesto diversas pruebas (ver Sección 2.3) para determinar si un generador tiene problemas de este tipo y se han realizado numerosos estudios para determinadas familias (e.g. Park y Miller, 1988, estudiaron que parámetros son adecuados para $m = 2^{31} - 1$).

- En ciertos contextos muy exigentes (por ejemplo en criptografía), se recomienda considerar un "periodo de seguridad" $\approx \sqrt{p}$ para evitar este tipo de problemas.
- Aunque estos generadores tienen limitaciones en su capacidad para producir secuencias muy largas de números i.i.d. $\mathcal{U}(0,1)$, son un elemento básico en generadores más avanzados (siguiente sección).

Ejemplo 2.1

Consideramos el generador congruencial, de ciclo máximo, definido por:

$$x_{n+1} = (5x_n + 1) \mod 512,$$
$$u_{n+1} = \frac{x_{n+1}}{512}, \; n = 0, 1, ...$$

a) Generar 500 valores de este generador, obtener el tiempo de CPU, representar su distribución mediante un histograma (en escala de densidades) y compararla con la densidad teórica.

```
set.rng(321, "lcg", a = 5, c = 1, m = 512)  # Establecer semilla y parámetros
nsim <- 500
system.time(u <- rng(nsim))
```

```
##    user  system elapsed
##       0       0       0
```

```
hist(u, freq = FALSE)
abline(h = 1)                    # Densidad uniforme
```

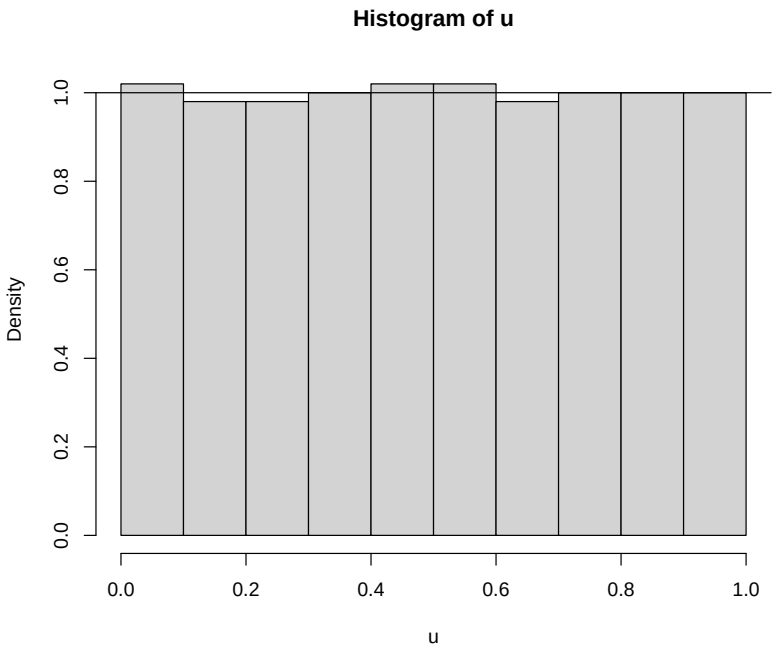


Figura 2.2: Histograma de los valores generados.

En este caso concreto la distribución de los valores generados es aparentemente más uniforme de lo que cabría esperar, lo que induciría a sospechar de la calidad de este generador (ver Ejemplo 2.2 en Sección 2.3).

b) Calcular la media de las simulaciones (`mean`) y compararla con la teórica.

La aproximación por simulación de la media teórica es:

```
mean(u)
```

```
## [1] 0.4999609
```

La media teórica es 0.5. Error absoluto 3.90625×10^{-5}.

c) Aproximar (mediante simulación) la probabilidad del intervalo $(0.4; 0.8)$ y compararla con la teórica.

La probabilidad teórica es 0.8 - 0.4 = 0.4

La aproximación mediante simulación:

```
sum((0.4 < u) & (u < 0.8))/nsim
```

```
## [1] 0.402
```

```
mean((0.4 < u) & (u < 0.8))      # Alternativa
```

```
## [1] 0.402
```

2.2 Extensiones

Se han considerado diversas extensiones del generador congruencial lineal simple:

- Lineal múltiple: $x_i = a_0 + a_1 x_{i-1} + a_2 x_{i-2} + \cdots + a_k x_{i-k} \bmod m$, con periodo $p \leq m^k - 1$.
- No lineal: $x_i = f(x_{i-1}, x_{i-2}, \cdots, x_{i-k}) \bmod m$. Por ejemplo $x_i = a_0 + a_1 x_{i-1} + a_2 x_{i-1}^2 \bmod m$.
- Matricial: $x_i = A_0 + A_1 x_{i-1} + A_2 x_{i-2} + \cdots + A_k x_{i-k} \bmod m$.

Un ejemplo de generador congruencia lineal múltiple es el denominado *generador de Fibonacci retardado* (Fibonacci-lagged generator; Knuth, 1969):

$$x_n = (x_{n-37} + x_{n-100}) \bmod 2^{30},$$

con un período aproximado de 2^{129} y que puede ser empleado en R (lo cual no sería en principio recomendable; ver Knuth Recent News 2002) estableciendo `kind` a `"Knuth-TAOCP-2002"` o `"Knuth-TAOCP"` en la llamada a `set.seed()` o `RNGkind()`.

El generador *Mersenne-Twister* (Matsumoto y Nishimura, 1998), empleado por defecto en R, de periodo $2^{19937} - 1$ y equidistribution en 623 dimensiones, se puede expresar como un generador congruencial matricial lineal. En cada iteración (*twist*) genera 624 valores (los últimos componentes de la semilla son los 624 enteros de 32 bits correspondientes, el segundo componente es el índice/posición correspondiente al último valor devuelto; el conjunto de enteros solo cambia cada 624 generaciones).

```
set.seed(1)
u <- runif(1)
seed <- .Random.seed
u <- runif(623)
sum(seed != .Random.seed)
```

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

```
# Solo cambia el índice:
seed[2]; .Random.seed[2]
```

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

```
## [1] 624
```

```
u <- runif(1)
# Cada 624 generaciones cambia el conjunto de enteros y el índice se inicializa
sum(seed != .Random.seed)
```

```
## [1] 624
```

```
seed[2]; .Random.seed[2]
```

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

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

Un caso particular del generador lineal múltiple son los denominados *generadores de registros desfasados* (más relacionados con la criptografía). Se generan bits de forma secuencial considerando $m = 2$ y $a_i \in \{0, 1\}$ y se van combinando l bits para obtener valores en el intervalo $(0, 1)$, por ejemplo $u_i = 0.x_{it+1}x_{it+2} \dots x_{it+l}$, siendo t un parámetro denominado *aniquilación* (Tausworthe, 1965). Los cálculos se pueden realizar rápidamente mediante operaciones lógicas (los sumandos de la combinación lineal se traducen en un "o" exclusivo XOR), empleando directamente los registros del procesador (ver por ejemplo, Ripley, 1987, Algoritmo 2.1).

Otras alternativas consisten en la combinanción de varios generadores, las más empleadas son:

- Combinar las salidas: por ejemplo $u_i = \sum_{l=1}^{L} u_i^{(l)} \bmod 1$, donde $u_i^{(l)}$ es el i-ésimo valor obtenido con el generador l.
- Barajar las salidas: por ejemplo se crea una tabla empleando un generador y se utiliza otro para seleccionar el índice del valor que se va a devolver y posteriormente actualizar.

El generador *L'Ecuyer-CMRG* (L'Ecuyer, 1999), empleado como base para la generación de múltiples secuencias en el paquete `parallel`, combina dos generadores concruenciales lineales múltiples de orden $k = 3$ (el periodo aproximado es 2^{191}).

2.3 Análisis de la calidad de un generador

Para verificar si un generador tiene las propiedades estadísticas deseadas hay disponibles una gran cantidad de test de hipótesis y métodos gráficos, incluyendo métodos genéricos (de bondad de ajuste y aleatoriedad) y contrastes específicos para generadores aleatorios. Se trata principalmente de contrastar si las muestras generadas son i.i.d. $\mathcal{U}(0, 1)$ (análisis univariante). Aunque los métodos más avanzados tratan de contrastar si las d-uplas:

$$(U_{t+1}, U_{t+2}, \dots, U_{t+d});\ t = (i-1)d,\ i = 1, \dots, m$$

son i.i.d. $\mathcal{U}(0, 1)^d$ (uniformes independientes en el hipercubo; análisis multivariante). En el Apéndice A se describen algunos de estos métodos.

En esta sección emplearemos únicamente métodos genéricos, ya que también pueden ser de utilidad para evaluar generadores de variables no uniformes y para la construcción de modelos del sistema real (e.g. para modelar variables que se tratarán como entradas del modelo general). Sin embargo, los métodos clásicos pueden no ser muy adecuados para evaluar generadores de números pseudoaleatorios (ver L'Ecuyer y Simard, 2007). La recomendación sería emplear baterías de contrastes recientes, como las descritas en la Subsección 2.3.2.

Hay que destacar algunas diferencias entre el uso de este tipo de métodos en inferencia y en simulación. Por ejemplo, si empleamos un constrate de hipótesis del modo habitual, desconfiamos del generador si la muestra (secuencia) no se ajusta a la distribución teórica (p-valor $\leq \alpha$). En simulación, además, también se sospecha si se ajusta demasiado bien a la distribución teórica (p-valor $\geq 1 - \alpha$), lo que indicaría que no reproduce adecuadamente la variabilidad.

Uno de los contrastes más conocidos es el test chi-cuadrado de bondad de ajuste (`chisq.test` para el caso discreto). Aunque si la variable de interés es continua, habría que discretizarla (con la correspondiente perdida de información). Por ejemplo, se podría emplear la función `simres::chisq.cont.test()` (fichero *test.R*), que imita a las incluidas en R:

```
simres::chisq.cont.test
```

```
## function(x, distribution = "norm", nclass = floor(length(x)/5),
##                              output = TRUE, nestpar = 0, ...) {
##   # Función distribución
##   q.distrib <- eval(parse(text = paste("q", distribution, sep = "")))
##   # Puntos de corte
##   q <- q.distrib((1:(nclass - 1))/nclass, ...)
##   tol <- sqrt(.Machine$double.eps)
##   xbreaks <- c(min(x) - tol, q, max(x) + tol)
##   # Gráficos y frecuencias
##   if (output) {
##     xhist <- hist(x, breaks = xbreaks, freq = FALSE,
##                   lty = 2, border = "grey50")
##     # Función densidad
##     d.distrib <- eval(parse(text = paste("d", distribution, sep = "")))
##     curve(d.distrib(x, ...), add = TRUE)
##   } else {
##     xhist <- hist(x, breaks = xbreaks, plot = FALSE)
##   }
##   # Cálculo estadístico y p-valor
##   O <- xhist$counts  # Equivalente a table(cut(x, xbreaks)) pero más eficiente
##   E <- length(x)/nclass
##   DNAME <- deparse(substitute(x))
##   METHOD <- "Pearson's Chi-squared test"
##   STATISTIC <- sum((O - E)^2/E)
##   names(STATISTIC) <- "X-squared"
##   PARAMETER <- nclass - nestpar - 1
##   names(PARAMETER) <- "df"
##   PVAL <- pchisq(STATISTIC, PARAMETER, lower.tail = FALSE)
##   # Preparar resultados
##   classes <- format(xbreaks)
##   classes <- paste("(", classes[-(nclass + 1)], ",", classes[-1], "]",
##                    sep = "")
##   RESULTS <- list(classes = classes, observed = O, expected = E,
##                   residuals = (O - E)/sqrt(E))
##   if (output) {
##     cat("\nPearson's Chi-squared test table\n")
##     print(as.data.frame(RESULTS))
##   }
##   if (any(E < 5))
##     warning("Chi-squared approximation may be incorrect")
##   structure(c(list(statistic = STATISTIC, parameter = PARAMETER, p.value = PVAL,
##                    method = METHOD, data.name = DNAME), RESULTS), class = "htest")
## }
## <bytecode: 0x0000000039d9a648>
## <environment: namespace:simres>
```

Ejemplo 2.2 (análisis de un generador congruencial continuación)

Continuando con el generador congruencial del Ejemplo 2.1:

```
set.rng(321, "lcg", a = 5, c = 1, m = 512)  # Establecer semilla y parámetros
nsim <- 500
```

```
u <- rng(nsim)
```

Al aplicar el test chi-cuadrado obtendríamos:

```
chisq.cont.test(u, distribution = "unif",
                nclass = 10, nestpar = 0, min = 0, max = 1)
```

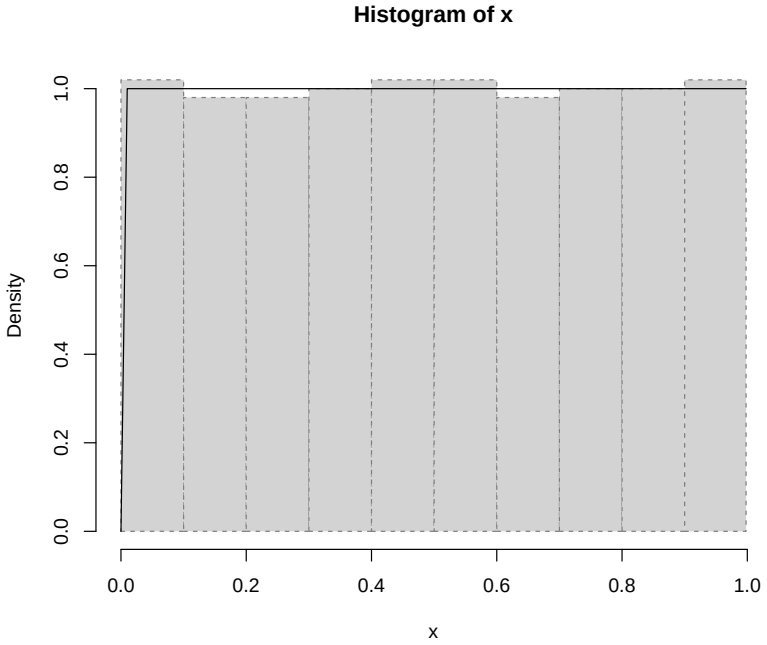


Figura 2.3: Gráfico resultante de aplicar la función ‘chisq.cont.test()‘ comparando el histograma de los valores generados con la densidad uniforme.

```
##
## Pearson's Chi-squared test table
##                          classes observed expected  residuals
## 1  (-1.490116e-08, 1.000000e-01]       51       50  0.1414214
## 2  ( 1.000000e-01, 2.000000e-01]       49       50 -0.1414214
## 3  ( 2.000000e-01, 3.000000e-01]       49       50 -0.1414214
## 4  ( 3.000000e-01, 4.000000e-01]       50       50  0.0000000
## 5  ( 4.000000e-01, 5.000000e-01]       51       50  0.1414214
## 6  ( 5.000000e-01, 6.000000e-01]       51       50  0.1414214
## 7  ( 6.000000e-01, 7.000000e-01]       49       50 -0.1414214
## 8  ( 7.000000e-01, 8.000000e-01]       50       50  0.0000000
## 9  ( 8.000000e-01, 9.000000e-01]       50       50  0.0000000
## 10 ( 9.000000e-01, 9.980469e-01]       50       50  0.0000000
```

```
##
##  Pearson's Chi-squared test
##
## data:  u
## X-squared = 0.12, df = 9, p-value = 1
```

Alternativamente, por ejemplo si solo se pretende aplicar el contraste, se podría emplear la función `simres::freq.test()` (fichero *test.R*) para este caso particular (ver Sección A.3.1).

Como se muestra en la Figura 2.3 el histograma de la secuencia generada es muy plano (comparado con lo que cabría esperar de una muestra de tamaño 500 de una uniforme), y consecuentemente el

p-valor del contraste chi-cuadrado es prácticamente 1, lo que indicaría que este generador no reproduce adecuadamente la variabilidad de una distribución uniforme.

Otro contraste de bondad de ajuste muy conocido es el test de Kolmogorov-Smirnov, implementado en `ks.test` (ver Sección A.1.5). Este contraste de hipótesis compara la función de distribución bajo la hipótesis nula con la función de distribución empírica (ver Sección A.1.2), representadas en la Figura 2.4:

```
# Distribución empírica
curve(ecdf(u)(x), type = "s", lwd = 2)
curve(punif(x, 0, 1), add = TRUE)
```

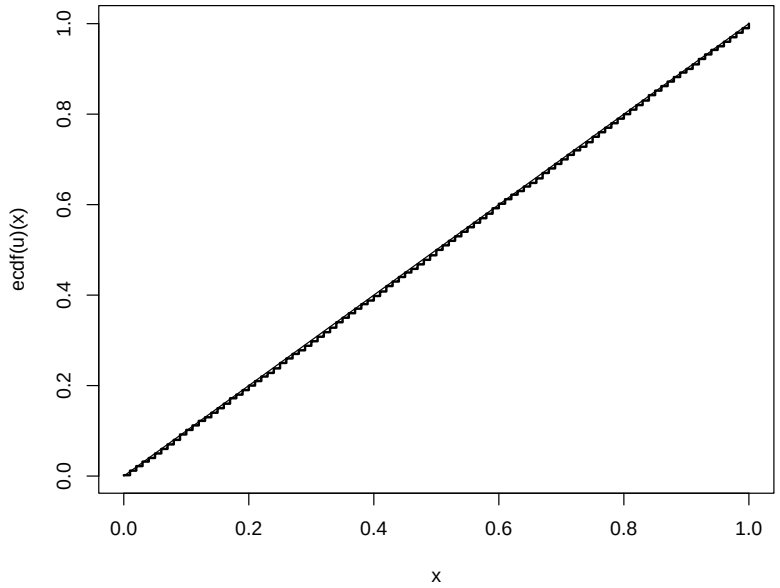


Figura 2.4: Comparación de la distribución empírica de la secuencia generada con la función de distribución uniforme.

Podemos realizar el contraste con el siguiente código:

```
# Test de Kolmogorov-Smirnov
ks.test(u, "punif", 0, 1)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  u
## D = 0.0033281, p-value = 1
## alternative hypothesis: two-sided
```

En la Sección A.1 se describen con más detalle estos contrastes de bondad de ajuste.

Adicionalmente podríamos estudiar la aleatoriedad de los valores generados (ver Sección A.2), por ejemplo mediante un gráfico secuencial y el de dispersión retardado.

```
plot(as.ts(u))
```

```
plot(u[-nsim],u[-1])
```

Si se observa algún tipo de patrón indicaría dependencia (se podría considerar como una versión

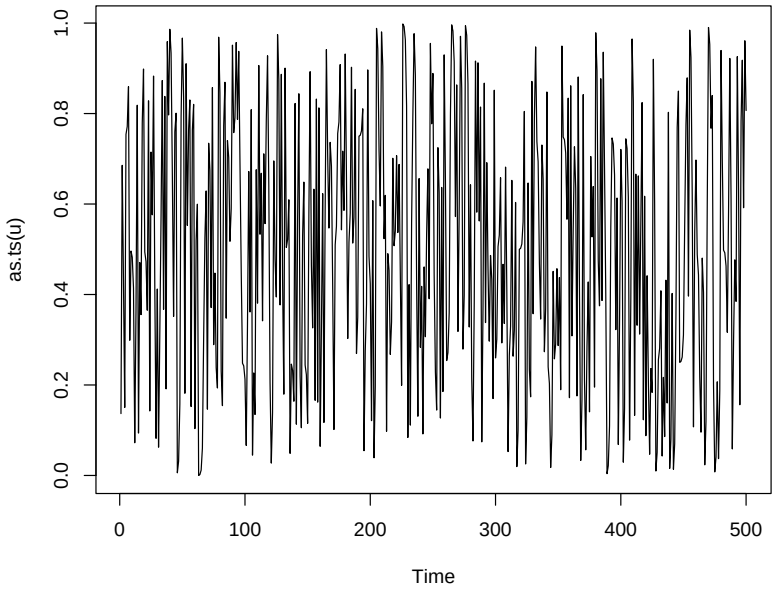


Figura 2.5: Gráfico secuencial de los valores generados.

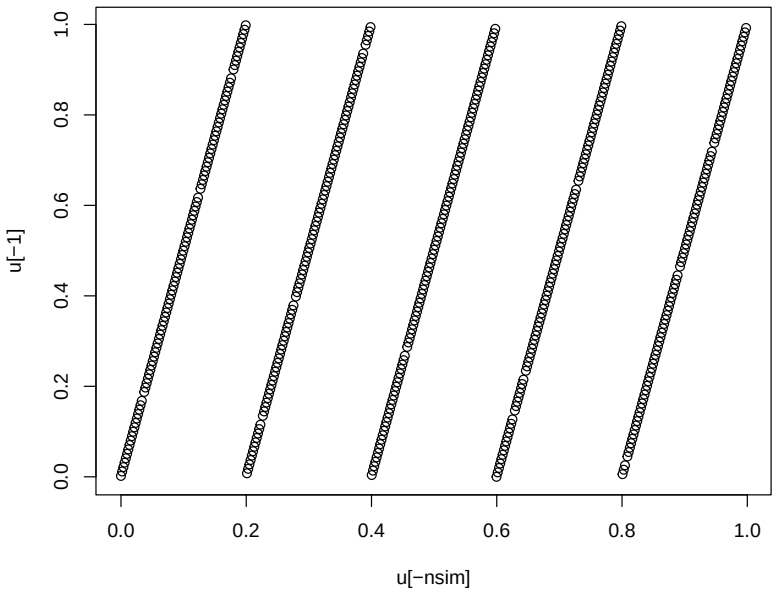


Figura 2.6: Gráfico de dispersión retardado de los valores generados.

descriptiva del denominado "Parking lot test"), ver Ejemplo A.2.

También podemos analizar las autocorrelaciones (las correlaciones de (u_i, u_{i+k}), con $k = 1, \dots, K$):

```
acf(u)
```

Por ejemplo, para contrastar si las diez primeras autocorrelaciones son nulas podríamos emplear el

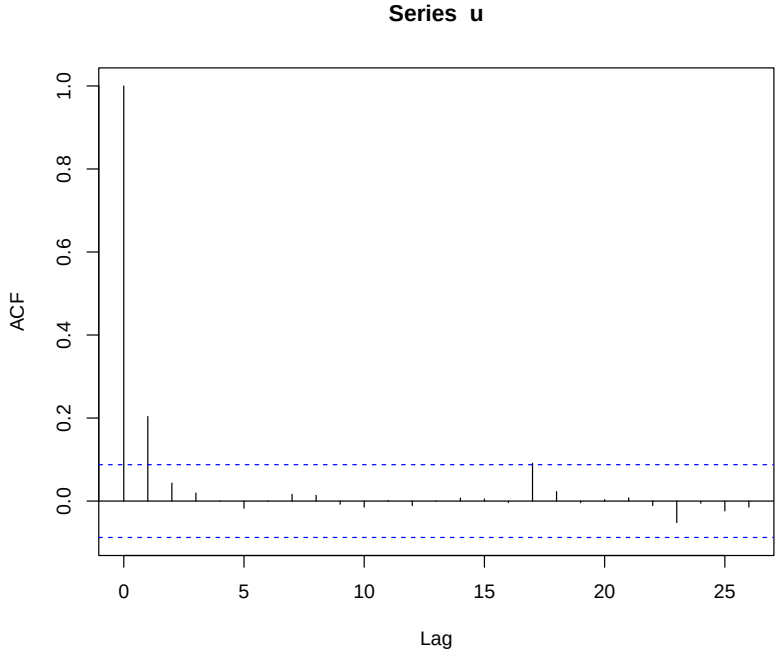


Figura 2.7: Autocorrelaciones de los valores generados.

test de Ljung-Box:

```
Box.test(u, lag = 10, type = "Ljung")
```

```
##
##  Box-Ljung test
##
## data:  u
## X-squared = 22.533, df = 10, p-value = 0.01261
```

2.3.1 Repetición de contrastes

Los contrastes se plantean habitualmente desde el punto de vista de la inferencia estadística: se realiza una prueba sobre la única muestra disponible. Si se realiza una única prueba, en las condiciones de H_0 hay una probabilidad α de rechazarla. En simulación tiene mucho más sentido realizar un gran número de pruebas:

- La proporción de rechazos debería aproximarse al valor de α (se puede comprobar para distintos valores de α).
- La distribución del estadístico debería ajustarse a la teórica bajo H_0 (se podría realizar un nuevo contraste de bondad de ajuste).
- Los p-valores obtenidos deberían ajustarse a una $\mathcal{U}(0,1)$ (se podría realizar también un contraste de bondad de ajuste).

Este procedimiento es también el habitual para validar un método de contraste de hipótesis por simulación (ver Sección 7.4.3).

Ejemplo 2.3

Continuando con el generador congruencial RANDU, podemos pensar en estudiar la uniformidad de los valores generados empleando repetidamente el test chi-cuadrado:

```
# Valores iniciales
set.rng(543210, "lcg", a = 2^16 + 3, c = 0, m = 2^31)  # Establecer semilla y parámetros
# set.seed(543210)
n <- 500
nsim <- 1000
estadistico <- numeric(nsim)
pvalor <- numeric(nsim)

# Realizar contrastes
for(isim in 1:nsim) {
  u <- rng(n)    # Generar
  # u <- runif(n)
  tmp <- freq.test(u, nclass = 100)
  # tmp <- chisq.cont.test(u, distribution = "unif", nclass = 100,
  #     output = FALSE, nestpar = 0, min = 0, max = 1)
  estadistico[isim] <- tmp$statistic
  pvalor[isim] <- tmp$p.value
}
```

Por ejemplo, podemos comparar la proporción de rechazos observados con los que cabría esperar con los niveles de significación habituales:

```
{
cat("Proporción de rechazos al 1% =", mean(pvalor < 0.01), "\n") # sum(pvalor < 0.01)/nsim
cat("Proporción de rechazos al 5% =", mean(pvalor < 0.05), "\n")   # sum(pvalor < 0.05)/nsim
cat("Proporción de rechazos al 10% =", mean(pvalor < 0.1), "\n")   # sum(pvalor < 0.1)/nsim
}
```

```
## Proporción de rechazos al 1% = 0.014
## Proporción de rechazos al 5% = 0.051
## Proporción de rechazos al 10% = 0.112
```

Las proporciones de rechazo obtenidas deberían comportarse como una aproximación por simulación de los niveles teóricos. En este caso no se observa nada extraño, por lo que no habría motivos para sospechar de la uniformidad de los valores generados (aparentemente no hay problemas con la uniformidad de este generador).

Adicionalmente, si queremos estudiar la proporción de rechazos (el *tamaño del contraste*) para los posibles valores de α, podemos emplear la distribución empírica del p-valor (proporción de veces que resultó menor que un determinado valor):

```
# Distribución empírica
plot(ecdf(pvalor), do.points = FALSE, lwd = 2,
     xlab = 'Nivel de significación', ylab = 'Proporción de rechazos')
abline(a = 0, b = 1, lty = 2)   # curve(punif(x, 0, 1), add = TRUE)
```

También podemos estudiar la distribución del estadístico del contraste. En este caso, como la distribución bajo la hipótesis nula está implementada en R, podemos compararla fácilmente con la de los valores generados (debería ser una aproximación por simulación de la distribución teórica):

```
# Histograma
hist(estadistico, breaks = "FD", freq = FALSE, main = "")
curve(dchisq(x, 99), add = TRUE)
```

Además de la comparación gráfica, podríamos emplear un test de bondad de ajuste para contrastar si la distribución del estadístico es la teórica bajo la hipótesis nula:

```
# Test chi-cuadrado (chi-cuadrado sobre chi-cuadrado)
# chisq.cont.test(estadistico, distribution="chisq", nclass=20, nestpar=0, df=99)
# Test de Kolmogorov-Smirnov
```

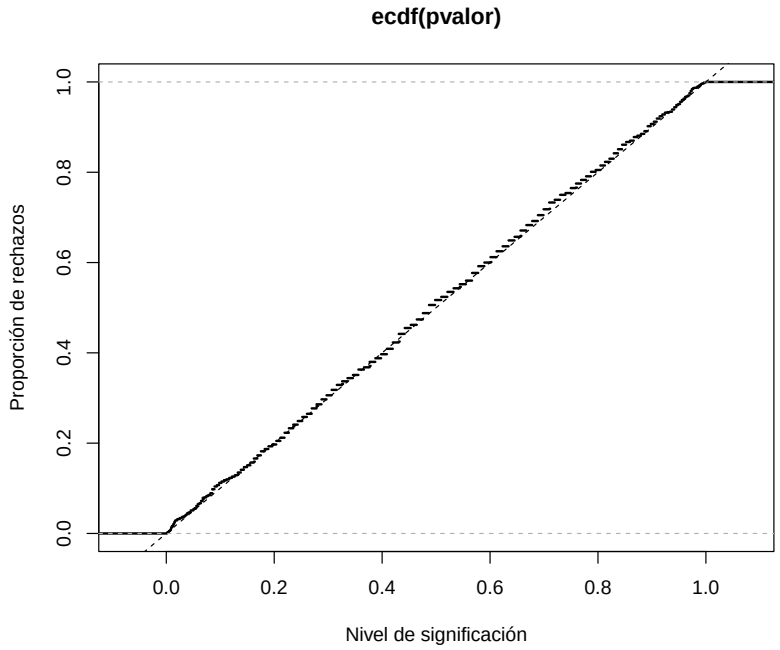


Figura 2.8: Proporción de rechazos con los distintos niveles de significación.

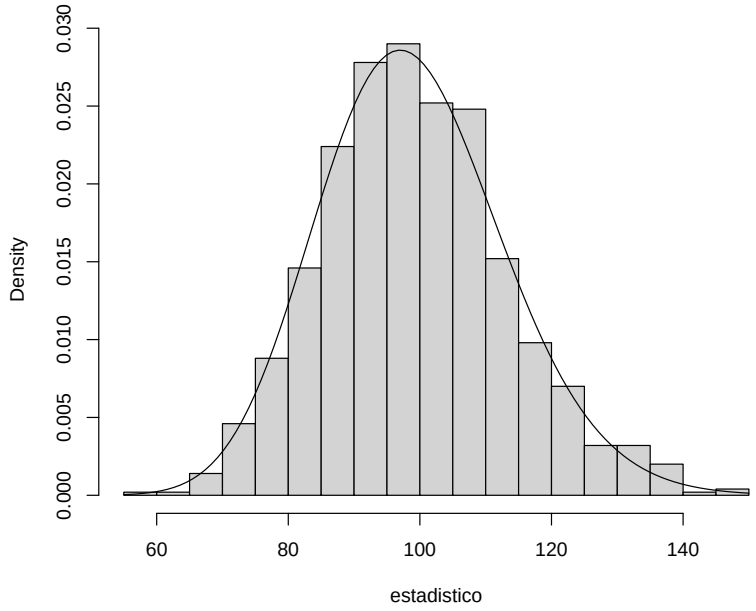


Figura 2.9: Distribución del estadístico del constraste.

```
ks.test(estadistico, "pchisq", df = 99)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
```

```
## data:  estadistico
## D = 0.023499, p-value = 0.6388
## alternative hypothesis: two-sided
```

En este caso la distribución observada del estadístico es la que cabría esperar de una muestra de este tamaño de la distribución teórica, por tanto, según este criterio, aparentemente no habría problemas con la uniformidad de este generador (hay que recordar que estamos utilizando contrastes de hipótesis como herramienta para ver si hay algún problema con el generador, no tiene mucho sentido hablar de aceptar o rechazar una hipótesis).

En lugar de estudiar la distribución del estadístico de contraste siempre podemos analizar la distribución del p-valor. Mientras que la distribución teórica del estadístico depende del contraste y puede ser complicada, la del p-valor es siempre una uniforme.

```r
# Histograma
hist(pvalor, freq = FALSE, main = "")
abline(h=1) # curve(dunif(x,0,1), add=TRUE)
```

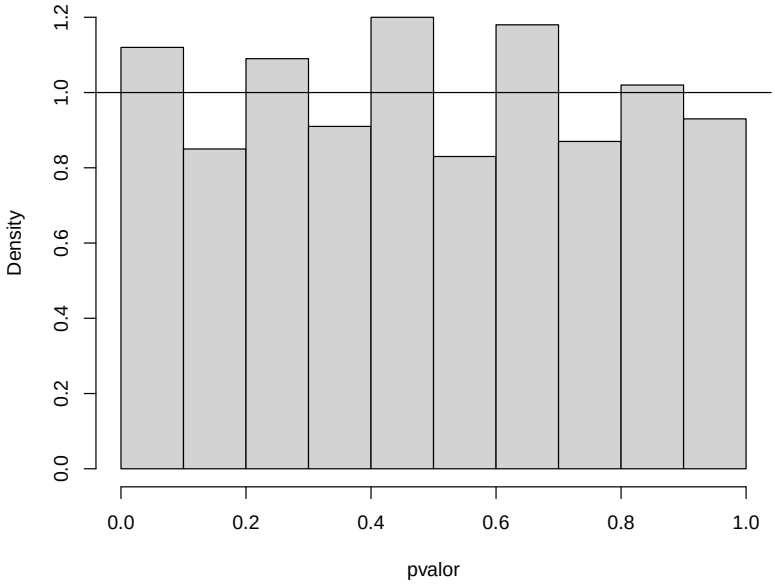


Figura 2.10: Distribución del p-valor del constraste.

```r
# Test chi-cuadrado
# chisq.cont.test(pvalor, distribution="unif", nclass=20, nestpar=0, min=0, max=1)
# Test de Kolmogorov-Smirnov
ks.test(pvalor, "punif",  min = 0, max = 1)
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  pvalor
## D = 0.023499, p-value = 0.6388
## alternative hypothesis: two-sided
```

Como podemos observar, obtendríamos los mismos resultados que al analizar la distribución del estadístico.

Alternativamente podríamos emplear la función `rephtest()` del paquete `simres` (fichero *test.R*):

```
set.rng(543210, "lcg", a = 2^16 + 3, c = 0, m = 2^31)
# res <- rephtest(n = 30, test = chisq.cont.test, rand.gen = rng,
#           distribution = "unif", output = FALSE, nestpar = 0)
res <- rephtest(n = 30, test = freq.test, rand.gen = rng, nclass = 6)
str(res)
```

```
## List of 2
##  $ statistics: num [1:1000] 5.2 6.8 12.4 0.8 5.6 7.6 6.4 9.6 5.2 3.2 ...
##  $ p.values  : num [1:1000] 0.392 0.2359 0.0297 0.977 0.3471 ...
##  - attr(*, "class")= chr "rhtest"
##  - attr(*, "method")= chr "Chi-squared test for given probabilities"
##  - attr(*, "names.stat")= chr "X-squared"
##  - attr(*, "parameter")= Named num 5
##   ..- attr(*, "names")= chr "df"
```

```
summary(res)
```

```
## Proportion of rejections:
##    1%    5%   10%   25%   50%
## 0.013 0.054 0.096 0.255 0.544
```

```
old.par <- par(mfrow = c(1, 2))
plot(res, 2:3)
```

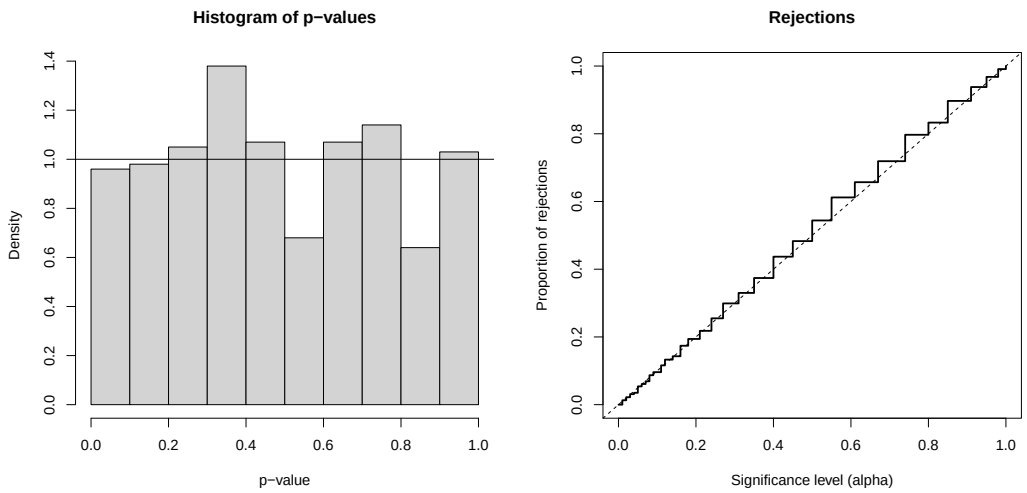


Figura 2.11: Distribución de los p-valores y proporción de rechazos.

```
par(old.par)
```

2.3.2 Baterías de contrastes

Hay numerosos ejemplos de generadores que pasaron diferentes test de uniformidad y aleatoriedad pero que fallaron estrepitosamente al considerar nuevos contrastes diseñados específicamente para generadores aleatorios (ver Marsaglia *et al.*, 1990). Por este motivo, el procedimiento habitual en la práctica es aplicar un número más o menos elevado de contrastes (de distinto tipo y difíciles de pasar, e.g. Marsaglia y Tsang, 2002), de forma que si el generador los pasa tendremos mayor confianza en que sus propiedades son las adecuadas. Este conjunto de pruebas es lo que se denomina batería de contrastes. Una de las primeras se introdujo en Knuth (1969) y de las más recientes podríamos destacar:

- Diehard tests (The Marsaglia Random Number CDROM, 1995): http://www.stat.fsu.edu/pub/diehard (versión archivada el 2016-01-25).

- Dieharder (Brown y Bauer, 2003): Dieharder Page, paquete `RDieHarder`.
- TestU01 (L'Ecuyer y Simard, 2007): http://simul.iro.umontreal.ca/testu01/tu01.html.
- NIST test suite (National Institute of Standards and Technology, USA, 2010): http://csrc.nist.gov/groups/ST/toolkit/rng.

Para más detalles, ver por ejemplo[4]:

- Marsaglia, G. y Tsang, W.W. (2002). Some difficult-to-pass tests of randomness. Journal of Statistical Software, 7(3), 1-9.
- Demirhan, H. y Bitirim, N. (2016). CryptRndTest: an R package for testing the cryptographic randomness. The R Journal, 8(1), 233-247.

Estas baterías de contrastes se suelen emplear si el generador va a ser utilizado en criptografía o si es muy importante la impredecibilidad (normalmente con generadores de números "verdaderamente aleatorios" por hardware). Si el objetivo es únicamente obtener resultados estadísticos (como en nuestro caso) no sería tan importante que el generador no superase alguno de estos test.

2.4 Ejercicios

Ejercicio 2.1 (Método de los cuadrados medios)

Uno de los primeros generadores utilizados fue el denominado método de los cuadrados medios propuesto por Von Neumann (1946). Con este procedimiento se generan números pseudoaleatorios de 4 dígitos de la siguiente forma:

i. Se escoge un número de cuatro dígitos x_0 (semilla).

ii. Se eleva al cuadrado (x_0^2) y se toman los cuatro dígitos centrales (x_1).

iii. Se genera el número pseudo-aleatorio como

$$u_1 = \frac{x_1}{10^4}.$$

iv. Volver al paso ii y repetir el proceso.

Para obtener los k (número par) dígitos centrales de x_i^2 se puede utilizar que:

$$x_{i+1} = \left\lfloor \left(x_i^2 - \left\lfloor \frac{x_i^2}{10^{(2k-\frac{k}{2})}} \right\rfloor 10^{(2k-\frac{k}{2})} \right) / 10^{\frac{k}{2}} \right\rfloor$$

Estudiar las características de este generador a partir de una secuencia de 500 valores. Emplear únicamente métodos gráficos.

Este algoritmo está implementado en la función `simres::rvng()` (ver también `simres::rng()`; fichero *rng.R*):

```
simres::rvng
```

```
## function(n, seed = as.numeric(Sys.time()), k = 4) {
##   seed <- seed %% 10^k
##   aux <- 10^(2*k-k/2)
##   aux2 <- 10^(k/2)
##   u <- numeric(n)
##   for(i in 1:n) {
##     z <- seed^2
##     seed <- trunc((z - trunc(z/aux)*aux)/aux2)
##     u[i] <- seed/10^k
```

[4]También puede ser de interés el enlace Randomness Tests: A Literature Survey y la entidad certificadora (gratuita) en línea CAcert.

```
##   }
##   # Almacenar semilla y parámetros
##   assign(".rng", list(seed = seed, type = "vm", parameters = list(k = k)),
##       envir = globalenv())
##   # .rng <<- list(seed = seed, type = "vm", parameters = list(k = k))
##   # Para continuar con semilla y parámetros:
##   #   with(.rng, rvng(n, seed, parameters$k))
##   # Devolver valores
##   return(u)
## }
## <bytecode: 0x000000003ff4da60>
## <environment: namespace:simres>
```

Ejercicio 2.2

Considerando el generador congruencial multiplicativo de parámetros $a = 7^5 = 16807$, $c = 0$ y $m = 2^{31} - 1$ (*minimal standar* de Park y Miller, 1988). ¿Se observan los mismos problemas que con el algoritmo RANDU al considerar las tripletas (x_k, x_{k+1}, x_{k+2})?

Capítulo 3

Análisis de resultados de simulación

En este capítulo nos centraremos en la aproximación mediante simulación de la media teórica de un estadístico a partir de la media muestral de una secuencia de simulaciones de dicho estadístico. La aproximación de una probabilidad sería un caso particular considerando una variable de Bernoulli.

En primer lugar se tratará el análisis de la convergencia y la precisión de la aproximación por simulación. Al final del capítulo se incluye una breve introducción a los problemas de estabilización y dependencia (con los que nos solemos encontrar en simulación dinámica y MCMC).

3.1 Convergencia

Supongamos que estamos interesados en aproximar la media teórica $\mu = E(X)$ a partir de una secuencia i.i.d. X_1, X_2, ..., X_n obtenida mediante simulación, utilizando para ello la media muestral $\bar{X}_n$. Una justificación teórica de la validez de esta aproximación es *la ley (débil*[1]*) de los grandes números*:

Teorema 3.1 (Ley débil de los grandes números; Khintchine 1928)
Si X_1, X_2, ... es una secuencia de variables aleatorias independientes e idénticamente distribuidas con media finita $E(X_i) = \mu$ (i.e. $E(|X_i|) < \infty$) entonces $\overline{X}_n = (X_1 + ... + X_n)/n$ converge en probabilidad a μ:

$$\overline{X}_n \xrightarrow{p} \mu$$

Es decir, para cualquier $\varepsilon > 0$:

$$\lim_{n\to\infty} P\left(\left|\overline{X}_n - \mu\right| < \varepsilon\right) = 1.$$

Ejemplo 3.1 (Aproximación de una probabilidad)
Simulamos una variable aleatoria X con distribución de Bernoulli de parámetro $p = 0.5$:

```
p <- 0.5
set.seed(1)
nsim <- 10000 # nsim <- 100
rx <- runif(nsim) < p # rbinom(nsim, size = 1, prob = p)
```

La aproximación por simulación de $E(X) = p$ será:

[1] *La ley fuerte* establece la convergencia casi segura.

```
mean(rx)
```

```
## [1] 0.5047
```

Podemos generar un gráfico con la evolución de la aproximación:

```
plot(cumsum(rx)/1:nsim, type = "l", lwd = 2, xlab = "Número de generaciones",
     ylab = "Proporción muestral", ylim = c(0, 1))
abline(h = mean(rx), lty = 2)
# valor teórico
abline(h = p)
```

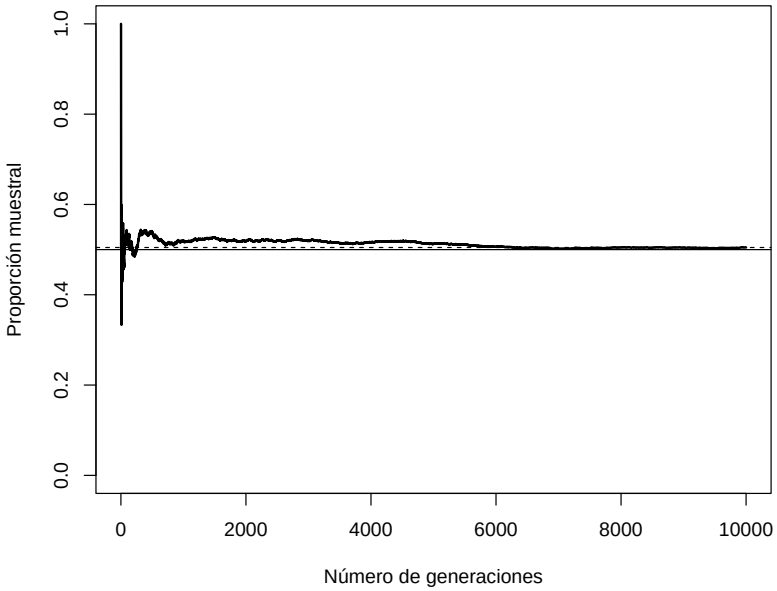


Figura 3.1: Aproximación de la proporción en función del número de generaciones.

3.1.1 Detección de problemas de convergencia

En la ley débil se requiere como condición suficiente que $E(|X_i|) < \infty$, en caso contrario la media muestral puede no converger a una constante. Un ejemplo conocido es la distribución de Cauchy:

```
set.seed(1)
nsim <- 10000
rx <- rcauchy(nsim) # rx <- rt(nsim, df = 2)
plot(cumsum(rx)/1:nsim, type = "l", lwd = 2,
     xlab = "Número de generaciones", ylab = "Media muestral")
```

Como conclusión, para detectar problemas de convergencia es especialmente recomendable representar la evolución de la aproximación de la característica de interés (sobre el número de generaciones), además de realizar otros análisis descriptivos de las simulaciones. Por ejemplo, en este caso podemos observar los valores que producen estos saltos mediante un gráfico de cajas:

```
boxplot(rx)
```

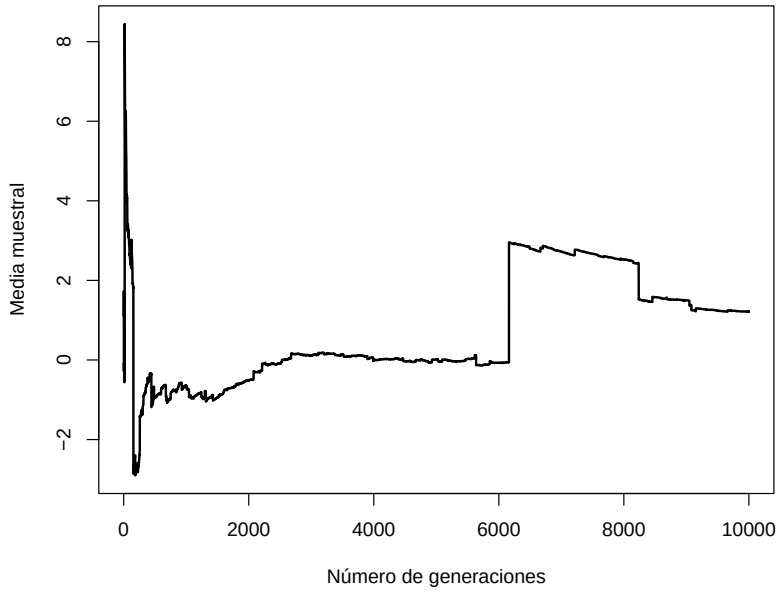


Figura 3.2: Evolución de la media muestral de una distribución de Cauchy en función del número de generaciones.

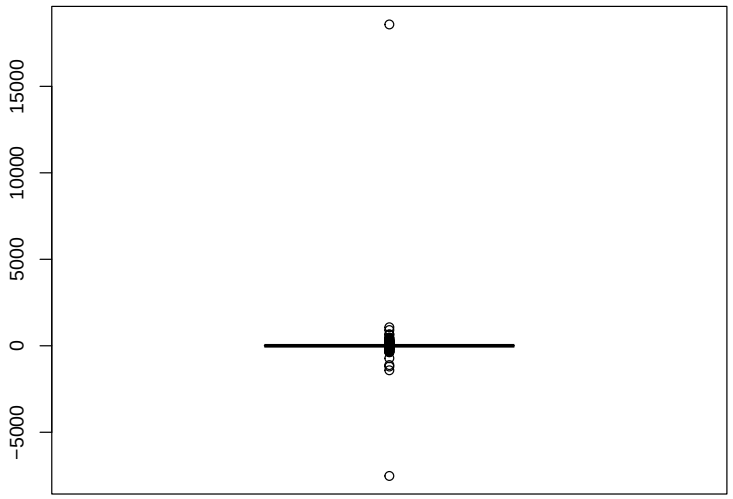


Figura 3.3: Gráfico de cajas de 10000 generaciones de una distribución de Cauchy.

3.1.2 Precisión

Una forma de medir la precisión de un estimador es utilizando su varianza, o también su desviación típica que recibe el nombre de error estándar. En el caso de la media muestral $\overline{X}_n$, suponiendo además

que $Var\left(X_i\right) = \sigma^2 < \infty$, un estimador insesgado de $Var\left(\overline{X}_n\right) = \sigma^2/n$ es:

$$\widehat{Var}\left(\overline{X}_n\right) = \frac{\widehat{S}_n^2}{n}$$

donde:

$$\widehat{S}_n^2 = \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i - \overline{X}\right)^2$$

es la cuasi-varianza muestral[2].

Los valores obtenidos servirían como medidas básicas de la precisión de la aproximación, aunque su principal aplicación es la construcción de intervalos de confianza. Si se endurecen las suposiciones de la ley débil de los grandes números (Teorema 3.1), exigiendo la existencia de varianza finita ($E\left(X_i^2\right) < \infty$), se dispone de un resultado más preciso sobre las variaciones de la aproximación por simulación en torno al límite teórico.

Teorema 3.2 (central del límite CLT)
Si X_1, X_2, ... es una secuencia de variables aleatorias independientes e idénticamente distribuidas con $E\left(X_i\right) = \mu$ y $Var\left(X_i\right) = \sigma^2 < \infty$, entonces la media muestral estandarizada converge en distribución a una normal estándar:

$$Z_n = \frac{\overline{X}_n - \mu}{\frac{\sigma}{\sqrt{n}}} \xrightarrow{d} N(0,1).$$

Es decir, $\lim\limits_{n\to\infty} F_{Z_n}(z) = \Phi(z)$.

Por tanto, un intervalo de confianza asintótico para μ es:

$$IC_{1-\alpha}(\mu) = \left(\overline{X}_n - z_{1-\alpha/2}\frac{\widehat{S}_n}{\sqrt{n}},\ \overline{X}_n + z_{1-\alpha/2}\frac{\widehat{S}_n}{\sqrt{n}}\right).$$

También podemos utilizar el error máximo (con nivel de confianza $1-\alpha$) de la estimación $z_{1-\alpha/2}\dfrac{\widehat{S}_n}{\sqrt{n}}$ como medida de su precisión.

El CLT es un resultado asintótico. En la práctica la convergencia es aleatoria, ya que depende de la muestra simulada (las generaciones pseudoaleatorias). Además, la convergencia puede considerarse lenta, en el sentido de que, por ejemplo, para doblar la precisión (disminuir el error a la mitad), necesitaríamos un número de generaciones cuatro veces mayor (Ver Sección 3.2). Pero una ventaja es que este error no depende del número de dimensiones (en el caso multidimensional puede ser mucho más rápida que otras alternativas numéricas; ver Apéndice B).

Ejemplo 3.2 (Aproximación de la media de una distribución normal)
Como ejemplo simulamos valores de una normal estándar:

---

[2]Esto sería también válido para el caso de una proporción, donde $E(X) = p$, $Var(X) = p(1-p)$ y $\hat{p}_n = \overline{X}_n$, obteniéndose que:

$$\widehat{Var}\left(\hat{p}_n\right) = \frac{\widehat{S}_n^2}{n} = \frac{\hat{p}_n(1-\hat{p}_n)}{n-1},$$

aunque lo más habitual es emplear:

$$\frac{S_n^2}{n} = \frac{\hat{p}_n(1-\hat{p}_n)}{n},$$

donde S_n^2 es la varianza muestral. Hay que tener en cuenta que en simulación el número de generaciones es normalmente grande y en la práctica no va haber diferencias apreciables.

```
xsd <- 1
xmed <- 0
set.seed(1)
nsim <- 1000
rx <- rnorm(nsim, xmed, xsd)
```

La aproximación por simulación de la media será:

```
mean(rx)
```

```
## [1] -0.01164814
```

Como medida de la precisión de la aproximación podemos utilizar el error máximo:

```
2*sd(rx)/sqrt(nsim)
```

```
## [1] 0.06545382
```

(es habitual emplear 2 en lugar de 1.96, lo que se correspondería con $1 - \alpha = 0.9545$ en el caso de normalidad). Podemos añadir también los correspondientes intervalos de confianza al gráfico de convergencia:

```
n <- 1:nsim
est <- cumsum(rx)/n
# (cumsum(rx^2) - n*est^2)/(n-1) # Cuasi-varianzas muestrales
esterr <- sqrt((cumsum(rx^2)/n - est^2)/(n-1)) # Errores estándar de la media
plot(est, type = "l", lwd = 2, xlab = "Número de generaciones",
     ylab = "Media y rango de error", ylim = c(-1, 1))
abline(h = est[nsim], lty=2)
lines(est + 2*esterr, lty=3)
lines(est - 2*esterr, lty=3)
abline(h = xmed)
```

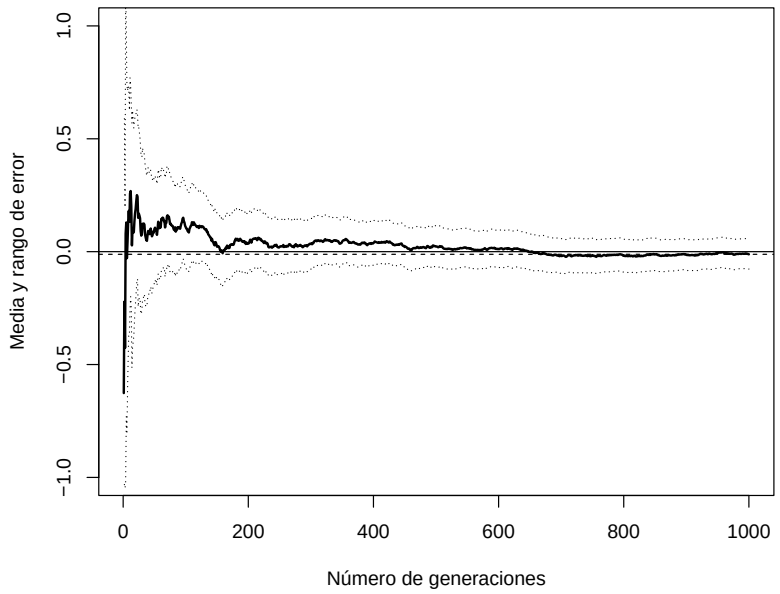


Figura 3.4: Gráfico de convergencia incluyendo el error de la aproximación.

Podemos generar este gráfico empleando la función `conv.plot()` (o `mc.plot()`) del paquete `simres` (fichero *mc.plot.R*).

3.2 Determinación del número de generaciones

Lo más habitual es seleccionar un valor de n del orden de varias centenas o millares. En los casos en los que la simulación se utiliza para aproximar una característica central de la distribución (como una media) puede bastar un número de generaciones del orden de $n = 100, 200, 500$. Sin embargo, en otros casos, por ejemplo para aproximar el p-valor de un contraste de hipótesis o construir intervalos de confianza, pueden ser necesarios valores del tipo $n = 1000, 2000, 5000, 10000$.

En muchas ocasiones puede interesar obtener una aproximación con un nivel de precisión fijado. Para una precisión absoluta ε, se trata de determinar n de forma que:

$$z_{1-\alpha/2}\frac{\widehat{S}_n}{\sqrt{n}} < \varepsilon$$

Un algoritmo (para un lenguaje de programación vectorial como R) podría ser el siguiente:

1. Hacer $j = 0$ y fijar un tamaño inicial n_0 (e.g. 60 ó 100, dependiendo del tiempo de computación requerido).
2. Generar $\{X_i\}_{i=1}^{n_0}$ y calcular $\overline{X}_{n_0}$ y $\widehat{S}_{n_0}$.
3. Mientras $z_{1-\alpha/2}\widehat{S}_{n_j}\Big/\sqrt{n_j} > \varepsilon$ hacer:

   3.1. $j = j + 1$.

   3.2. $n_j = \left\lceil \left(z_{1-\alpha/2}\widehat{S}_{n_{j-1}}\Big/\varepsilon \right)^2 \right\rceil$.

   3.3. Generar $\{X_i\}_{i=n_{j-1}+1}^{n_j}$ y calcular $\overline{X}_{n_j}$ y $\widehat{S}_{n_j}$.
4. Devolver $\overline{X}_{n_j}$ y $z_{1-\alpha/2}\widehat{S}_{n_j}\Big/\sqrt{n_j}$.

Para una precisión relativa $\varepsilon\,|\mu|$ se procede análogamente de forma que:

$$z_{1-\alpha/2}\frac{\widehat{S}_n}{\sqrt{n}} < \varepsilon\left|\overline{X}_n\right|.$$

3.3 El problema de la dependencia

En el caso de dependencia, bajo condiciones muy generales se verifica la ley débil de los grandes números. Sin embargo, la estimación de la precisión se complica:

$$Var\left(\overline{X}\right) = \frac{1}{n^2}\left(\sum_{i=1}^{n} Var\left(X_i\right) + 2\sum_{i<j} Cov\left(X_i, X_j\right)\right).$$

Ejemplo 3.3 (aproximación de una proporción bajo dependencia cadena de Markov)

Supongamos que en A Coruña llueve de media uno de cada tres días al año, y que la probabilidad de que un día llueva solo depende de lo que ocurrió el día anterior, siendo 0.94 si el día anterior llovió y 0.03 si no.

Podemos generar valores de la variable indicadora de día lluvioso con el siguiente código:

```
# Variable dicotómica 0/1 (FALSE/TRUE)
set.seed(1)
nsim <- 10000
alpha <- 0.03 # prob de cambio si seco
beta <- 0.06  # prob de cambio si lluvia
rx <- logical(nsim) # x == "llueve"
rx[1] <- FALSE # El primer día no llueve
for (i in 2:nsim)
  rx[i] <- if (rx[i-1]) runif(1) > beta else runif(1) < alpha
```

Si generamos el gráfico de convergencia asumiendo independencia:

```
n <- 1:nsim
est <- cumsum(rx)/n
esterr <- sqrt(est*(1-est)/(n-1)) # OJO! Supone independencia
plot(est, type="l", lwd=2, ylab="Probabilidad",
     xlab="Número de simulaciones", ylim=c(0,0.6))
abline(h = est[nsim], lty=2)
lines(est + 2*esterr, lty=2)
lines(est - 2*esterr, lty=2)
abline(h = 1/3, col="darkgray") # Probabilidad teórica
```

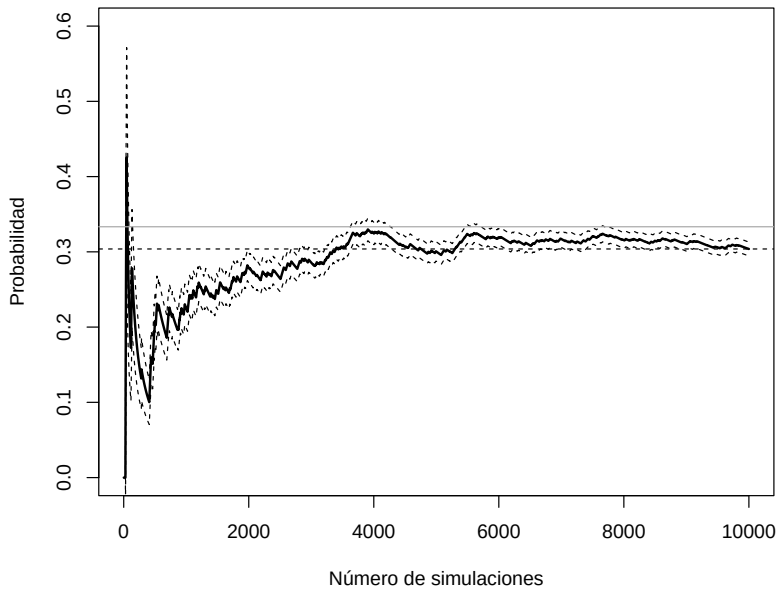


Figura 3.5: Gráfico de convergencia incluyendo el error de la aproximación (calculado asumiendo independencia).

La probabilidad teórica, obtenida empleando resultados de cadenas de Markov, es $p = 1/3$ y la aproximación de la proporción sería correcta (es consistente):

```
est[nsim]
```

```
## [1] 0.3038
```

Sin embargo, al ser datos dependientes, la aproximación anterior del error estándar no es adecuada:

```
esterr[nsim]
```

```
## [1] 0.004599203
```

En este caso al haber dependencia positiva se produce una subestimación del verdadero error estándar. Podemos generar el gráfico de autocorrelaciones:

```
acf(as.numeric(rx))
```

El gráfico anterior sugiere que si solo tomamos 1 de cada 25 valores podría ser razonable asumir independencia.

```
lag <- 24
xlag <- c(rep(FALSE, lag), TRUE)
```

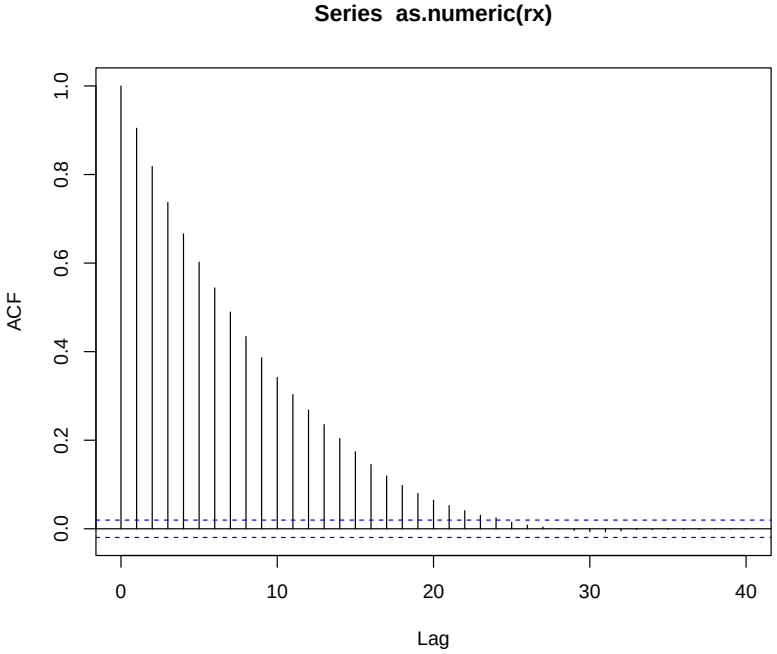


Figura 3.6: Correlograma de la secuencia indicadora de días de lluvia.

```
rxi <- rx[xlag]
acf(as.numeric(rxi))
```

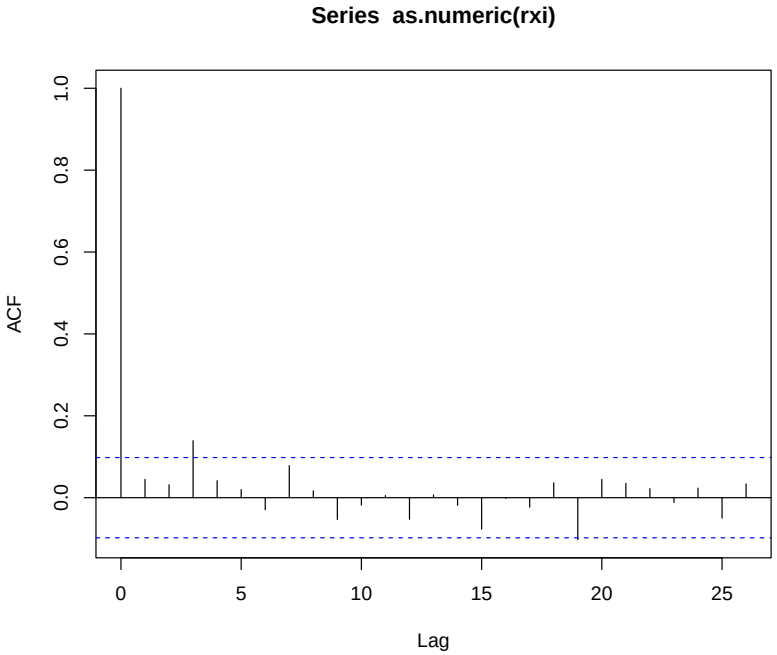


Figura 3.7: Correlograma de la subsecuencia de días de lluvia obtenida seleccionando uno de cada 25 valores.

```
nrxi <- length(rxi)
n <- 1:nrxi
est <- cumsum(rxi)/n
esterr <- sqrt(est*(1-est)/(n-1))
plot(est, type="l", lwd=2, ylab="Probabilidad",
     xlab=paste("Número de simulaciones /", lag + 1), ylim=c(0,0.6))
abline(h = est[length(rxi)], lty=2)
lines(est + 2*esterr, lty=2) # Supone independencia
lines(est - 2*esterr, lty=2)
abline(h = 1/3, col="darkgray")     # Prob. teor. cadenas Markov
```

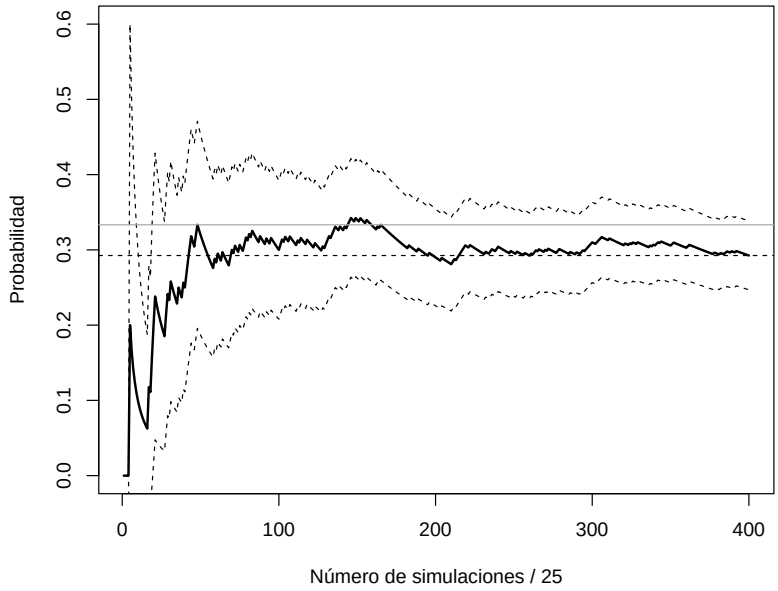


Figura 3.8: Gráfico de convergencia de la aproximación de la probabilidad a partir de la subsecuencia de días de lluvia (calculando el error de aproximación asumiendo independencia).

Esta forma de proceder podría ser adecuada para tratar de aproximar la precisión:

```
esterr[nrxi]
```

```
## [1] 0.02277402
```

pero no sería la más eficiente para aproximar la media. Siempre es preferible emplear todas las observaciones.

Por ejemplo, se podría pensar en considerar las medias de grupos de 25 valores consecutivos y suponer que hay independencia entre ellas:

```
rxm <- rowMeans(matrix(rx, ncol = lag + 1, byrow = TRUE))
nrxm <- length(rxm)
n <- 1:nrxm
est <- cumsum(rxm)/n
esterr <- sqrt((cumsum(rxm^2)/n - est^2)/(n-1)) # Errores estándar
plot(est, type="l", lwd=2, ylab="Probabilidad",
     xlab=paste("Número de simulaciones /", lag + 1), ylim=c(0,0.6))
abline(h = est[length(rxm)], lty=2)
lines(est + 2*esterr, lty=2) # OJO! Supone independencia
```

```
lines(est - 2*esterr, lty=2)
abline(h = 1/3, col="darkgray")      # Prob. teor. cadenas Markov
```

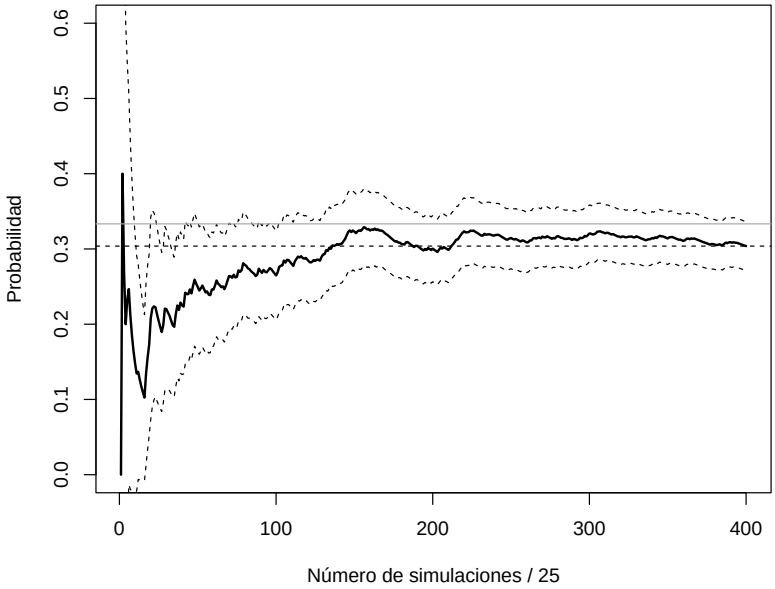


Figura 3.9: Gráfico de convergencia de las medias por lotes.

Esta es la idea del método de medias por lotes (*batch means*; *macro-micro replicaciones*) para la estimación de la precisión. Supongamos que la correlación entre X_i y X_{i+k} es aproximadamente nula, y consideramos las subsecuencias (lotes) $(X_{t+1}, X_{t+2}, \dots, X_{t+k})$ con $t = (j-1)k$, $j = 1, \dots, m$ y $n = mk$. Entonces:

$$Var\left(\bar{X}\right) = Var\left(\frac{1}{n}\sum_{i=1}^{n} X_i\right) = Var\left(\frac{1}{m}\sum_{j=1}^{m}\left(\frac{1}{k}\sum_{t=(i-1)k+1}^{ik} X_t\right)\right)$$
$$\approx \frac{1}{m^2}\sum_{j=1}^{m} Var\left(\frac{1}{k}\sum_{t=(i-1)k+1}^{ik} X_t\right) = \frac{1}{m}Var\left(\bar{X}_k\right)$$

donde $\bar{X}_k$ es la media de una subsecuencia de longitud k.

Alternativamente se podría recurrir a la generación de múltiples secuencias independientes entre sí:

```
# Variable dicotómica 0/1 (FALSE/TRUE)
set.seed(1)
nsim <- 1000
nsec <- 10
alpha <- 0.03 # prob de cambio si seco
beta <- 0.06  # prob de cambio si lluvia
rxm <- matrix(FALSE, nrow = nsec, ncol= nsim)
for (i in 1:nsec) {
  # rxm[i, 1] <- FALSE # El primer día no llueve
  # rxm[i, 1] <- runif(1) < 1/2 # El primer día llueve con probabilidad 1/2
  rxm[i, 1] <- runif(1) < 1/3 # El primer día llueve con probabilidad 1/3 (ideal)
  for (j in 2:nsim)
    rxm[i, j] <- if (rxm[i, j-1]) runif(1) > beta else runif(1) < alpha
```

```
}
```

La idea sería considerar las medias de las series como una muestra independiente de una nueva variable y estimar su varianza de la forma habitual:

```
# Media de cada secuencia
n <- 1:nsim
est <- apply(rxm, 1, function(x) cumsum(x)/n)
matplot(n, est, type = 'l', lty = 3, col = "lightgray",
     ylab="Probabilidad", xlab="Número de simulaciones")
# Aproximación
mest <- apply(est, 1, mean)
lines(mest, lwd = 2)
abline(h = mest[nsim], lty = 2)
# Precisión
mesterr <- apply(est, 1, sd)/sqrt(nsec)
lines(mest + 2*mesterr, lty = 2)
lines(mest - 2*mesterr, lty = 2)
# Prob. teor. cadenas Markov
abline(h = 1/3, col="darkgray")
```

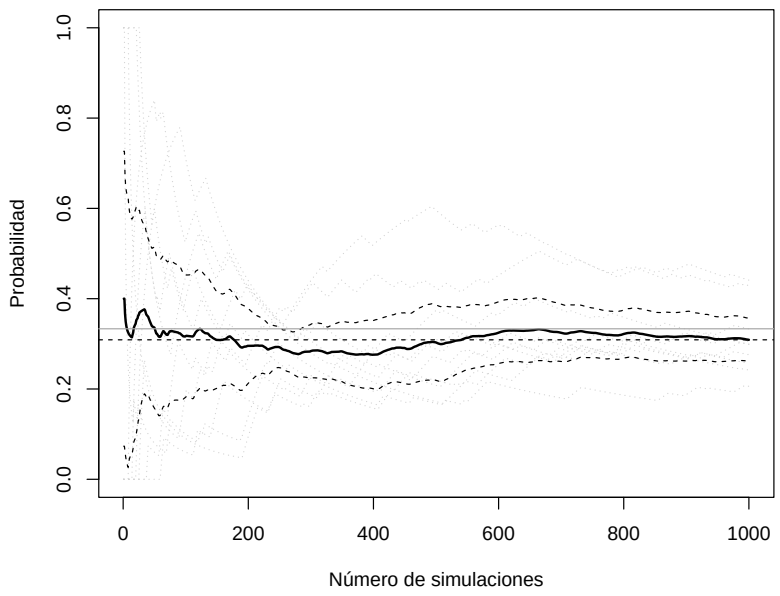


Figura 3.10: Gráfico de convergencia de la media de 10 secuencias generadas de forma independiente.

```
# Aproximación final
mest[nsim] # mean(rxm)
```

```
## [1] 0.3089
```

```
# Error estándar
mesterr[nsim]
```

```
## [1] 0.02403491
```

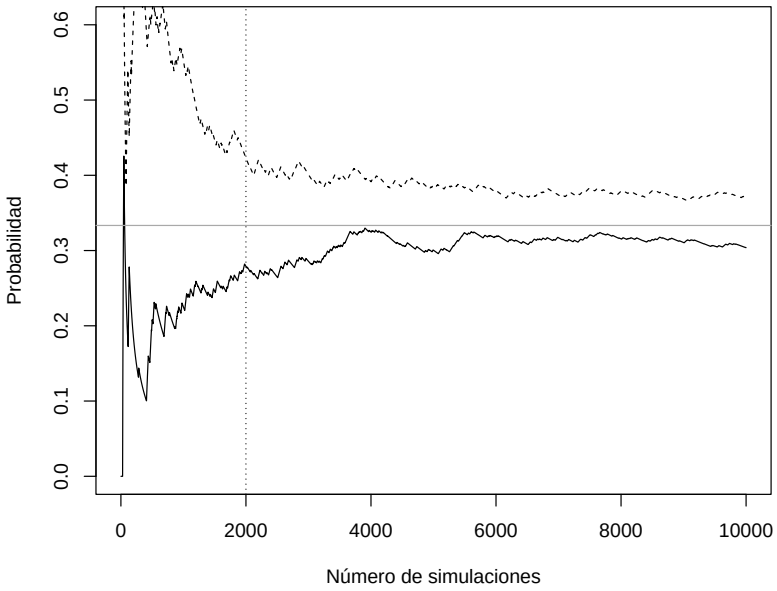
Trataremos este tipo de problemas en la diagnosis de algoritmos Monte Carlo de Cadenas de Markov (MCMC, Capítulo 10). Aparecen también en la simulación dinámica (por eventos o cuantos).

3.3.1 Periodo de calentamiento

En el caso de simulación de datos dependientes (simulación dinámica) pueden aparecer problemas de estabilización. Puede ocurrir que el sistema evolucione lentamente en el tiempo hasta alcanzar su distribución estacionaria, siendo muy sensible a las condiciones iniciales con las que se comienzó la simulación. En tal caso resulta conveniente ignorar los resultados obtenidos durante un cierto período inicial de tiempo (denominado período de calentamiento o estabilización), cuyo único objeto es conseguir que se estabilice la distribución de probabilidad.

Como ejemplo comparamos la simulación del Ejemplo 3.3 con la obtenida considerando como punto de partida un día lluvioso (con una semilla distinta para evitar dependencia).

```
set.seed(2)
nsim <- 10000
rx2 <- logical(nsim)
rx2[1] <- TRUE # El primer día llueve
for (i in 2:nsim)
  rx2[i] <- if (rx2[i-1]) runif(1) > beta else runif(1) < alpha
n <- 1:nsim
est <- cumsum(rx)/n
est2 <- cumsum(rx2)/n
plot(est, type="l", ylab="Probabilidad",
     xlab="Número de simulaciones", ylim=c(0,0.6))
lines(est2, lty = 2)
# Ejemplo periodo calentamiento nburn = 2000
abline(v = 2000, lty = 3)
# Prob. teor. cadenas Markov
abline(h = 1/3, col="darkgray")
```



En estos casos puede ser recomendable ignorar los primeros valores generados (por ejemplo los primeros 2000) y recalcular los estadísticos deseados.

También trataremos este tipo de problemas en la diagnosis de algoritmos MCMC.

Ejemplo 3.4 (simulación de un proceso autorregresivo serie de tiempo)

$$X_t = \mu + \rho * (X_{t-1} - \mu) + \varepsilon_t$$

Podemos tener en cuenta que en este caso la varianza es:

$$\mathrm{var}(X_t) = \mathrm{E}(X_t^2) - \mu^2 = \frac{\sigma_\varepsilon^2}{1-\rho^2}.$$

Establecemos los parámetros:

```
nsim <- 200    # Numero de simulaciones
xmed <- 0      # Media
rho <- 0.5     # Coeficiente AR
nburn <- 10    # Periodo de calentamiento (burn-in)
```

Se podría fijar la varianza del error:

```
evar <- 1
# Varianza de la respuesta
xvar <- evar / (1 - rho^2)
```

pero la recomendación sería fijar la varianza de la respuesta:

```
xvar <- 1
# Varianza del error
evar <- xvar*(1 - rho^2)
```

Para simular la serie, al ser un $AR(1)$, normalmente simularíamos el primer valor

```
x[1] <- rnorm(1, mean = xmed, sd = sqrt(xvar))
```

o lo fijamos a la media. Después generamos los siguientes valores de forma recursiva.

Como ejemplo nos alejamos un poco de la distribución estacionaria, para que el "periodo de calentamiento" sea mayor:

```
set.seed(1)
x <- numeric(nsim + nburn)
# Establecer el primer valor
x[1] <- -10
# Simular el resto de la secuencia
for (i in 2:length(x))
  x[i] <- xmed + rho*(x[i-1] - xmed) + rnorm(1, sd=sqrt(evar))
x <- as.ts(x)
plot(x)
abline(v = nburn, lty = 2)
```

y eliminamos el periodo de calentamiento:

```
rx <- x[-seq_len(nburn)]
```

Para simular una serie de tiempo en R se puede emplear la función `arima.sim` del paquete base `stats`. En este caso el periodo de calentamiento se establece mediante el parámetro `n.start` (que se fija automáticamente a un valor adecuado).

Por ejemplo, podemos generar este serie autoregresiva con:

```
rx2 <- arima.sim(list(order = c(1,0,0), ar = rho), n = nsim,
                 n.start = nburn, sd = sqrt(evar))
```

La recomendación es fijar la varianza de las series simuladas si se quieren comparar resultados considerando distintos parámetros de dependencia.

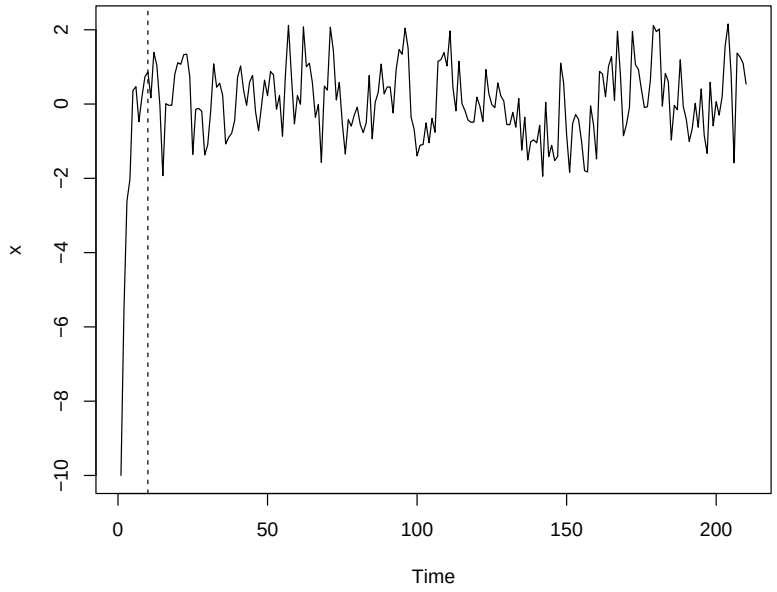


Figura 3.11: Ejemplo de una simulación de una serie de tiempo autorregresiva.

3.4 Observaciones

Como comentarios finales, podríamos añadir que:

- En el caso de que la característica de interés de la distribución de X no sea la media, los resultados anteriores no serían en principio aplicables.
- Incluso en el caso de la media, con el CLT obtenemos intervalos de confianza puntuales, que no habría que confundir con *bandas de confianza* (es muy probable que no contengan el verdadero valor del parámetro en todo el rango).
- En muchos casos (por ejemplo, cuando la generación de múltiples secuencias de simulación supone un coste computacional importante), puede ser preferible emplear un método de remuestreo para aproximar la precisión de la aproximación (ver Capítulo 8).

Capítulo 4

Simulación de variables continuas

En este capítulo se expondrán métodos generales para simular distribuciones continuas: el método de inversión (siguiente sección), los basados en aceptación-rechazo (secciones 4.2 y 4.3) y el método de composición (Sección 4.4). En todos los casos como punto de partida es necesario disponer de un método de generación de números pseudoaleatorios uniformes en $(0, 1)$.

4.1 Método de inversión

En general sería el método preferible para la simulación de una variable continua (siempre que se disponga de la función cuantil). Está basado en los siguientes resultados:

Si X es una variable aleatoria con función de distribución F continua y estrictamente monótona (invertible), entonces:

$$U = F(X) \sim \mathcal{U}(0, 1)$$

ya que:

$$G(u) = P(Y \leq u) = P(F(X) \leq u) = P\left(X \leq F^{-1}(u)\right) = F\left(F^{-1}(u)\right) = u$$

El recíproco también es cierto, si $U \sim \mathcal{U}(0, 1)$ entonces:

$$F^{-1}(U) \sim X$$

A partir de este resultado se deduce el siguiente algoritmo genérico para simular una variable continua con función de distribución F invertible:

Algoritmo 4.1 (Método de inversión)

1. Generar $U \sim \mathcal{U}(0, 1)$.
2. Devolver $X = F^{-1}(U)$.

Ejemplo 4.1 (simulación de una distribución exponencial)

La distribución exponencial $\exp(\lambda)$ de parámetro $\lambda > 0$ tiene como función de densidad $f(x) = \lambda e^{-\lambda x}$, si $x \geq 0$, y como función de distribución:

$$F(x) = \begin{cases} 1 - e^{-\lambda x} & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$$

Teniendo en cuenta que:

$$1 - e^{-\lambda x} = u \Leftrightarrow x = -\frac{\ln(1-u)}{\lambda}$$

el algoritmo para simular esta variable mediante el método de inversión es:

1. Generar $U \sim \mathcal{U}(0,1)$.

2. Devolver $X = -\dfrac{\ln(1-U)}{\lambda}$.

En el último paso podemos emplear directamente U en lugar de $1-U$, ya que $1-U \sim \mathcal{U}(0,1)$. Esta última expresión para acelerar los cálculos es la que denominaremos *forma simplificada*.

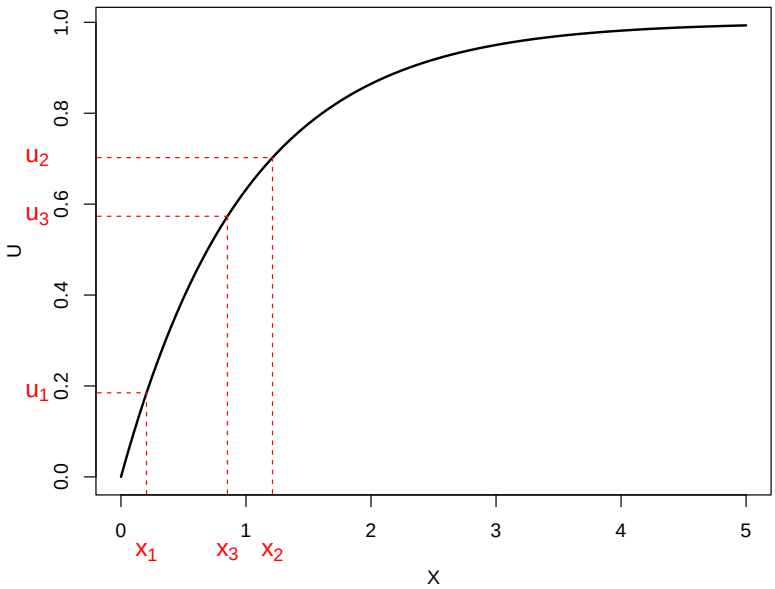


Figura 4.1: Ilustración de la simulación de una distribución exponencial por el método de inversión.

El código para implementar este algoritmo en R podría ser el siguiente:

```
tini <- proc.time()

lambda <- 2
nsim <- 10^5
set.seed(1)
u <- runif(nsim)
x <- -log(u)/lambda # -log(1-u)/lambda

tiempo <- proc.time() - tini
tiempo
```

```
##    user  system elapsed
##    0.02    0.00    0.01
```

```
hist(x, breaks = "FD", freq = FALSE,
        main = "", xlim = c(0, 5), ylim = c(0, 2.5))
curve(dexp(x, lambda), lwd = 2, add = TRUE)
```

Como se observa en la Figura 4.2 se trata de un método exacto (si está bien implementado) y la distribución de los valores generados se aproxima a la distribución teórica como cabría esperar con una muestra de ese tamaño.

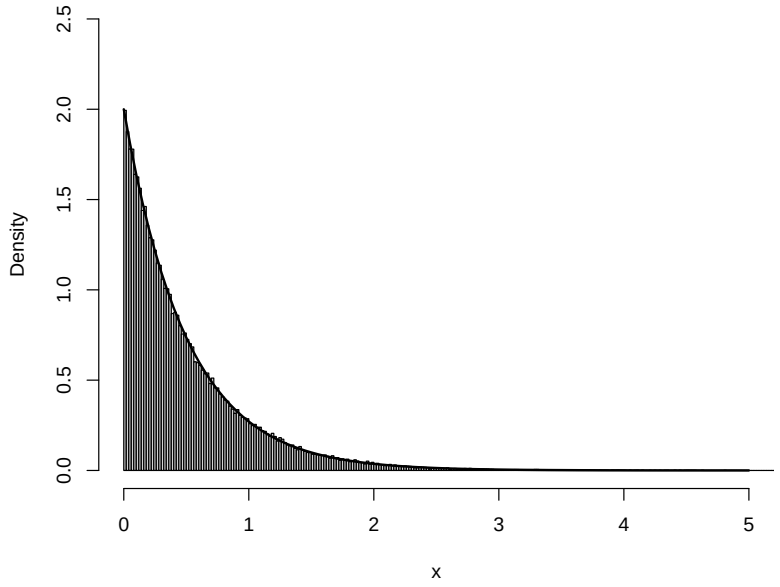


Figura 4.2: Distribución de los valores generados de una exponencial mediante el método de inversión.

4.1.1 Algunas distribuciones que pueden simularse por el método de inversión

A continuación se incluyen algunas distribuciones que se pueden simular fácilmente mediante el método de inversión. Se adjunta una forma simplificada del método que tiene por objeto evitar cálculos innecesarios (tal y como se hizo en el ejemplo de la exponencial).

Nombre	Densidad	$F(x)$	$F^{-1}(U)$	Forma simplificada
$\exp(\lambda)$ $(\lambda > 0)$	$\lambda e^{-\lambda x}$, si $x \geq 0$	$1 - e^{-\lambda x}$	$-\dfrac{\ln(1-U)}{\lambda}$	$-\dfrac{\ln U}{\lambda}$
Cauchy	$\dfrac{1}{\pi(1+x^2)}$	$\dfrac{1}{2} + \dfrac{\arctan x}{\pi}$	$\tan\left(\pi\left(U - \dfrac{1}{2}\right)\right)$	$\tan \pi U$
Triangular en $(0, a)$	$\dfrac{2}{a}\left(1 - \dfrac{x}{a}\right)$, si $0 \leq x \leq a$	$\dfrac{2}{a}\left(x - \dfrac{x^2}{2a}\right)$	$a\left(1 - \sqrt{1-U}\right)$	$a\left(1 - \sqrt{U}\right)$
Pareto $(a, b > 0)$	$\dfrac{ab^a}{x^{a+1}}$, si $x \geq b$	$1 - \left(\dfrac{b}{x}\right)^a$	$\dfrac{b}{(1-U)^{1/a}}$	$\dfrac{b}{U^{1/a}}$
Weibull $(\lambda, \alpha > 0)$	$\alpha\lambda^\alpha x^{\alpha-1} e^{-(\lambda x)^\alpha}$, si $x \geq 0$	$1 - e^{-(\lambda x)^\alpha}$	$\dfrac{(-\ln(1-U))^{1/\alpha}}{\lambda}$	$\dfrac{(-\ln U)^{1/\alpha}}{\lambda}$

Ejercicio 4.1 (distribución doble exponencial)

La distribución doble exponencial[1] (o distribución de Laplace) de parámetro λ tiene función de densidad:

$$f(x) = \frac{\lambda}{2} e^{-\lambda|x|},\ x \in \mathbb{R}$$

y función de distribución:

[1] Esta distribución también se puede generar fácilmente simulando una distribución exponencial y asignando un signo positivo o negativo con equiprobabilidad (ver Ejemplo 4.6) y función `simres::rdexp()` (fichero *ddexp.R*).

$$F(x) = \int_{-\infty}^{x} f(t)\,dt = \begin{cases} \frac{1}{2}e^{\lambda x} & \text{si } x < 0 \\ 1 - \frac{1}{2}e^{-\lambda x} & \text{si } x \geq 0 \end{cases}$$

a) Escribir una función que permita generar, por el método de inversión, una muestra de n observaciones de esta distribución.

```
ddexp <- function(x, lambda = 1){
# Densidad doble exponencial
  lambda*exp(-lambda*abs(x))/2
}

rdexp <- function(lambda = 1){
# Simulación por inversión
# Doble exponencial
  u <- runif(1)
  if (u<0.5) {
    return(log(2*u)/lambda)
  } else {
    return(-log(2*(1-u))/lambda)
  }
}

rdexpn <- function(n = 1000, lambda = 1) {
# Simulación n valores de doble exponencial
    x <- numeric(n)
    for(i in 1:n) x[i]<-rdexp(lambda)
    return(x)
}
```

b) Generar 10^4 valores de la distribución doble exponencial de parámetro $\lambda = 2$ y obtener el tiempo de CPU que tarda en generar la secuencia.

```
set.seed(54321)
system.time(x <- rdexpn(10^4, 2))
```

```
##    user  system elapsed
##    0.03    0.00    0.03
```

c) Representar el histograma y compararlo con la densidad teórica.

```
hist(x, breaks = "FD", freq = FALSE, main="")
# lines(density(x), col = 'blue')
curve(ddexp(x, 2), add = TRUE)
```

Como se trata de un método exacto de simulación, si está bien implementado, la distribución de los valores generados debería comportarse como una muestra genuina de la distribución objetivo.

4.1.2 Ventajas e inconvenientes

La principal ventaja de este método es que, en general, sería aplicable a cualquier distribución continua (como se muestra en la Sección 5.1, se puede extender al caso de que la función de distribución no sea invertible, incluyendo distribuciones discretas).

Uno de los principales problemas es que puede no ser posible encontrar una expresión explícita para $F^{-1}(u)$ (en ocasiones, como en el caso de la distribución normal, ni siquiera se dispone de una expresión explícita para la función de distribución). Además, aún disponiendo de una expresión explícita para $F^{-1}(u)$, su evaluación directa puede requerir de mucho tiempo de computación.

Como alternativa a estos inconvenientes se podrían emplear métodos numéricos para resolver $F(x) - u = 0$ de forma aproximada, aunque habría que resolver numéricamente esta ecuación para cada valor

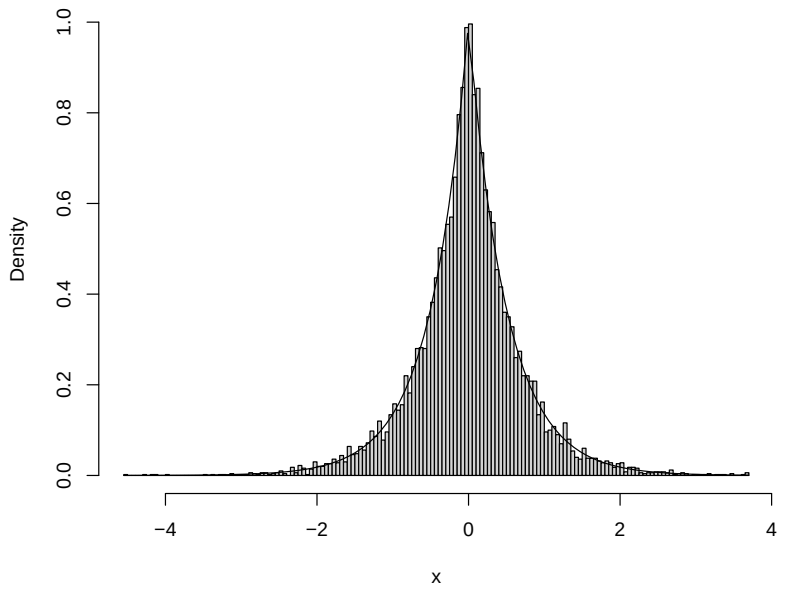


Figura 4.3: Distribución de los valores generados de una doble exponencial mediante el método de inversión.

aleatorio que se desea generar. Otra posibilidad, en principio preferible, sería emplear una aproximación a $F^{-1}(u)$, dando lugar al *método de inversión aproximada* (como se indicó en la Sección 1.3.1, R emplea por defecto este método para la generación de la distribución normal).

4.1.3 Inversión aproximada

En muchos casos en los que no se puede emplear la expresión exacta de la función cuantil $F^{-1}(u)$, se dispone de una aproximación suficientemente buena que se puede emplear en el algoritmo anterior (se obtendrían simulaciones con una distribución aproximada a la deseada).

Por ejemplo, para aproximar la función cuantil de la normal estándar, Odeh y Evans (1974) consideraron la siguiente función auxiliar[2]:

$$g(v) = \sqrt{-2\ln v}\frac{A\left(\sqrt{-2\ln v}\right)}{B\left(\sqrt{-2\ln v}\right)},$$

siendo $A(x) = \sum_{i=0}^{4} a_i x^i$ y $B(x) = \sum_{i=0}^{4} b_i x^i$ con:

$$\begin{array}{ll} a_0 = -0.322232431088 & b_0 = 0.0993484626060 \\ a_1 = -1 & b_1 = 0.588581570495 \\ a_2 = -0.342242088547 & b_2 = 0.531103462366 \\ a_3 = -0.0204231210245 & b_3 = 0.103537752850 \\ a_4 = -0.0000453642210148 & b_4 = 0.0038560700634 \end{array}$$

La aproximación consiste en utilizar $g(1-u)$ en lugar de $F^{-1}(u)$ para los valores de $u \in [10^{-20}, \frac{1}{2}]$ y $-g(u)$ si $u \in [\frac{1}{2}, 1-10^{-20}]$. Para $u \notin [10^{-20}, 1-10^{-20}]$ (que sólo ocurre con una probabilidad de $2 \cdot 10^{-20}$) la aproximación no es recomendable.

[2] R emplea una aproximación similar, basada en el algoritmo de Wichura (1988) más preciso, y que está implementado en el fichero fuente qnorm.c.

Algoritmo 4.2 (de Odeh y Evans)

1. Generar $U \sim U(0,1)$.
2. Si $U < 10^{-20}$ ó $U > 1 - 10^{-20}$ entonces volver a 1.
3. Si $U < 0.5$ entonces hacer $X = g(1-U)$ en caso contrario hacer $X = -g(U)$.
4. Devolver X.

En manuales de funciones matemáticas, como Abramowitz y Stegun (1964), se tienen aproximaciones de la función cuantil de las principales distribuciones (por ejemplo en la página 993 las correspondientes a la normal estándar).

4.2 Método de aceptación rechazo

Se trata de un método universal alternativo al de inversión para el caso de que no se pueda emplear la función cuantil, pero se dispone de una expresión (preferiblemente sencilla) para la función de densidad objetivo $f(x)$.

La idea es simular una variable aleatoria bidimensional (X, Y) con distribución uniforme en el hipografo de f (el conjunto de puntos del plano comprendidos entre el eje OX y f):

$$A_f = \left\{(x,y) \in \mathbb{R}^2 : 0 \leq y \leq f(x)\right\}.$$

De esta forma la primera componente tendrá la distribución deseada (Figura 4.4):

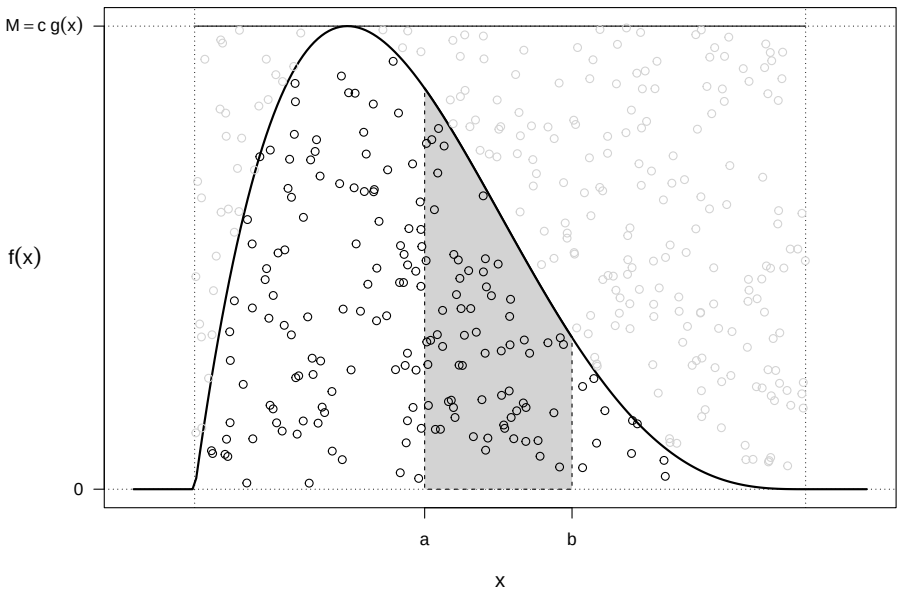


Figura 4.4: Puntos con distribución uniforme en el hipografo de una función de densidad.

$$P(a < X < b) = \frac{\text{Area de } \left\{(x,y) \in \mathbb{R}^2 : a < x < b;\ 0 \leq y \leq f(x)\right\}}{\text{Area de } A_f} = \int_a^b f(x)dx$$

El resultado anterior es también válido para una cuasi-densidad f^* (no depende de la constante normalizadora):

- Si $(X, Y) \sim \mathcal{U}\left(A_{f^*}\right)$ entonces[3] $X \sim f$.

[3]Emplearemos también $X \sim f$ para indicar que X es una variable aleatoria con función de densidad f.

Para simular una distribución uniforme en el hipografo A_f (o en A_{f^*}), lo que se hace es utilizar una variable aleatoria auxiliar T con función de densidad g, similar a f y fácil de simular, y una constante $c > 0$ verificando:

$$f(x) \leq c \cdot g(x), \ \forall x \in \mathbb{R}.$$

Podemos generar valores en $A_{cg} \supset A_f$ empleando el resultado:

- Si T es una variable aleatoria con función de densidad g y $U \sim \mathcal{U}(0,1)$ entonces

$$(T, c \cdot U \cdot g(x)) \sim \mathcal{U}\left(A_{cg}\right)$$

  siendo $A_{cg} = \left\{(x,y) \in \mathbb{R}^2 : 0 \leq y \leq cg\left(x\right)\right\}$.

Teniendo en cuenta además que:

- Si $(T,Y) \sim \mathcal{U}\left(A\right)$ y $B \subset A \Rightarrow (T,Y)|_B \sim \mathcal{U}\left(B\right)$.

Entonces, si (T,Y) sigue una distribución uniforme en A_{cg}, aceptando los valores de (T,Y) que pertenezcan a A_f (o a A_{f^*}) se obtendrán generaciones con distribución uniforme sobre A_f (o A_{f^*}) y la densidad de la primera componente será f.

4.2.1 Algoritmo

Supongamos que f es la densidad objetivo y g es una densidad auxiliar (fácil de simular y similar a f), de forma que existe una constante $c > 0$ tal que:

$$f(x) \leq c \cdot g(x), \ \forall x \in \mathbb{R},$$

(de donde se deduce que el soporte de g debe contener el de f).

Algoritmo 4.3 (Método de aceptación-rechazo; Von Neuman 1951)

1. Generar $U \sim \mathcal{U}(0,1)$.
2. Generar $T \sim g$.
3. Si $c \cdot U \cdot g\left(T\right) \leq f\left(T\right)$ devolver $X = T$,

   en caso contrario volver al paso 1.

4.2.2 Densidades acotadas en un intervalo cerrado

Sea f una función de densidad cualquiera con soporte en un intervalo cerrado $[a,b]$ (es decir, $\{x : f(x) > 0\} = [a,b]$) de tal forma que existe una constante $M > 0$ tal que $f(x) \leq M \ \forall x$ (es decir, f es acotada superiormente). En este caso puede tomarse como densidad auxiliar g, la de una $\mathcal{U}(a,b)$. En efecto, tomando $c = M\left(b-a\right)$ y teniendo en cuenta que

$$g(x) = \begin{cases} \frac{1}{b-a} & \text{si } x \in [a,b] \\ 0 & \text{en caso contrario} \end{cases}$$

se tiene que $f(x) \leq M = \frac{c}{b-a} = c \cdot g(x)$, $\forall x \in [a,b]$. Así pues, el algoritmo quedaría como sigue:

1. Generar $U, V \sim \mathcal{U}(0,1)$.
2. Hacer $T = a + \left(b-a\right)V$.
3. Si $M \cdot U \leq f\left(T\right)$ devolver $X = T$,

   en caso contrario volver al paso 1.

Nota: No confundir M con $c = M\left(b-a\right)$.

Ejemplo 4.2 (simulación de distribución beta a partir de la uniforme)
Para simular una variable con función de densidad $\mathcal{B}eta(\alpha, \beta)$:

$$f(x) = \frac{\Gamma(\alpha+\beta)}{\Gamma(\alpha)\Gamma(\beta)} x^{\alpha-1}(1-x)^{\beta-1} \text{ si } 0 \leq x \leq 1,$$

(siguiendo la notación de la función `dbeta(x, shape1, shape2)` de `R`), podemos considerar como distribución auxiliar una $\mathcal{U}(0,1)$, con $g(x) = 1$ si $0 \leq x \leq 1$.

Esta distribución está acotada y es unimodal, si α y β son mayores[4] que 1, y su moda es $\frac{\alpha-1}{\alpha+\beta-2}$, por lo que:

$$c = M = \max_{0 \leq x \leq 1} f(x) = f\left(\frac{\alpha-1}{\alpha+\beta-2}\right).$$

Por ejemplo, considerando $\alpha = 2$ y $\beta = 4$, si comparamos la densidad objetivo con la auxiliar reescalada (Figura 4.5), confirmamos que esta última está por encima (y llegan a tocarse, por lo que se está empleando la cota óptima; ver siguiente sección).

```
# densidad objetivo: dbeta
# densidad auxiliar: dunif
s1 <- 2
s2 <- 4
curve(dbeta(x, s1, s2), -0.1, 1.1, lwd = 2)
m <- dbeta((s1 - 1)/(s1 + s2 - 2), s1, s2)
# abline(h = m, lty = 2)
segments(0, m, 1, m, lty = 2, lwd = 2)
abline(v = 0, lty = 3)
abline(v = 1, lty = 3)
abline(h = 0, lty = 3)
```

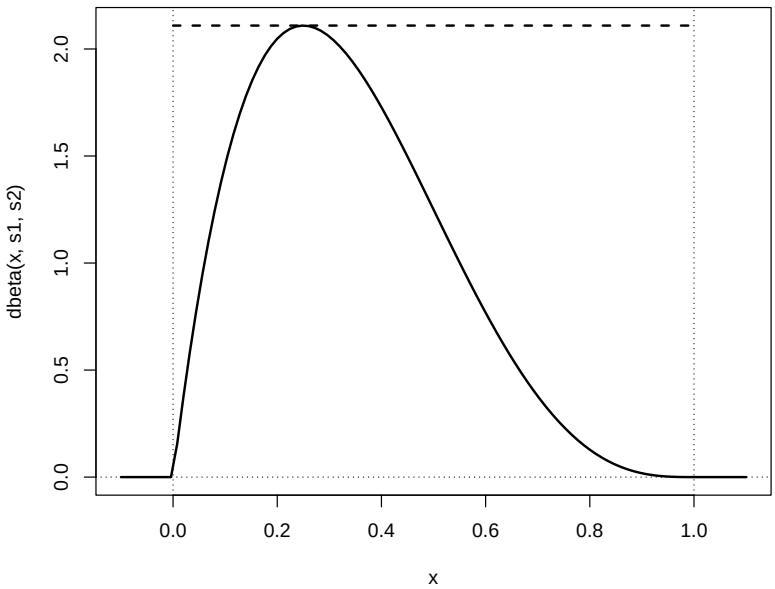


Figura 4.5: Densidad objetivo beta (línea continua) y densidad auxiliar unifome reescalada (línea discontinua).

[4]Si α o β son iguales a 1 puede simularse fácilmente por el método de inversión y si alguno es menor que 1 esta densidad no está acotada.

El siguiente código implementa el método de aceptación-rechazo para simular valores de la densidad objetivo (se incluye una variable "global" `ngen` para contar el número de generaciones de la distribución auxiliar):

```
ngen <- 0

rbeta2 <- function(s1 = 2, s2 = 2) {
  # Simulación por aceptación-rechazo
  # Beta a partir de uniforme
  m <- dbeta((s1 - 1)/(s1 + s2 - 2), s1, s2)
  while (TRUE) {
    u <- runif(1)
    x <- runif(1)
    ngen <<- ngen+1
    if (m*u <= dbeta(x, s1, s2)) return(x)
  }
}

rbeta2n <- function(n = 1000, s1 = 2, s2 = 2) {
  # Simulación n valores Beta(s1, s2)
  x <- numeric(n)
  for(i in 1:n) x[i]<-rbeta2(s1, s2)
  return(x)
}
```

Empleando estas funciones podemos generar una muestra de 10^3 observaciones de una $\mathcal{B}eta(2,4)$ (calculando de paso el tiempo de CPU):

```
set.seed(1)
nsim <- 1000
ngen <- 0
system.time(x <- rbeta2n(nsim, s1, s2))
```

```
##    user  system elapsed
##    0.03    0.00    0.03
```

Para analizar la eficiencia podemos emplear el número de generaciones de la distribución auxiliar (siguiente sección):

```
{cat("Número de generaciones = ", ngen)
cat("\nNúmero medio de generaciones = ", ngen/nsim)
cat("\nProporción de rechazos = ", 1-nsim/ngen, "\n")}
```

```
## Número de generaciones =  2121
## Número medio de generaciones =  2.121
## Proporción de rechazos =  0.5285243
```

Finalmente podemos representar la distribución de los valores generados y compararla con la densidad teórica:

```
hist(x, breaks = "FD", freq = FALSE, main = "")
curve(dbeta(x, s1, s2), col = 2, lwd = 2, add = TRUE)
```

Al ser un método exacto de simulación (si está bien implementado), la distribución de los valores generados debería comportarse como una muestra genuina de la distribución objetivo.

Ejercicio 4.2

Dar un algoritmo para simular la función de densidad dada por $f(x) = \frac{1}{16}(3x^2 + 2x + 2)$ si $0 \leq x \leq 2$, cero en otro caso. Estudiar su eficiencia.

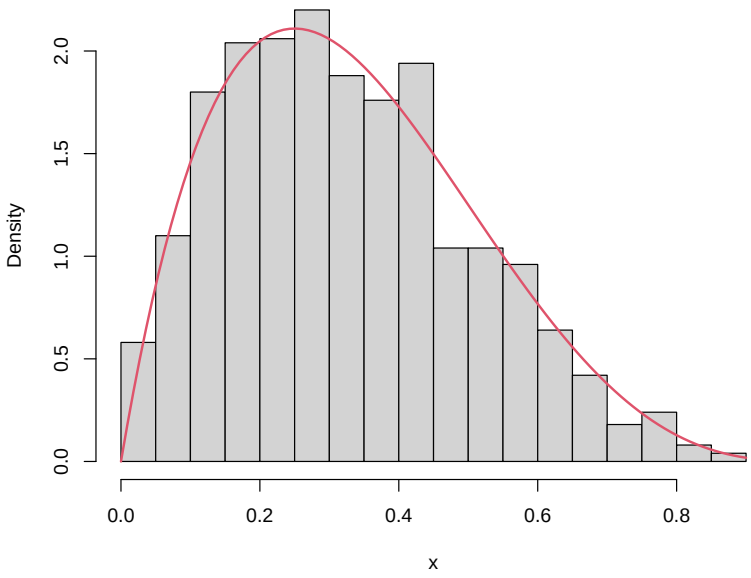


Figura 4.6: Distribución de los valores generados mediante el método de aceptación-rechazo.

4.2.3 Eficiencia del algoritmo

Como medida de la eficiencia del algoritmo de aceptación-rechazo podríamos considerar el número de iteraciones del algoritmo, es decir, el número de generaciones de la densidad auxiliar y de comparaciones para aceptar un valor de la densidad objetivo. Este número N es aleatorio y sigue una distribución geométrica (número de pruebas necesarias hasta obtener el primer éxito) con parámetro p (probabilidad de éxito) la probabilidad de aceptación en el paso 3:

$$p = \frac{\text{area}\left(A_f\right)}{\text{area}\left(A_{cg}\right)} = \frac{1}{c}.$$

Por tanto:

$$E\left(N\right) = \frac{1}{p} = c$$

es el número medio de iteraciones del algoritmo (el número medio de pares de variables (T, U) que se necesitan generar, y de comparaciones, para obtener una simulación de la densidad objetivo).

Es obvio, por tanto, que cuanto más cercano a 1 sea el valor de c más eficiente será el algoritmo (el caso de $c = 1$ se correspondería con $g = f$ y no tendría sentido emplear este método). Una vez fijada la densidad g, el valor óptimo será:

$$c_{\text{opt}} = \max_{\{x:g(x)>0\}} \frac{f(x)}{g(x)}.$$

Nota: Hay que tener en cuenta que la cota óptima es el número medio de iteraciones c solo si conocemos las constantes normalizadoras. Si solo se conoce la cuasidensidad f^* de la distribución objetivo (o de la auxiliar), la correspondiente cota óptima:

$$\tilde{c} = \max_{\{x:g(x)>0\}} \frac{f^*(x)}{g(x)}$$

asumirá la constante desconocida, aunque siempre podemos aproximar por simulación el verdadero valor de c y a partir de él la constante normalizadora (ver Ejercicio 4.3). Basta con tener en cuenta

que, si $f(x) = f^*(x)/k$:

$$\frac{1}{c} = \frac{\text{area}\left(A_{f^*}\right)}{\text{area}\left(A_{\tilde{c}g}\right)} = \frac{k}{\tilde{c}},$$

y por tanto $k = \tilde{c}/c$.

Ejemplo 4.3 (simulación de la normal a partir de la doble exponencial)
Se trata de simular la distribución normal estándar, con función de densidad:

$$f(x) = \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}},\ x \in \mathbb{R},$$

empleando el método de aceptación-rechazo considerando como distribución auxiliar una doble exponencial con $\lambda = 1$ (o distribución de Laplace):

$$g(x) = \frac{1}{2} e^{-|x|},\ x \in \mathbb{R}.$$

Esta distribución se utilizó en el Ejercicio 4.1, donde se definió la densidad auxiliar `ddexp(x, lambda)` y la función `rdexp(lambda)` para generar un valor aleatorio de esta distribución.

En este caso el soporte de ambas densidades es la recta real y el valor óptimo para c es:

$$c_{\text{opt}} = \max_{x\in\mathbb{R}} \frac{f(x)}{g(x)} = \max_{x\in\mathbb{R}} \frac{\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}}{\frac{1}{2} e^{-|x|}} = \sqrt{\frac{2}{\pi}} \max_{x\in\mathbb{R}} e^{\varphi(x)} = \sqrt{\frac{2}{\pi}} e^{\max_{x\in\mathbb{R}} \varphi(x)},$$

donde $\varphi(x) = -\frac{x^2}{2} + |x|$. Dado que esta función es simétrica, continua en toda la recta real y diferenciable tantas veces como se desee salvo en $x = 0$, bastará encontrar su máximo absoluto en el intervalo $[0, \infty]$:

$$x > 0 \Rightarrow \varphi'(x) = -x + 1, \varphi''(x) = -1;$$
$$\{x > 0, \varphi'(x) = 0\} \Leftrightarrow x = 1.$$

Por tanto, como $\varphi''(1) < 0$, φ alcanza un máximo relativo en $x = 1$ y otro de idéntico valor en $x = -1$. Resulta fácil demostrar que ambos son máximos absolutos (por los intervalos de crecimiento y decrecimiento de la función). Como consecuencia:

$$c_{\text{opt}} = \sqrt{\frac{2}{\pi}} e^{\varphi(1)} = \sqrt{\frac{2}{\pi}} e^{1/2} = \sqrt{\frac{2e}{\pi}} \approx 1.3155.$$

Si comparamos la densidad objetivo con la auxiliar reescalada con los parámetros óptimos (Figura 4.7), vemos que esta última está por encima, como debería ocurrir, pero llegan a tocarse (lo que validaría el cálculo para la obtención de la cota óptima).

```
# densidad objetivo: dnorm
# densidad auxiliar: ddexp
c.opt <- sqrt(2*exp(1)/pi)
lambda.opt <- 1
curve(c.opt * ddexp(x), xlim = c(-4, 4), lty = 2)
curve(dnorm, add = TRUE)
```

Alternativamente, en lugar de obtener la cota óptima de modo analítico, podríamos aproximarla numéricamente:

```
# NOTA: Cuidado con los límites
# optimize(f = function(x) dnorm(x)/ddexp(x), maximum = TRUE, interval = c(-0.5,0.5))
optimize(f = function(x) dnorm(x)/ddexp(x), maximum = TRUE, interval = c(0, 2))
```

```
## $maximum
## [1] 1
##
## $objective
## [1] 1.315489
```

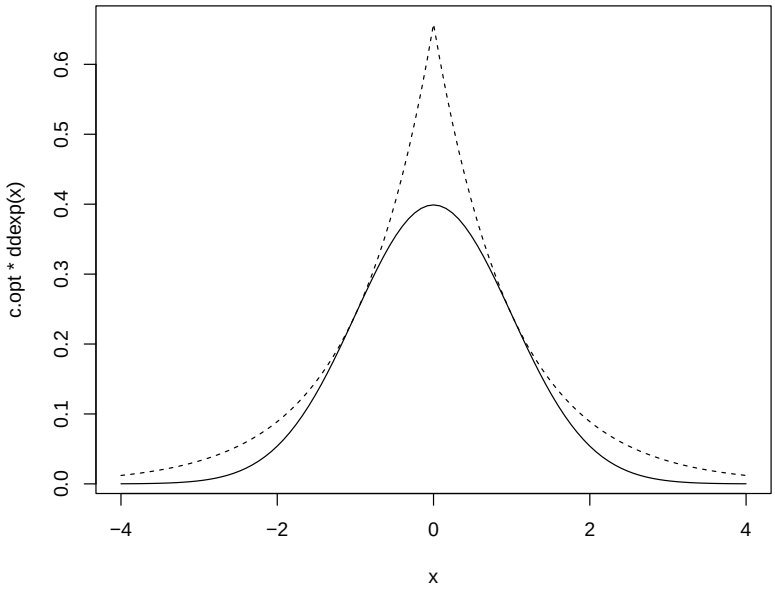


Figura 4.7: Densidad objetivo (normal estándar, línea continua) y densidad auxiliar (doble exponencial, línea discontinua) reescalada.

Vemos que la aproximación numérica coincide con el valor óptimo real $c_{\text{opt}} \approx 1.3154892$ (que se alcanza en $x = \pm 1$).

Para establecer la condición de aceptación o rechazo es recomendable emplear $c \cdot U \cdot g(T) \leq f(T)$. Aunque, en este caso concreto, se puede tener en cuenta que:

$$c \cdot U \cdot \frac{g(T)}{f(T)} = \sqrt{\frac{2e}{\pi}} U \sqrt{\frac{\pi}{2}} \exp\left(\frac{T^2}{2} - |T|\right) = U \cdot \exp\left(\frac{T^2}{2} - |T| + \frac{1}{2}\right).$$

Teniendo en cuenta los resultados anteriores, podríamos emplear el siguiente código para generar los valores de la densidad objetivo:

```
ngen <- 0

rnormAR <- function() {
# Simulación por aceptación-rechazo
# Normal estandar a partir de doble exponencial
  c.opt <- sqrt(2*exp(1)/pi)
  lambda.opt <- 1
  while (TRUE) {
    u <- runif(1)
    x <- rdexp(lambda.opt) # rdexpn(1, lambda.opt)
    ngen <<- ngen + 1 # Comentar esta línea para uso normal
    # if (u*exp((x^2+1)*0.5-abs(x)) <= 1) return(x)
    if (c.opt * u * ddexp(x, lambda.opt) <= dnorm(x)) return(x)
  }
}

rnormARn <- function(n = 1000) {
# Simulación n valores N(0,1)
    x <- numeric(n)
```

```
    for(i in 1:n) x[i] <- rnormAR()
    return(x)
}
```

Generamos una muestra de 10^4 observaciones:

```
set.seed(1)
nsim <- 10^4
ngen <- 0
system.time(x <- rnormARn(nsim))
```

```
##    user  system elapsed
##    0.11    0.00    0.11
```

Evaluamos la eficiencia:

```
{cat("Número de generaciones = ", ngen)
cat("\nNúmero medio de generaciones = ", ngen/nsim)
cat("\nProporción de rechazos = ", 1-nsim/ngen, "\n")}
```

```
## Número de generaciones =  13178
## Número medio de generaciones =  1.3178
## Proporción de rechazos =  0.2411595
```

Estos valores serían aproximaciones por simulación de los correspondientes valores teóricos (valor medio $c \approx 1.3155$ y probabilidad de rechazo $1 - p = 1 - 1/c \approx 0.23983$). A partir de ellos podríamos decir que el algoritmo es bastante eficiente.

Finalmente comparamos la distribución de los valores generados con la densidad teórica:

```
hist(x, breaks = "FD", freq = FALSE, main = "")
curve(dnorm, add = TRUE)
```

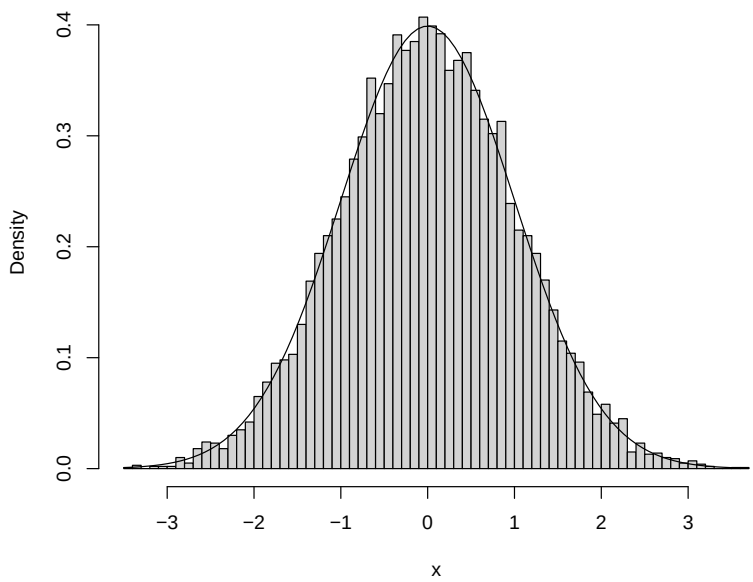


Figura 4.8: Distribución de los valores generados mediante el método de aceptación-rechazo.

Podemos observar que la distribución de los valores generados es la que cabría esperar de una muestra de tamaño `nsim` de la distribución objetivo (lo que nos ayudaría a confirmar que el algoritmo está

bien implementado, al ser un método exacto de simulación).

4.2.4 Elección de la densidad auxiliar

El principal problema con este método es encontrar una densidad auxiliar g de forma que c_{opt} sea próximo a 1. Una solución intermedia consiste en seleccionar una familia paramétrica de densidades $\{g_\theta : \theta \in \Theta\}$ entre las que haya alguna que se parezca bastante a f, encontrar el valor de c óptimo para cada densidad de esa familia:

$$c_\theta = \max_x \frac{f(x)}{g_\theta(x)}$$

y, finalmente, elegir el mejor valor θ_0 del parámetro, en el sentido de ofrecer el menor posible c_θ:

$$c_{\theta_0} = \min_{\theta\in\Theta} \max_x \frac{f(x)}{g_\theta(x)}.$$

Ejemplo 4.4 (simulación de la normal mediante la doble exponencial continuación)
Continuando con el Ejemplo 4.3 anterior sobre la simulación de una normal estándar mediante el método de aceptación-rechazo, en lugar de fijar la densidad auxiliar a una doble exponencial con $\lambda = 1$, consideraremos el caso general de $\lambda > 0$:

$$g_\lambda(x) = \frac{\lambda}{2} e^{-\lambda|x|},\ x \in \mathbb{R}.$$

Si pretendemos encontrar el mejor valor de λ, en términos de eficiencia del algoritmo, debemos calcular:

$$c_{\lambda_0} = \min_{\lambda>0} \max_{x\in\mathbb{R}} \frac{f(x)}{g_{\lambda(x)}} = \min_{\lambda>0} \max_{x\in\mathbb{R}} \frac{\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}}{\frac{\lambda}{2} e^{-\lambda|x|}}.$$

De forma totalmente análoga a la vista para el caso $\lambda = 1$, se tiene que:

$$c_\lambda = \max_{x\in\mathbb{R}} \frac{\frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}}}{\frac{\lambda}{2} e^{-\lambda|x|}} = \frac{1}{\lambda}\sqrt{\frac{2}{\pi}} \max_{x\in\mathbb{R}} e^{\varphi_{\lambda(x)}} = \frac{1}{\lambda}\sqrt{\frac{2}{\pi}} e^{\max_{x\in\mathbb{R}} \varphi_{\lambda(x)}},$$

donde $\varphi_\lambda(x) = -\frac{x^2}{2} + \lambda\,|x|$. De forma totalmente similar también puede probarse que φ_λ alcanza su máximo absoluto en los puntos $x = \pm\lambda$, siendo dicho valor máximo $\varphi_\lambda(\pm\lambda) = \frac{\lambda^2}{2}$. Como consecuencia:

$$c_\lambda = \frac{1}{\lambda}\sqrt{\frac{2}{\pi}} e^{\varphi_\lambda(\pm\lambda)} = \frac{e^{\frac{\lambda^2}{2}}}{\lambda}\sqrt{\frac{2}{\pi}}.$$

Finalmente debemos encontrar λ_0 tal que $c_{\lambda_0} = \min_{\lambda>0} c_\lambda$. Como:

$$\frac{\partial c_\lambda}{\partial \lambda} = \sqrt{\frac{2}{\pi}} \frac{e^{\frac{\lambda^2}{2}}\,(\lambda^2 - 1)}{\lambda^2},$$

entonces $\frac{\partial c_\lambda}{\partial \lambda} = 0 \Leftrightarrow \lambda = 1$, ya que $\lambda > 0$. Además:

$$\left.\frac{\partial^2 c_\lambda}{\partial \lambda^2}\right|_{\lambda=1} = \left.\sqrt{\frac{2}{\pi}} \frac{e^{\frac{\lambda^2}{2}}\,(\lambda^5 - \lambda^3 + 2\lambda)}{\lambda^4}\right|_{\lambda=1} = 2\sqrt{\frac{2e}{\pi}} > 0,$$

luego en $\lambda = 1$ se alcanza el mínimo.

```
curve(exp(x^2/2)/x*sqrt(2/pi), 0.1, 2.5,
      xlab = expression(lambda), ylab = expression(c[lambda]))
abline(v = 1, lty = 2)
```

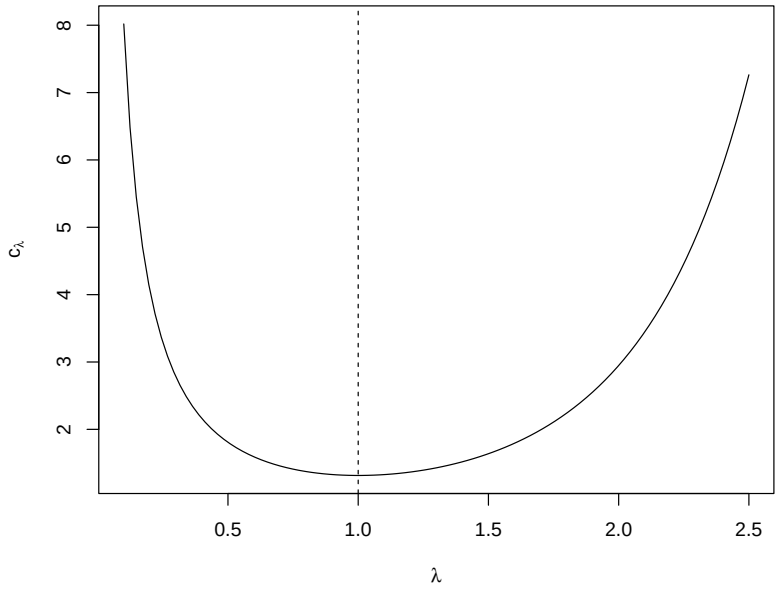


Figura 4.9: Representación de la cota óptima dependiedo del valor del parámetro.

De esto se deduce que la mejor densidad auxiliar doble exponencial es la correspondiente a $\lambda = 1$. Por tanto el algoritmo más eficiente, con esta familia de densidades auxiliares, es el expuesto en el Ejemplo 4.3.

Alternativamente también podríamos aproximar simultáneamente el parámetro óptimo y la cota óptima de la densidad auxiliar numéricamente:

```
# Obtención de valores c y lambda óptimos aproximados
fopt <- function(lambda) {
  # Obtiene c fijado lambda
  optimize(f = function(x) dnorm(x)/ddexp(x,lambda),
           maximum = TRUE, interval = c(0, 2))$objective
}

# Encontrar lambda que minimiza
res <- optimize(fopt, interval = c(0.5, 2))
lambda.opt2 <- res$minimum
c.opt2 <- res$objective
lambda.opt2
```

```
## [1] 0.9999987
```

```
c.opt2
```

```
## [1] 1.315489
```

4.2.5 Ejemplo: inferencia bayesiana

El algoritmo de aceptación-rechazo se emplea habitualmente en inferencia bayesiana. Denotando por:

- $f(x|\theta)$ la densidad muestral.
- $\pi(\theta)$ la densidad a priori.
- $\mathbf{x} = (x_1, ..., x_n)^\top$ la muestra observada.

El objetivo sería simular la distribución a posteriori de θ:

$$\pi(\theta|\mathbf{x}) = \frac{L(\mathbf{x}|\theta)\pi(\theta)}{\int L(\mathbf{x}|\theta)\pi(\theta)d\theta},$$

siendo $L(\mathbf{x}|\theta)$ la función de verosimilitud ($L(\mathbf{x}|\theta) = \prod_{i=1}^{n} f(x_i|\theta)$ suponiendo i.i.d.). Es decir:

$$\pi(\theta|\mathbf{x}) \propto L(\mathbf{x}|\theta)\pi(\theta).$$

Como esta distribución cambia al variar la muestra observada, puede resultar difícil encontrar una densidad auxiliar adecuada para simular valores de la densidad a posteriori $\pi(\theta|\mathbf{x})$. Por ejemplo, podríamos emplear la densidad a priori $\pi(\theta)$ como densidad auxiliar. Teniendo en cuenta que:

- $\pi(\theta|\mathbf{x})/\pi(\theta) \propto L(\mathbf{x}|\theta)$
- $L(\mathbf{x}|\theta) \leq \tilde{c} = L(\mathbf{x}|\hat{\theta})$ siendo $\hat{\theta}$ el estimador máximo verosímil de θ.

El algoritmo sería el siguiente:

1. Generar $U \sim \mathcal{U}(0,1)$.
2. Generar $\tilde{\theta} \sim \pi(\theta)$.
3. Si $L(\mathbf{x}|\hat{\theta}) \cdot U \leq L(\mathbf{x}|\tilde{\theta})$ devolver $\tilde{\theta}$,

   en caso contrario volver al paso 1.

Aunque, como se muestra en el siguiente ejercicio, esta elección de densidad auxiliar puede ser muy poco adecuada, siendo preferible en la práctica emplear un método adaptativo que construya la densidad auxiliar de forma automática (Sección 4.3.1).

Ejercicio 4.3 (Simulación de la distribución a posteriori a partir de la distribución a priori)
Para la estimación Bayes de la media de una normal se suele utilizar como distribución a priori una Cauchy.

a) Generar una muestra i.i.d. $X_i \sim N(\theta_0, 1)$ de tamaño $n = 10$ con $\theta_0 = 1$. Utilizar una $Cauchy(0,1)$ (`rcauchy()`) como distribución a priori y como densidad auxiliar para simular por aceptación-rechazo una muestra de la densidad a posteriori (emplear `dnorm()` para construir la verosimilitud). Obtener el intervalo de probabilidad/credibilidad al 95%.

```r
mu0 <- 1
n <- 10
nsim <- 10^4
set.seed(54321)
x <- rnorm(n, mean = mu0)

# Función de verosimilitud
# lik1 <- function(mu) prod(dnorm(x, mean = mu)) # escalar
lik <- Vectorize(function(mu) prod(dnorm(x, mean = mu))) # vectorial

# Cota óptima
# Estimación por máxima verosimilitud
emv <- optimize(f = lik, int = range(x), maximum = TRUE)
emv
```

```
## $maximum
## [1] 0.7353805
##
## $objective
## [1] 3.303574e-08
```

```
c <- emv$objective
```

En este caso concreto, ya sabríamos que el estimador máximo verosímil es la media muestral:

```
mean(x)
```

```
## [1] 0.7353958
```

y por tanto:

```
c <- lik(mean(x))
c
```

```
## [1] 3.303574e-08
```

```
# f.cuasi <- function(mu) sapply(mu, lik1)*dcauchy(mu)
f.cuasi <- function(mu) lik(mu)*dcauchy(mu)
curve(c * dcauchy(x), xlim = c(-4, 4), ylim = c(0, c/pi), lty = 2,
      xlab = "mu", ylab = "cuasidensidad")
curve(f.cuasi, add = TRUE)
```

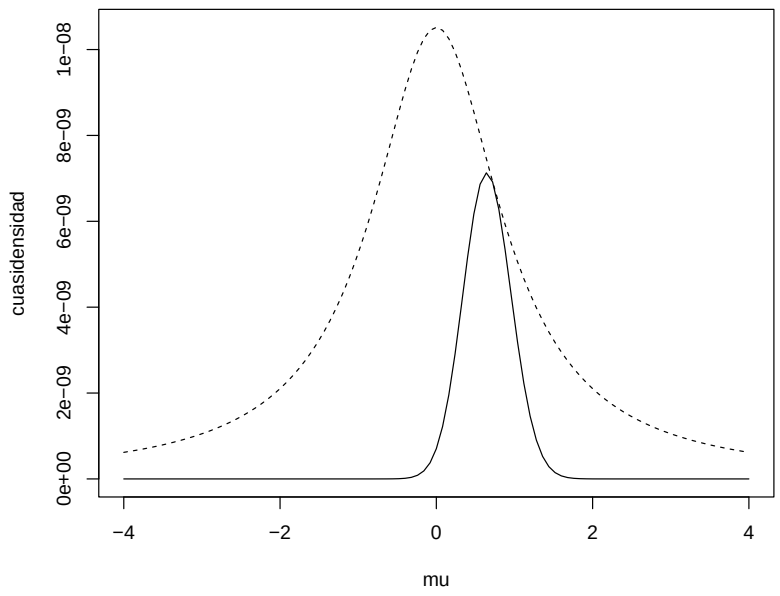


Figura 4.10: Comparación de la cuasidensidad a posteriori (línea contínua) con la densidad a priori reescalada (línea discontinua).

Por ejemplo, podríamos emplear el siguiente código para generar simulaciones de la distribución a posteriori mediante aceptación-rechazo a partir de la distribución de Cauchy:

```
ngen <- nsim
mu <- rcauchy(nsim)
ind <- c*runif(nsim) > lik(mu) # TRUE si no verifica condición
# Volver a generar si no verifica condición
while (sum(ind)>0){
  le <- sum(ind)
  ngen <- ngen + le
  mu[ind] <- rcauchy(le)
  ind[ind] <- c*runif(le) > lik(mu[ind]) # TRUE si no verifica condición
```

```
}

{ # Número generaciones
  cat("Número de generaciones = ", ngen)
  cat("\nNúmero medio de generaciones = ", ngen/nsim)
  cat("\nProporción de rechazos = ", 1-nsim/ngen,"\n")
}
```

```
## Número de generaciones =  59422
## Número medio de generaciones =  5.9422
## Proporción de rechazos =  0.8317122
```

A partir de la aproximación del número medio de generaciones podemos aproximar la constante normalizadora:

```
cte <- c*nsim/ngen
# integrate(f.cuasi, -Inf, Inf)
f.aprox <- function(mu) f.cuasi(mu)/cte
```

Finalmente, a partir de los valores generados podemos aproximar el intervalo de probabilidad al 95% (intervalo de credibilidad bayesiano):

```
q <- quantile(mu, c(0.025, 0.975))
q
```

```
##       2.5%      97.5%
## 0.05001092 1.26026227
```

```
# Representar estimador e IC Bayes
hist(mu, freq=FALSE, breaks = "FD", main="")
# abline(v = mean(x), lty = 3) # Estimación frecuentista
abline(v = mean(mu), lty = 2, lwd = 2)  # Estimación Bayesiana
abline(v = q, lty = 2)
curve(f.aprox, col = "blue", add = TRUE)
```

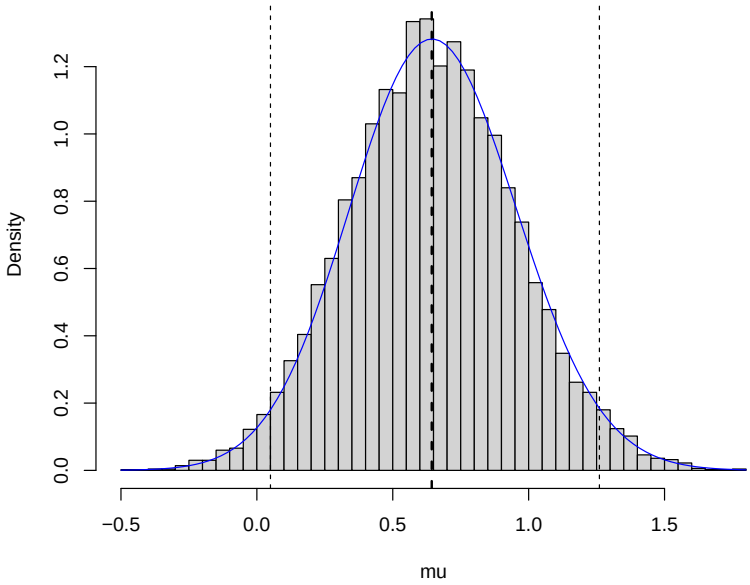


Figura 4.11: Distribución de los valores generados y aproximación del intervalo de credibilidad.

b) Repetir el apartado anterior con $n = 100$.

4.3 Modificaciones del método de aceptación-rechazo

En el tiempo de computación del algoritmo de aceptación-rechazo influye:

- La proporción de aceptación (debería ser grande).
- La dificultad de simular con la densidad auxiliar.
- El tiempo necesario para hacer la comparación en el paso 3.

En ciertos casos el tiempo de computación necesario para evaluar $f(x)$ puede ser alto. Para evitar evaluaciones de la densidad se puede emplear una función "squeeze" que aproxime la densidad por abajo (una envolvente inferior):

$$s(x) \leq f(x), \ \forall x \in \mathbb{R}.$$

Algoritmo 4.4 (Marsaglia 1977)

1. Generar $U \sim \mathcal{U}(0,1)$ y $T \sim g$.
2. Si $c \cdot U \cdot g(T) \leq s(T)$ devolver $X = T$,

   en caso contrario

   2.a. si $c \cdot U \cdot g(T) \leq f(T)$ devolver $X = T$,

   2.b. en caso contrario volver al paso 1.

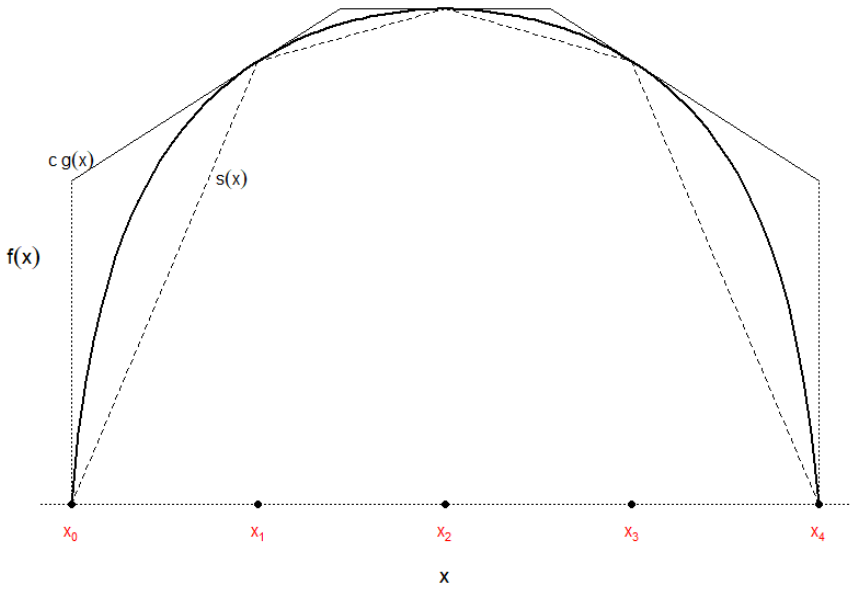


Figura 4.12: Ilustración del algoritmo de aceptación-rechazo con envolvente inferior (función "squeeze").

Cuanto mayor sea el área bajo $s(x)$ (más próxima a 1) más efectivo será el algoritmo.

Se han desarrollado métodos generales para la construcción de las funciones g y s de forma automática (cada vez que se evalúa la densidad se mejoran las aproximaciones). Estos métodos se basan principalmente en que una transformación de la densidad objetivo es cóncava o convexa.

4.3.1 Muestreo por rechazo adaptativo (ARS)

Supongamos que f es una cuasi-densidad log-cóncava (i.e. $\frac{\partial^2}{\partial x^2} \log f(x) < 0, \ \forall x$).

Sea $S_n = \{x_i : i = 0, \cdots, n+1\}$ con $f(x_i)$ conocidos.

Denotamos por $L_{i,i+1}(x)$ la recta pasando por $(x_i, \log f(x_i))$ y $(x_{i+1}, \log f(x_{i+1}))$

- $L_{i,i+1}(x) \leq \log f(x)$ en el intervalo $I_i = (x_i, x_{i+1}]$
- $L_{i,i+1}(x) \geq \log f(x)$ fuera de I_i

En el intervalo I_i se definen las envolventes de $\log f(x)$:

- $\underline{\phi}_n(x) = L_{i,i+1}(x)$
- $\overline{\phi}_n(x) = \min\left\{L_{i-1,i}(x), L_{i+1,i+2}(x)\right\}$

Las envolventes de $f(x)$ en I_i serán:

- $s_n(x) = \exp\left(\underline{\phi}_n(x)\right)$
- $G_n(x) = \exp\left(\overline{\phi}_n(x)\right)$

Tenemos entonces que:

$$s_n(x) \leq f(x) \leq G_n(x) = c \cdot g_n(x)$$

donde $g_n(x)$ es una mixtura discreta de distribuciones tipo exponencial truncadas (las tasas pueden ser negativas), que se puede simular fácilmente combinando el método de composición (Sección 4.4) con el método de inversión.

Algoritmo 4.5 (Gilks 1992)

1. Inicializar n y s_n.
2. Generar $U \sim \mathcal{U}(0,1)$ y $T \sim g_n$.
3. Si $U \cdot G_n(T) \leq s_n(T)$ devolver $X = T$,

   en caso contrario,

   3.a Si $U \cdot G_n(T) \leq f(T)$ devolver $X = T$.

   3.b Hacer $n = n+1$, añadir T a S_n y actualizar s_n y G_n.
4. Volver al paso 2.

Gilks y Wild (1992) propusieron una ligera modificación empleando tangentes para construir la cota superior, de esta forma se obtiene un método más eficiente pero requiere especificar la derivada de la densidad objetivo (ver Figura 4.12).

La mayoría de las densidades de la familia exponencial de distribuciones son log-cóncavas. Hörmann (1995) extendió esta aproximación al caso de densidades T_c-cóncavas:

$$T_c(x) = signo(c)x^c T_0(x) = \log(x).$$

Aparte de la transformación logarítmica, la transformación $T_{-1/2}(x) = -1/\sqrt{x}$ es habitualmente la más empleada.

4.3.2 Método del cociente de uniformes

Se puede ver como una modificación del método de aceptación-rechazo, de especial interés cuando el soporte no es acotado.

Si (U, V) se distribuye uniformemente sobre:

$$C_{f^*} = \left\{(u, v) \in \mathbb{R}^2 : 0 < u \leq \sqrt{f^*(v/u)}\right\},$$

siendo f^* una función no negativa integrable (cuasi-densidad), entonces $X = V/U$ tiene función de densidad proporcional a f^* (Kinderman y Monahan, 1977). Además C_{f^*} tiene área finita, por lo que pueden generarse fácilmente los valores (U, V) con distribución $\mathcal{U}\left(C_{f^*}\right)$ a partir de componentes uniformes unidimensionales (aceptando los puntos dentro de C_{f^*}).

De modo análogo al método de aceptación-rechazo, hay modificaciones para acelerar los cálculos y automatizar el proceso, construyendo regiones mediante polígonos:

$$C_i \subset C_{f^*} \subset C_s.$$

También se puede extender al caso multivariante y considerar transformaciones adicionales. Ver por ejemplo el paquete `rust`.

Ejemplo 4.5 (simulación de la distribución de Cauchy mediante cociente de uniformes) Si consideramos la distribución de Cauchy:

$$f(x) = \frac{1}{\pi(1 + x^2)},\ x \in \mathbb{R},$$

eliminando la constante por comodidad $f(x) \propto 1/(1 + x^2)$, se tiene que:

$$\begin{aligned} C_{f^*} &= \left\{(u, v) \in \mathbb{R}^2 : 0 < u \leq \frac{1}{\sqrt{1 + (v/u)^2}}\right\} \\ &= \left\{(u, v) \in \mathbb{R}^2 : u > 0, u^2 \leq \frac{u^2}{u^2 + v^2}\right\} \\ &= \left\{(u, v) \in \mathbb{R}^2 : u > 0, u^2 + v^2 \leq 1\right\}, \end{aligned}$$

dando como resultando que C_{f^*} es el semicírculo de radio uno, y podemos generar valores con distribución uniforme en esta región a partir de $\mathcal{U}\left([0, 1] \times [-1, 1]\right)$.

El correspondiente algoritmo está implementado en la función `rcauchy.rou()` del paquete `simres` (fichero *ar.R*):

```
simres::rcauchy.rou
```

```
## function(n) {
##   # Cauchy mediante cociente de uniformes
##   ngen <- n
##   u <- runif(n, 0, 1)
##   v <- runif(n, -1, 1)
##   x <- v/u
##   ind <- u^2 + v^2 > 1 # TRUE si no verifica condición
##   # Volver a generar si no verifica condición
##   while (le <- sum(ind)){ # mientras le = sum(ind) > 0
##     ngen <- ngen + le
##     u <- runif(le, 0, 1)
##     v <- runif(le, -1, 1)
##     x[ind] <- v/u
##     ind[ind] <- u^2 + v^2 > 1 # TRUE si no verifica condición
##   }
```

```
##   attr(x, "ngen") <- ngen
##   return(x)
## }
## <bytecode: 0x0000000020cac378>
## <environment: namespace:simres>
```

```
set.seed(1)
nsim <- 10^4
rx <- simres::rcauchy.rou(nsim)

hist(rx, breaks = "FD", freq = FALSE, main = "", xlim = c(-6, 6))
curve(dcauchy, add = TRUE)
```

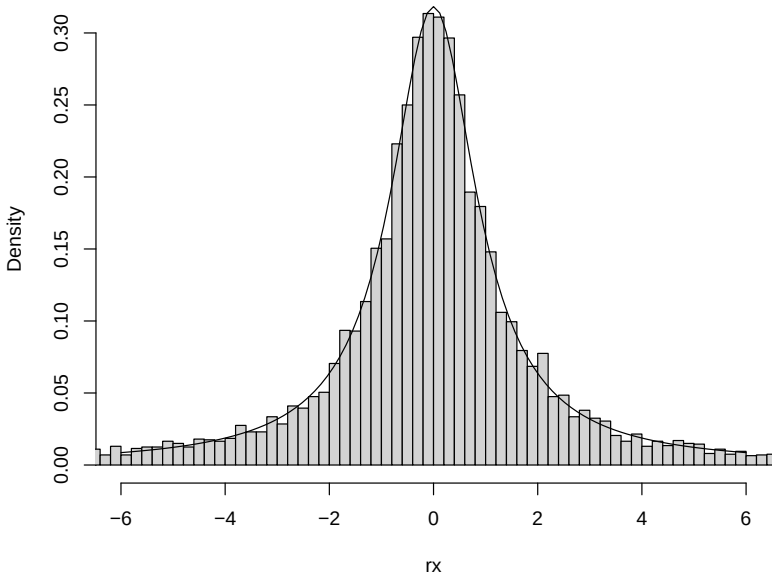


Figura 4.13: Distribución de los valores generados mediante el método de cociente de uniformes.

```
# Número generaciones
ngen <- attr(rx, "ngen")
{cat("Número de generaciones = ", ngen)
cat("\nNúmero medio de generaciones = ", ngen/nsim)
cat("\nProporción de rechazos = ", 1-nsim/ngen,"\n")}
```

```
## Número de generaciones =  12751
## Número medio de generaciones =  1.2751
## Proporción de rechazos =  0.2157478
```

4.4 Método de composición (o de simulación condicional)

En ocasiones la densidad de interés se puede expresar como una mixtura discreta de densidades:

$$f(x) = \sum_{j=1}^{k} p_j f_j(x)$$

con $\sum_{j=1}^{k} p_j = 1$, $p_j \geq 0$ y f_j densidades (sería también válido para funciones de distribución, incluyendo el caso discreto).

Algoritmo 4.6 (simulación de una mixtura discreta)

1. Generar J con distribución $P(J = j) = p_j$.
2. Generar $X \sim f_J$.

Ejemplo 4.6 (distribución doble exponencial)

A partir de la densidad de la distribución doble exponencial:

$$f(x) = \frac{\lambda}{2} e^{-\lambda|x|}, \ \forall x \in \mathbb{R},$$

se deduce que:

$$f(x) = \frac{1}{2} f_1(x) + \frac{1}{2} f_2(x)$$

siendo:

$$f_1(x) = \begin{cases} \lambda e^{-\lambda x} & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}, \ f_2(x) = \begin{cases} \lambda e^{\lambda x} & \text{si } x < 0 \\ 0 & \text{si } x \geq 0 \end{cases}$$

El algoritmo resultante sería el siguiente (empleando dos números pseudoaleatorios uniformes, el primero para seleccionar el índice y el segundo para generar un valor de la correspondiente componente mediante el método de inversión):

1. Generar $U, V \sim \mathcal{U}(0, 1)$.
2. Si $U < 0.5$ devolver $X = -\ln(1 - V)/\lambda$.
3. En caso contrario devolver $X = \ln(V)/\lambda$.

Este método está implementado en la función `rdexp()` del paquete `simres` (fichero *ddexp.R*).

Observaciones:

- En ocasiones se hace un reciclado de los números aleatorios (solo se genera una uniforme, e.g. $V = 2(U - 0.5)$ si $U \in (0.5, 1)$).
- En ciertas ocasiones por comodidad, para simular una muestra de tamaño n, se simulan muestras de tamaño np_i con densidad f_i y se combinan aleatoriamente.

Otro ejemplo de una mixtura discreta es el estimador tipo núcleo de la densidad (ver e.g. la ayuda de la función `density()` de R o la Sección 9.3). Simular a partir de una estimación de este tipo es lo que se conoce como *bootstrap suavizado*.

En el caso de una mixtura continua tendríamos:

$$f(x) = \int g(x|y)h(y)dy$$

Algoritmo 4.7 (simulación de una mixtura continua)

1. Generar $Y \sim h$.
2. Generar $X \sim g(\cdot|Y)$.

Este algoritmo es muy empleado en Inferencia Bayesiana y en la simulación de algunas variables discretas (como la Binomial Negativa, denominada también distribución Gamma-Poisson, o la distribución Beta-Binomial; ver Sección 5.7), ya que el resultado sería válido cambiando las funciones de densidad f y g por funciones de masa de probabilidad.

4.5 Métodos específicos para la generación de algunas distribuciones notables

En el pasado se ha realizado un esfuerzo considerable para desarrollar métodos eficientes para la simulación de las distribuciones de probabilidad más importantes. Estos algoritmos se describen en la mayoría de los libros clásicos de simulación (e.g. Cao, 2002, Capítulo 5)[5], principalmente porque resultaba necesario implementar estos métodos durante el desarrollo de software estadístico. Hoy en día estos algoritmos están disponibles en numerosas bibliotecas y no es necesario su implementación (por ejemplo, se puede recurrir a R o emplear su librería matemática disponible en https://svn.r-project.org/R/trunk/src/nmath). Sin embargo, además de que muchos de ellos servirían como ilustración de la aplicación de los métodos generales expuestos en secciones anteriores, pueden servir como punto de partida para la generación de otras distribuciones.

Entre los distintos métodos disponibles para la generación de las distribuciones continuas más conocidas podríamos destacar:

- Método de Box-Müller para la generación de normales independientes (que se puede generalizar para otras distribuciones o incorporar dependencia).
- Algoritmos de Jöhnk (1963) y Cheng (1978) para la generación de la distribución beta (como ejemplo de la eficiencia de los métodos de aceptación-rechazo).

4.5.1 Método de Box-Müller

Se basa en la siguiente propiedad. Dadas dos variables aleatorias independientes $E \sim \exp(1)$ y $U \sim \mathcal{U}(0,1)$, las variables $\sqrt{2E}\cos 2\pi U$ y $\sqrt{2E}\operatorname{sen} 2\pi U$ son $\mathcal{N}(0,1)$ independientes.

Algoritmo 4.8 (de Box-Müller 1958)

1. Generar $U, V \sim \mathcal{U}(0,1)$.
2. Hacer $W_1 = \sqrt{-2\ln U}$ y $W_2 = 2\pi V$.
3. Devolver $X_1 = W_1 \cos W_2$, $X_2 = W_1 \operatorname{sen} W_2$.

Podemos hacer que la función `rnorm()` de R emplee este algoritmo estableciendo el parámetro `normal.kind` a `"Box-Muller"` en una llamada previa a `set.seed()` o `RNGkind()`.

Este método está relacionado con el denominado *método FFT* (transformada de Fourier; e.g. Davies y Harte, 1987) para la generación de una normal multidimensional con dependencia, que resulta ser equivalente al *Circular embedding* (Dietrich and Newsam, 1997). La idea de estos métodos es que, considerando módulos exponenciales y fases uniformes generamos normales independientes, pero cambiando la varianza de los módulos (W_1) podemos inducir dependencia. Adicionalmente, cambiando la distribución de las fases (W_2) se generan distribuciones distintas de la normal.

4.5.2 Simulación de la distribución beta

Existen multitud de algoritmos para simular la distribución $\mathcal{B}eta(a,b)$. Probablemente, el más sencillo de todos es el que se obtiene a partir de la distribución gamma o de la chi-cuadrado, si se dispone de un algoritmo para generar estas distribuciones, empleando la definición habitual de la distribución beta:

Si $Y \sim \mathcal{G}amma(a,s)$ y $Z \sim \mathcal{G}amma(b,s)$ son independientes, entonces

$$X = \frac{Y}{Y+Z} \sim \mathcal{B}eta(a,b).$$

[5] Cuidado con la notación y la parametrización empleadas, puede variar entre referencias. Por ejemplo, en Cao (2002) la notación de la distribución Gamma es ligeramente distinta a la empleada en R y en el presente libro.

Como la distribución resultante no depende de s y $\chi^2_n \stackrel{d}{=} \mathcal{G}amma\left(\frac{n}{2}, \frac{1}{2}\right)$, se podría considerar $Y \sim \chi^2_{2a}$ y $Z \sim \chi^2_{2b}$ independientes.

También se podrían emplear resultado conocidos relacionados con esta distribución, como por ejemplo que la distribución del estadístico de orden k de una muestra de tamaño n de una distribución uniforme tiene una distribución beta:

$$U_{(k)} \sim \mathcal{B}eta(k, n+1-k).$$

El resultado es el algoritmo de Fox (1963), que podría ser adecuado para simular esta distribución cuando $a, b \in \mathbb{N}$ y son valores pequeños.

Algoritmo 4.9 (de Fox 1963)

1. Generar $U_1, U_2, \dots, U_{a+b-1} \sim \mathcal{U}(0,1)$.
2. Ordenar: $U_{(1)} \leq U_{(2)} \leq \cdots \leq U_{(a+b-1)}$.
3. Devolver $X = U_{(a)}$.

Es obvio que este algoritmo puede resultar muy lento si alguno de los dos parámetros es elevado (pues habrá que simular muchas uniformes para conseguir un valor simulado de la beta). Además, en función de cuál de los dos parámetros, a ó b, sea mayor, resultará más eficiente, en el paso 2, comenzar a ordenar por el mayor, luego el segundo mayor, y así sucesivamente, o hacerlo empezando por el menor. En cualquier caso, es obvio que no es necesario ordenar todos los valores U_i generados, sino tan sólo encontrar el que ocupa el lugar a-ésimo.

Un método válido aunque a ó b no sean enteros es el dado por el algoritmo de Jöhnk (1964).

Algoritmo 4.10 (de Jöhnk 1964)

1. Generar $U_1, U_2 \sim \mathcal{U}(0,1)$.
2. Hacer $V = U_1^{\frac{1}{a}}$, $W = U_2^{\frac{1}{b}}$ y $S = V + W$.
3. Si $S \leq 1$ devolver $X = \frac{V}{S}$,

   en caso contrario volver al paso 1.

El método resulta extremadamente ineficiente para a ó b mayores que 1. Esto es debido a que la condición $S \leq 1$ del paso 3 puede tardar muchísimo en verificarse. Por este motivo, el algoritmo de Jöhnk sólo es recomendable para $a < 1$ y $b < 1$. Como remedio a esto puede usarse el algoritmo de Cheng (1978) que es algo más complicado de implementar[6] pero mucho más eficiente.

Algoritmo 4.11 (de Cheng 1978)

Inicialización:

1. Hacer $\alpha = a + b$.
2. Si $\min(a,b) \leq 1$ entonces hacer $\beta = \frac{1}{\min(a,b)}$, en otro caso hacer $\beta = \sqrt{\frac{\alpha-2}{2pq-\alpha}}$.
3. Hacer $\gamma = a + \frac{1}{\beta}$.

Simulación:

1. Generar $U_1, U_2 \sim \mathcal{U}(0,1)$.

[6] R implementa este algoritmo en el fichero fuente rbeta.c.

2. Hacer $V = \beta \cdot \ln\left(\frac{U_1}{1-U_1}\right)$ y $W = a \cdot e^V$.
3. Si $\alpha \cdot \ln\left(\frac{\alpha}{b+W}\right) + \gamma V - \ln 4 \geq \ln\left(U_1^2 U_2\right)$ devolver $X = \frac{W}{b+W}$,
   en caso contrario volver al paso 1.

Capítulo 5

Simulación de variables discretas

Se trata de simular una variable aleatoria discreta X con función de masa de probabilidad:

x_i	x_1	x_2	$\cdots$	x_n	$\cdots$
$P(X = x_i)$	p_1	p_2	$\cdots$	p_n	$\cdots$

Considerando como partida una $\mathcal{U}(0,1)$, la idea general consiste en asociar a cada posible valor x_i de X un subintervalo de $(0,1)$ de longitud igual a la correspondiente probabilidad. Por ejemplo, como ya se mostró en capítulos anteriores, es habitual emplear código de la forma:

```
x <- runif(nsim) < p
```

para simular una distribución *Bernoulli*(p).

Para generar variables discretas con dominio finito en R, si no se dispone de un algoritmo específico más eficiente, es recomendable emplear:

```
sample(valores, nsim, replace = TRUE, prob)
```

Esta función del paquete base implementa eficientemente el método "alias" que describiremos más adelante en la Sección 5.3.

5.1 Método de la transformación cuantil

Este método es una adaptación del método de inversión (válido para el caso continuo) a distribuciones discretas, por lo que también se denomina *método de inversión generalizado*. En este caso, la función de distribución es:

$$F(x) = \sum_{x_j \leq x} p_j,$$

y la distribución de la variable aleatoria $F(X)$ no es uniforme (es una variable aleatoria discreta que toma los valores $F(x_i)$ con probabilidad p_i, $i = 1, 2, ...$). Sin embargo, se puede generalizar el método de inversión a situaciones en las que F no es invertible considerando la función cuantil.

Se define la función cuantil o inversa generalizada de una función de distribución F como:

$$Q(u) = \inf\{x \in \mathbb{R} : F(x) \geq u\}, \ \forall u \in (0,1).$$

Si F es invertible $Q = F^{-1}$.

Teorema 5.1 (de inversión generalizada)
Si $U \sim \mathcal{U}(0,1)$, la variable aleatoria $Q(U)$ tiene función de distribución F.

Demostración: Bastaría ver que:

$$Q(u) \leq x \Longleftrightarrow u \leq F(x).$$

Como F es monótona y por la definición de Q:

$$Q(u) \leq x \Rightarrow u \leq F(Q(u)) \leq F(x).$$

Por otro lado como Q también es monótona:

$$u \leq F(x) \Rightarrow Q(u) \leq Q(F(x)) \leq x$$

A partir de este resultado se deduce el siguiente algoritmo general para simular una distribución de probabilidad discreta.

Algoritmo 5.1 (de transformación cuantil)

1. Generar $U \sim \mathcal{U}(0,1)$.
2. Devolver $X = Q(U)$.

El principal problema es el cálculo de $Q(U)$. En este caso, suponiendo por comodidad que los valores que toma la variable están ordenados ($x_1 < x_2 < \cdots$), la función cuantil será:

$$\begin{aligned} Q(U) &= \inf\left\{x_j : \textstyle\sum_{i=1}^{j} p_i \geq U\right\} \\ &= x_k, \text{ tal que } \textstyle\sum_{i=1}^{k-1} p_i < U \leq \sum_{i=1}^{k} p_i \end{aligned}$$

Para encontrar este valor se puede emplear el siguiente algoritmo:

Algoritmo 5.2 (de transformación cuantil con búsqueda secuencial)

1. Generar $U \sim \mathcal{U}(0,1)$.
2. Hacer $I = 1$ y $S = p_1$.
3. Mientras $U > S$ hacer $I = I + 1$ y $S = S + p_I$
4. Devolver $X = x_I$.

Este algoritmo no es muy eficiente, especialmente si el número de posibles valores de la variable es grande.

Nota: El algoritmo anterior es válido independientemente de que los valores que toma la variable estén ordenados.

Si la variable toma un número finito de valores, se podría implementar en R de la siguiente forma:

```
rpmf0 <- function(x, prob = 1/length(x), n = 1000) {
  X <- numeric(n)
  U <- runif(n)
  for(j in 1:n) {
    i <- 1
    Fx <- prob[1]
    while (Fx < U[j]) {
```

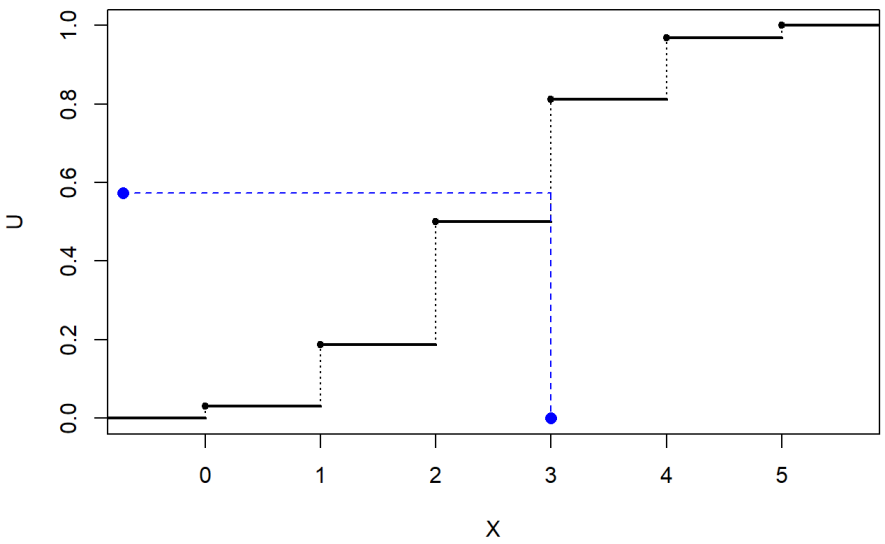


Figura 5.1: Ilustración de la simulación de una distribución discreta mediante transformación cuantil (con búsqueda secuencial).

```
      i <- i + 1
      Fx <- Fx + prob[i]
    }
    X[j] <- x[i]
  }
  return(X)
}
```

Adicionalmente, para disminuir ligeramente el tiempo de computación, se pueden almacenar las probabilidades acumuladas en una tabla. Este algoritmo está implementado en la función `rpmf()` del paquete `simres` (fichero *rpmf.R*), que ademas devuelve el número de comparaciones en un atributo `ncomp`:

```
library(simres)
rpmf
```

```
## function(x, prob = 1/length(x), n = 1000, as.factor = FALSE) {
##   # Numero de comparaciones
##   ncomp <- 0
##   # Inicializar FD
##   Fx <- cumsum(prob)
##   # Simular
##   X <- numeric(n)
##   U <- runif(n)
##   for(j in 1:n) {
##     i <- 1
##     while (Fx[i] < U[j]) i <- i + 1
##     X[j] <- x[i]
##     ncomp <- ncomp + i
##   }
##   if(as.factor) X <- factor(X, levels = x)
```

```
##   attr(X, "ncomp") <- ncomp
##   return(X)
## }
## <bytecode: 0x000000003ec29d68>
## <environment: namespace:simres>
```

Ejemplo 5.1 (Simulación de una binomial mediante el método de la transformación cuantil) Empleamos la rutina anterior para para generar una muestra de $nsim = 10^5$ observaciones de una variable $\mathcal{B}(10, 0.5)$ y obtenemos el tiempo de CPU empleado:

Calcular también la media muestral (compararla con la teórica np) y el número medio de comparaciones para generar cada observación.

Empleamos la rutina anterior para generar las simulaciones:

```r
set.seed(1)
n <- 10
p <- 0.5
nsim <- 10^5
x <- 0:n
pmf <- dbinom(x, n, p)
system.time( rx <- rpmf(x, pmf, nsim) )
```

```
##    user  system elapsed
##    0.05    0.00    0.05
```

A partir de ellas podríamos aproximar el valor esperado:

```r
mean(rx)
```

```
## [1] 4.99697
```

aunque en este caso el valor teórico es conocido $np = 5$.

Calculamos el número medio de comparaciones para generar cada observación:

```r
ncomp <- attr(rx, "ncomp")
ncomp/nsim
```

```
## [1] 5.99697
```

```r
# Se verá más adelante que el valor teórico es sum((1:length(x))*pmf)
```

Representamos la aproximación por simulación de la función de masa de probabilidad y la comparamos con la teórica:

```r
res <- table(rx)/nsim
# res <- table(factor(rx, levels = x))/nsim
plot(res, ylab = "frecuencia relativa", xlab = "valor")
points(x, pmf, pch = 4, col = "blue")  # Comparación teórica
```

También podríamos realizar comparaciones numéricas:

```r
res <- as.data.frame(res)
names(res) <- c("x", "psim")
res$pteor <- pmf
print(res, digits = 2)
```

```
##     x    psim   pteor
## 1   0 0.00100 0.00098
## 2   1 0.00981 0.00977
## 3   2 0.04457 0.04395
## 4   3 0.11865 0.11719
```

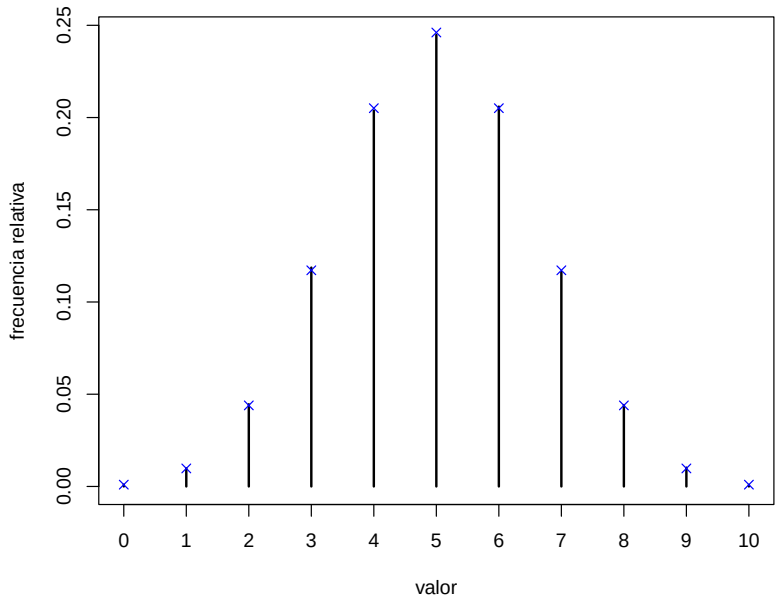


Figura 5.2: Comparación de las frecuencias relativas de los valores generados, mediante el método de la transformación cuantil, con las probabilidades teóricas.

```
## 5   4 0.20377 0.20508
## 6   5 0.24477 0.24609
## 7   6 0.20593 0.20508
## 8   7 0.11631 0.11719
## 9   8 0.04415 0.04395
## 10  9 0.01021 0.00977
## 11 10 0.00083 0.00098
```

```
# Máximo error absoluto
max(abs(res$psim - res$pteor))
```

```
## [1] 0.0014625
```

```
# Máximo error porcentual absoluto
100*max(abs(res$psim - res$pteor) / res$pteor)
```

```
## [1] 15.008
```

Nota: Puede ocurrir que no todos los valores sean generados en la simulación. En el código anterior si `length(x) > length(psim)`, la sentencia `res$pteor <- pmf` generará un error. Una posible solución sería trabajar con factores (llamar a la función `rpmf()` con `as.factor = TRUE` o emplear `res <- table(factor(rx, levels = x))/nsim`).

5.1.1 Eficiencia del algoritmo

Si consideramos la variable aleatoria $\mathcal{J}$ correspondiente a las etiquetas, su función de masa de probabilidad sería:

i	1	2	$\cdots$	n	$\cdots$
$P(\mathcal{J} = i)$	p_1	p_2	$\cdots$	p_n	$\cdots$

y el número de comparaciones en el paso 3 sería un valor aleatorio de esta variable. Una medida de la eficiencia del algoritmo de la transformación cuantil es el número medio de comparaciones:

$$E(\mathcal{J}) = \sum_i i p_i.$$

Realmente, cuando la variable toma un número finito de valores: x_1, x_2, ..., x_n, no sería necesario hacer la última comprobación $U > \sum_{i=1}^{n} p_i = 1$ y se generaría directamente x_n, por lo que el número medio de comparaciones sería:

$$\sum_{i=1}^{n-1} i p_i + (n-1) p_n.$$

Para disminuir el número esperado de comparaciones podemos reordenar los valores x_i de forma que las probabilidades correspondientes sean decrecientes. Esto equivale a considerar un etiquetado l de forma que:

$$p_{l(1)} \geq p_{l(2)} \geq \cdots \geq p_{l(n)} \geq \cdots$$

Ejemplo 5.2 (Simulación de una binomial, continuación)

Podemos repetir la simulación del Ejemplo 5.1 anterior ordenando previamente las probabilidades en orden decreciente y también empleando la función `sample()` de R.

```
set.seed(1)
tini <- proc.time()
# Ordenar
ind <- order(pmf, decreasing = TRUE)
# Generar
rx <- rpmf(x[ind], pmf[ind], nsim)
# Tiempo de CPU
tiempo <- proc.time() - tini
tiempo
```

```
##    user  system elapsed
##    0.05    0.00    0.04
```

```
# Número de comparaciones
ncomp <- attr(rx, "ncomp")
ncomp/nsim
```

```
## [1] 3.08369
```

```
sum((1:length(x))*pmf[ind]) # Valor teórico
```

```
## [1] 3.083984
```

Como ya se comentó, en R se recomienda emplear la función `sample()` (implementa eficientemente el método de Alias descrito en la Sección 5.3):

```
system.time( rx <- sample(x, nsim, replace = TRUE, prob = pmf) )
```

```
##    user  system elapsed
##       0       0       0
```

5.2 Método de la tabla guía

También conocido como método de búsqueda indexada (*Indexed Search*), la idea consiste en construir m subintervalos equiespaciados en $[0, 1]$ de la forma:

$$I_j = [u_j, u_{j+1}) = \left[\frac{j-1}{m}, \frac{j}{m}\right) \text{ para } j = 1, 2, \ldots, m$$

y utilizarlos como punto de partida para la búsqueda. En una tabla guía se almacenan los indices de los cuantiles correspondientes a los extremos inferiores de los intervalos:

$$g_j = Q_{\mathcal{J}}(u_j) = \inf\left\{i : F_i \geq u_j = \frac{j-1}{m}\right\}$$

El punto de partida para un valor U será g_{j_0} con:

$$j_0 = \lfloor mU \rfloor + 1$$

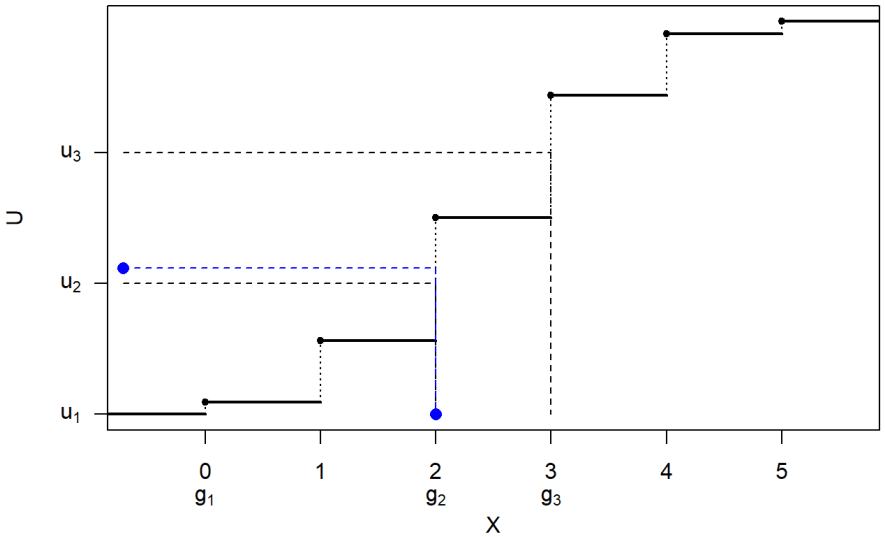


Figura 5.3: Ilustración de la simulación de una distribución discreta mediante tabla guía.

En este caso, puede verse que una cota del número medio de comparaciones es:

$$E(N) \leq 1 + \frac{n}{m}$$

Algoritmo 5.3 (de simulación mediante tabla guía; Chen y Asau, 1974)
Inicialización:

1. Hacer $F_1 = p_1$.
2. Desde $i = 2$ hasta n hacer $F_i = F_{i-1} + p_i$.

Cálculo de la tabla guía:

1. Hacer $g_1 = 1$ e $i = 1$.
2. Desde $j = 2$ hasta m hacer

   2.a Mientras $(j-1)/m > F_i$ hacer $i = i + 1$.

   2.b $g_j = i$

Simulación mediante tabla guía:

1. Generar $U \sim \mathcal{U}(0, 1)$.

2. Hacer $j = \lfloor mU \rfloor + 1$.
3. Hacer $i = g_j$.
4. Mientras $U > F_i$ hacer $i = i + 1$.
5. Devolver $X = x_i$.

Este algoritmo está implementado en la función `simres::rpmf.table()` (fichero *rpmf.R*) y devuelve también el número de comparaciones en un atributo `ncomp`:

```
rpmf.table
```

```
## function(x, prob = 1/length(x), m, n = 1000, as.factor = FALSE) {
##   # Inicializar tabla y FD
##   Fx <- cumsum(prob)
##   g <- rep(1,m)
##   i <- 1
##   for(j in 2:m) {
##     while (Fx[i] < (j-1)/m) i <- i + 1
##     g[j] <- i
##   }
##   ncomp <- i - 1
##   # Generar valores
##   X <- numeric(n)
##   U <- runif(n)
##   for(j in 1:n) {
##     i <- i0 <- g[floor(U[j] * m) + 1]
##     while (Fx[i] < U[j]) i <- i + 1
##     ncomp <- ncomp + i - i0
##     X[j] <- x[i]
##   }
##   if(as.factor) X <- factor(X, levels = x)
##   attr(X, "ncomp") <- ncomp
##   return(X)
## }
## <bytecode: 0x000000003c25d090>
## <environment: namespace:simres>
```

Ejemplo 5.3 (Simulación de una binomial mediante tabla guía)

Repetimos la simulación del Ejemplo 5.1 anterior empleando esta rutina con $m = n - 1$.

```
set.seed(1)
system.time( rx <- rpmf.table(x, pmf, n-1, nsim) )
```

```
##    user  system elapsed
##    0.04    0.00    0.04
```

Número medio de comparaciones:

```
ncomp <- attr(rx, "ncomp")
ncomp/nsim
```

```
## [1] 0.55951
```

```
sum((1:length(x))*pmf) # Numero esperado con búsqueda secuencial
```

```
## [1] 6
```

Análisis de los resultados:

```
res <- table(rx)/nsim
plot(res, ylab = "frecuencia relativa", xlab = "valores")
points(x, pmf, pch = 4, col = "blue")  # Comparación teórica
```

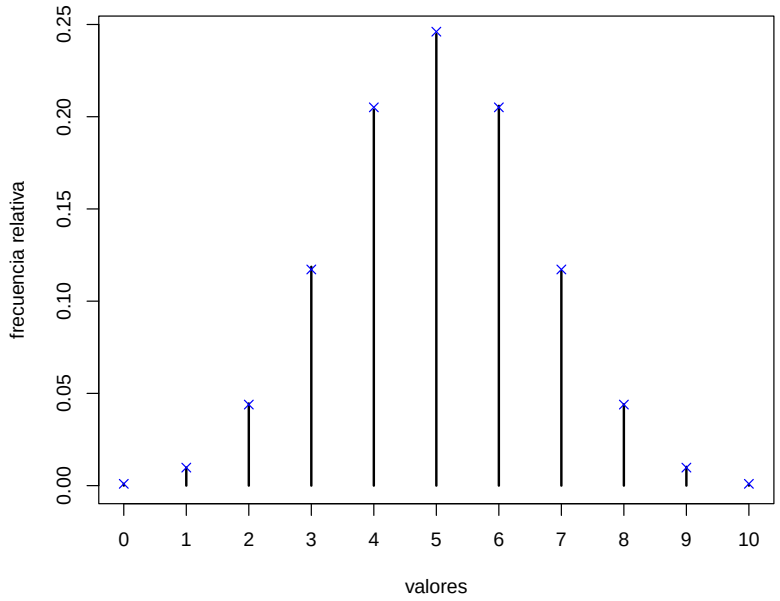


Figura 5.4: Comparación de las frecuencias relativas de los valores generados, mediante el método de la tabla guía, con las probabilidades teóricas.

5.3 Método de Alias

Se basa en representar la distribución de X como una mixtura (uniforme) de variables dicotómicas (Walker, 1977):

$$Q^{(i)} = \begin{cases} x_i & \text{con prob. } q_i \\ x_{a_i} & \text{con prob. } 1 - q_i \end{cases}$$

Hay varias formas de construir las tablas de probabilidades q_i y de alias a_i. Se suele emplear el denominado algoritmo "Robin Hood" de inicialización (Kronmal y Peterson, 1979). La idea es "tomar prestada" parte de la probabilidad de los valores más probables (ricos) para asignársela a los valores menos probables (pobres), recordando el valor de donde procede (almacenando el índice en a_i).

Algoritmo 5.4 ("Robin Hood" de inicialización; Kronmal y Peterson, 1979)

1. Desde $i = 1$ hasta n hacer $q_i = np_i$.
2. Establecer $L = \{l : q_l < 1\}$ y $H = \{h : q_h \geq 1\}$.
3. Si L ó H vacios terminar.
4. Seleccionar $l \in L$ y $h \in H$.
5. Hacer $a_l = h$.
6. Eliminar l de L.
7. Hacer $q_h = q_h - (1 - q_l)$.

8. Si $q_h < 1$ mover h de H a L.
9. Ir al paso 3.

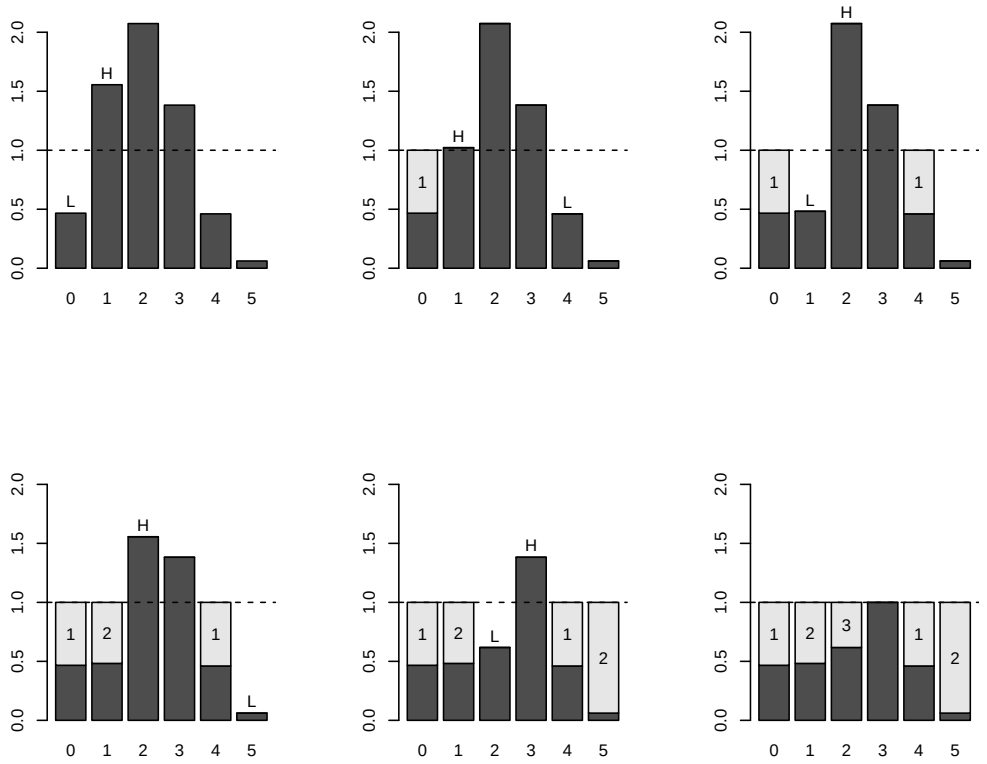


Figura 5.5: Pasos del algoritmo de inicialización del método Alias.

El algoritmo para generar las simulaciones es el estándar del método de composición:

Algoritmo 5.5 (método alias de simulación; Walker, 1977)

1. Generar $U, V \sim \mathcal{U}(0,1)$.
2. Hacer $i = \lfloor nU \rfloor + 1$.
3. Si $V < q_i$ devolver $X = x_i$.
4. En caso contrario devolver $X = x_{a_i}$.

Este algoritmo es muy eficiente y es el empleado en la función `sample()` de R[1].

Este método también está implementado en la función `simres::rpmf.alias()` (fichero *rpmf.R*), empleando código R menos eficiente:

[1] R implementa este algoritmo en el fichero fuente random.c (para muestreo probabilístico con reemplazamiento, función C `walker_ProbSampleReplace()`), aunque el paso 2 del algoritmo de simulación empleado por defecto cambió ligeramente a partir de la versión 3.6.0 para evitar posibles problemas de redondeo (ver Sección 1.3.1).

```
rpmf.alias
```

```
## function(x, prob = 1/length(x), n = 1000, as.factor = FALSE) {
##   # Inicializar tablas
##   a <- numeric(length(x))
##   q <- prob*length(x)
##   low <- q < 1
##   high <- which(!low)
##   low <- which(low)
##   while (length(high) && length(low)) {
##     l <- low[1]
##     h <- high[1]
##     a[l] <- h
##     q[h] <- q[h] - (1 - q[l])
##     if (q[h] < 1) {
##       high <- high[-1]
##       low[1] <- h
##     } else low <- low[-1]
##   } # while
##   # Generar valores
##   V <- runif(n)
##   i <- floor(runif(n)*length(x)) + 1
##   X <- x[ ifelse( V < q[i], i, a[i]) ]
##   if(as.factor) X <- factor(X, levels = x)
##   return(X)
## }
## <bytecode: 0x0000000035e78600>
## <environment: namespace:simres>
```

Ejemplo 5.4 (Simulación de una binomial mediante en método de Alias)

Repetimos la simulación del Ejemplo 5.1 anterior empleando esta rutina.

```
set.seed(1)
system.time( rx <- rpmf.alias(x, pmf, nsim) )
```

```
##    user  system elapsed
##    0.02    0.00    0.02
```

Análisis de los resultados:

```
res <- table(rx)/nsim
plot(res, ylab = "frecuencia relativa", xlab = "valores")
points(x, pmf, pch = 4, col = "blue")  # Comparación teórica
```

5.4 Simulación de una variable discreta con dominio infinito

Los métodos anteriores están pensados para variables que toman un número finito de valores. Si la variable discreta tiene dominio infinito no se podrían almacenar las probabilidades acumuladas, aunque en algunos casos podrían calcularse de forma recursiva.

Ejemplo 5.5 (distribución de Poisson)

Una variable X con distribución de Poisson de parámetro λ, toma los valores $x_1 = 0$, $x_2 = 1$, ... con probabilidades:

$$p_j = P\left(X = x_j\right) = P\left(X = j - 1\right) = \frac{e^{-\lambda}\lambda^{j-1}}{(j-1)!}, \; j = 1, 2, ...$$

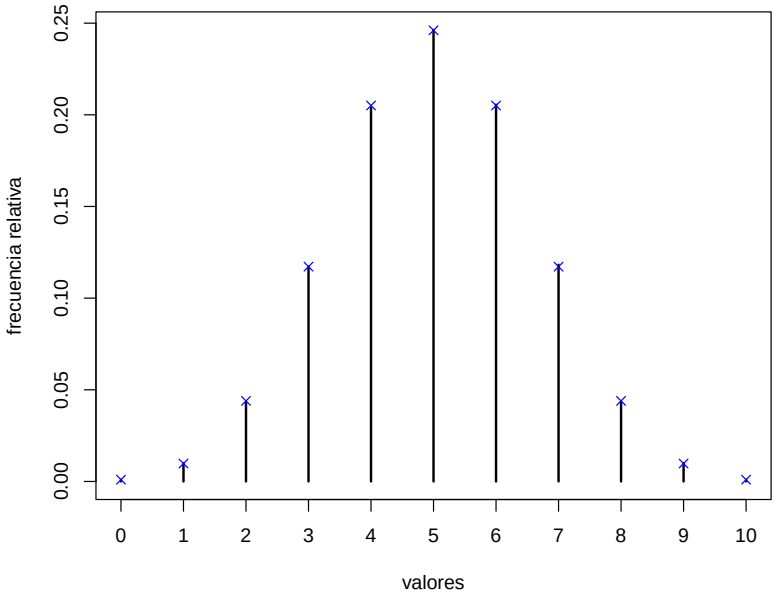


Figura 5.6: Comparación de las frecuencias relativas de los valores generados, mediante el método de alias, con las probabilidades teóricas.

En este caso, como:

$$P(X=j) = \frac{e^{-\lambda}\lambda^j}{j!} = \frac{\lambda}{j}\frac{e^{-\lambda}\lambda^{j-1}}{(j-1)!} = \frac{\lambda}{j}P(X=j-1),$$

el algoritmo de inversión con búsqueda secuencial sería:

1. Generar $U \sim \mathcal{U}(0,1)$.
2. Hacer $I=0$, $p=e^{-\lambda}$ y $S=p$.
3. Mientras $U > S$ hacer $I=I+1$, $p=\frac{\lambda}{I}p$ y $S=S+p$.
4. Devolver $X=I$.

Hay modificaciones de los algoritmos anteriores, por ejemplo el de tabla guía con búsqueda secuencial en la cola de la distribución, para variables con dominio infinito.

Como alternativa, siempre se puede pensar en truncar la distribución, eliminando los valores muy poco probables (teniendo en cuenta el número de generaciones que se pretenden realizar), aunque la distribución de las simulaciones será aproximada.

5.5 Cálculo directo de la función cuantil

En ocasiones el método de la transformación cuantil puede acelerarse computacionalmente porque, mediante cálculos directos, es posible encontrar el valor de la función cuantil en cualquier U, evitando el bucle de búsqueda. Normalmente se realiza mediante truncamiento de una distribución continua.

Ejemplo 5.6 (distribución uniforme discreta)

La función de masa de probabilidad de una distribución uniforme discreta en $\{1,2,\dots,n\}$ viene dada por

$$p_j = \frac{1}{n}, \text{ para } j=1,2,\dots n.$$

Pueden generarse valores de esta distribución de forma muy eficiente truncando la distribución uniforme:

1. Generar $U \sim \mathcal{U}(0,1)$.
2. Devolver $X = \lfloor nU \rfloor + 1$.

Ejemplo 5.7 (distribución geométrica)
La función de masa de probabilidad de una distribución geométrica es:

$$P(X = j) = P(I = j + 1) = p(1-p)^j, \; j = 0, 1, ...$$

Si se considera como variable aleatoria continua auxiliar una exponencial, con función de distribución $G(x) = 1 - e^{-\lambda x}$ si $x \geq 0$, se tiene que:

$$\begin{aligned} G(i) - G(i-1) &= 1 - e^{-\lambda i} - \left(1 - e^{-\lambda(i-1)}\right) = e^{-\lambda(i-1)} - e^{-\lambda i} \\ &= e^{-\lambda(i-1)}\left(1 - e^{-\lambda}\right) = \left(1 - e^{-\lambda}\right)\left(e^{-\lambda}\right)^{i-1} \\ &= p(1-p)^{i-1}, \end{aligned}$$

tomando $p = 1 - e^{-\lambda}$. De donde se obtendría el algoritmo:

0. Hacer $\lambda = -\ln(1-p)$.
1. Generar $U \sim \mathcal{U}(0,1)$.
2. Hacer $T = -\frac{\ln U}{\lambda}$.
3. Devolver $X = \lfloor T \rfloor$.

5.6 Otros métodos

Muchos de los métodos descritos en el Capítulo 4 para variables continuas son directamente aplicables al caso discreto:

- Aceptación-Rechazo (Sección 4.2): En principio habría que considerar una variable auxiliar discreta con el mismo soporte, pero también hay modificaciones para variables auxiliares continuas.
- Método de composición (Sección 4.4): es uno de los más empleados, por ejemplo en el método de Alias (Sección 5.3) y para simular la distribución binomial negativa (Sección 5.7).

Hay otros métodos que tratan de reducir el número medio de comparaciones de la búsqueda secuencial, por ejemplo los árboles (binarios) de Huffman (e.g. Cao, 2002, Sección 4.2). Estos métodos son muy poco eficientes para simular variables discretas pero pueden resultar de utilidad para diseñar experimentos de simulación más complejos (la idea es la misma, preocuparse principalmente por los sucesos más probables).

5.7 Métodos específicos para generación de distribuciones notables

Los comentarios al principio de la Sección 4.5 para el caso de variables continuas serían válidos también para distribuciones notables discretas.

Entre los distintos métodos disponibles para la generación de las distribuciones discretas más conocidas podríamos destacar el de la distribución binomial negativa mediante el método de composición (Sección 4.4).

La distribución binomial negativa, $BN(r,p)$, puede interpretarse como el número de fracasos antes del r-ésimo éxito[2] y su función de masa de probabilidad es

$$P(X=i) = \binom{i+r-1}{i} p^r (1-p)^i, \text{ para } i = 0, 1, ...$$

A partir de la propiedad

$$X|_Y \sim \text{Pois}\,(Y)\,,\, Y \sim \text{Gamma}\left(r, \frac{p}{1-p}\right) \Rightarrow X \sim BN(r,p)$$

se puede deducir el siguiente método específico de simulación.

Algoritmo 5.6 (distribución binomial negativa)

1. Simular $L \sim \text{Gamma}\left(r, \frac{p}{1-p}\right)$.
2. Simular $X \sim Pois\,(L)$.
3. Devolver X.

Por este motivo se denominada también a esta distribución *Gamma-Poisson*. Empleando una aproximación similar podríamos generar otras distribuciones, como la *Beta-Binomial*, empleadas habitualmente en inferencia bayesiana.

5.8 Ejercicios

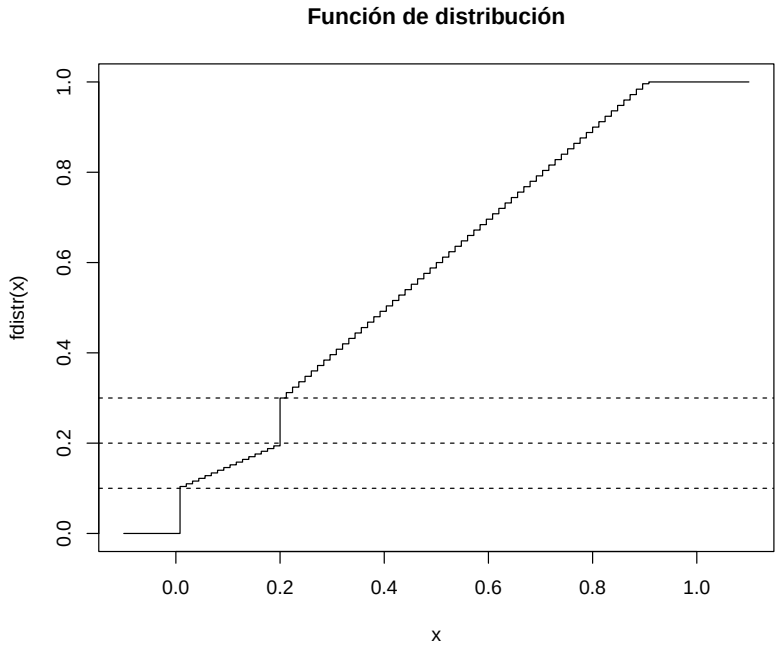
Ejercicio 5.1 (Simulación de una distribución mixta mediante el método de inversión generalizado) Consideramos la variable aleatoria con función de distribución dada por:

$$F(x) = \begin{cases} 0 & \text{si } x < 0 \\ \frac{x}{2} + \frac{1}{10} & \text{si } x \in [0, \frac{1}{5}) \\ x + \frac{1}{10} & \text{si } x \in [\frac{1}{5}, \frac{9}{10}] \\ 1 & \text{en otro caso} \end{cases}$$

Esta función está implementada en el siguiente código:

```
fdistr <- function(x) {
ifelse(x < 0, 0,
    ifelse(x < 1/5, x/2 + 1/10,
        ifelse(x <= 9/10, x + 1/10, 1) ) )
}
# Empleando ifelse la función es vectorial (y podemos emplear curve...)
curve(fdistr, from = -0.1, to = 1.1, type = 's',
      main = 'Función de distribución')
# Discontinuidades en 0 y 1/5
abline(h = c(1/10, 2/10, 3/10), lty = 2)
```

[2]La distribución binomial negativa es una generalización de la geométrica y, debido a su reproductividad en el parámetro r, podría simularse como suma de r variables geométricas. Sin embargo, este algoritmo puede ser muy costoso en tiempo de computación si r es elevado.



Nota: Esta variable toma los valores 0 y 1/5 con probabilidad 1/10.

a) Diseñar un algoritmo basado en el método de inversión generalizado para generar observaciones de esta variable.

b) Implementar el algoritmo en una función que permita generar $nsim$ valores de esta variable.

Ver solución en Sección D.3.1.

Ejercicio 5.2 (distribución hipergeométrica)

Se pretende simular $nsim = 10^4$ observaciones de una variable hipergeométrica (`dhyper(x, m, n, k)`) de parámetros $m =$ el número de grupo multiplicado por 10, $n = 100 - m$ y $k = 20$.

a) Comprobar que el rango de posibles valores de esta variable es `max(0, k-n):min(m, k)`. Generar los valores empleando el método de la transformación cuantil usando búsqueda secuencial. Obtener el tiempo de CPU empleado. Aproximar por simulación la función de masa de probabilidad, representarla gráficamente y compararla con la teórica. Calcular también la media muestral (compararla con la teórica $km/(m+n)$) y el número medio de comparaciones para generar cada observación.

b) Repetir el apartado anterior: ordenando previamente las probabilidades en orden decreciente, empleando la función `sample` de R, mediante una tabla guía (con $k-1$ subintervalos) y usando el método de Alias.

Capítulo 6

Simulación de distribuciones multivariantes

La simulación de vectores aleatorios $\mathbf{X} = (X_1, X_2, \dots, X_d)$ que sigan cierta distribución dada no es tarea siempre sencilla. En general, no resulta una extensión inmediata del caso unidimensional, aunque muchos de los algoritmos descritos en los temas anteriores (como el de aceptación-rechazo o el de composición) son válidos para distribuciones multivariantes. En este caso sin embargo, puede ser mucho más difícil cumplir los requerimientos (e.g. encontrar una densidad auxiliar adecuada) y los algoritmos obtenidos pueden ser computacionalmente poco eficientes (especialmente si el número de dimensiones es grande).

En las primeras secciones de este capítulo supondremos que se pretende simular una variable aleatoria multidimensional continua $\mathbf{X}$ con función de densidad conjunta $f(x_1, x_2, \dots, x_d)$ (aunque muchos resultados serán válidos para variables discretas multidimensionales, simplemente cambiando funciones de densidad por las correspondientes de masa de probabilidad). En la Sección 6.7 se tratará brevemente la simulación de vectores aleatorios discretos y de tablas de contingencia, centrándose en el caso bidimensional.

6.1 Simulación de componentes independientes

Si las componentes son independientes y f_i son las correspondientes densidades marginales, bastará con generar $X_i \sim f_i$. Las dificultades aparecerán cuando se quiera simular componentes con una determinada estructura de dependencia.

Ejemplo 6.1 (simulación de normales independientes)
Si $\mu = (\mu_1, \mu_2, \dots, \mu_d)^t$ es un vector (de medias) y Σ es una matriz $d \times d$ definida positiva (de varianzas-covarianzas), el vector aleatorio $\mathbf{X}$ sigue una distribución normal multivariante con esos parámetros, $\mathbf{X} \sim \mathcal{N}_d(\mu, \Sigma)$, si su función de densidad es de la forma:

$$f(\mathbf{x}) = \frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\mathbf{x}-\mu)^t\Sigma^{-1}(\mathbf{x}-\mu)\right),$$

donde $|\Sigma|$ es el determinante de Σ.

Si la matriz de covarianzas es diagonal $\Sigma = diag(\sigma_1^2, \sigma_2^2, \dots, \sigma_d^2)$, entonces las componentes $X_i \sim \mathcal{N}(\mu_i, \sigma_i^2)$ son independientes y podemos simular el vector aleatorio de forma trivial, por ejemplo mediante el siguiente algoritmo:

Algoritmo 6.1 (de simulación de normales independientes)

1. Simular $Z_1, Z_2, \dots, Z_d \sim \mathcal{N}(0, 1)$ independientes.

2. Para $i = 1, 2, \ldots, d$ hacer $X_i = \mu_i + \sigma_i Z_i$.

Las funciones implementadas en el paquete base de R permiten simular fácilmente en el caso independiente ya que admiten vectores como parámetros. Por ejemplo en el caso bidimensional con $X_1 \sim \mathcal{N}(0, 1)$ y $X_2 \sim \mathcal{N}(-1, 0.5^2)$:

```
f1 <- function(x) dnorm(x)
f2 <- function(x) dnorm(x, -1, 0.5)
curve(f1, -3, 3, ylim = c(0, f2(-1)), ylab = "fdp")
curve(f2, add = TRUE, lty = 2)
```

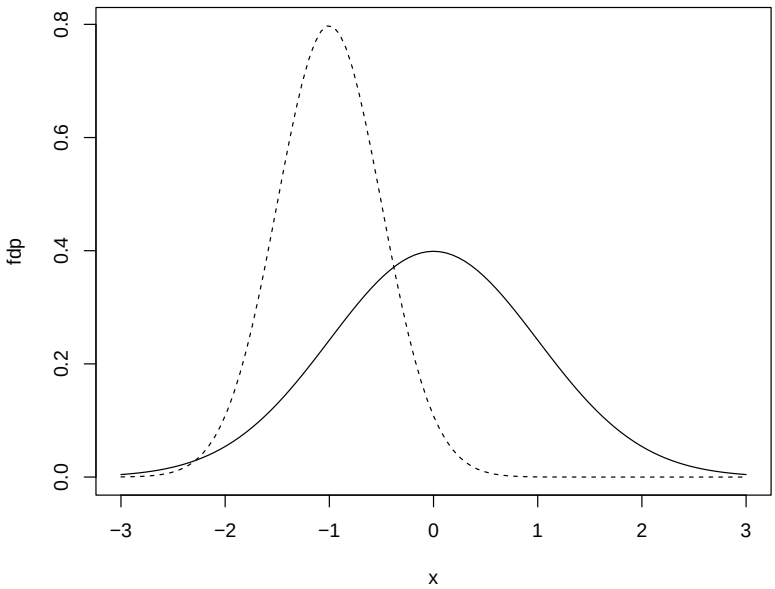


Figura 6.1: Densidades marginales de las componentes del Ejemplo 6.1.

Para simular una generación bastaría con:

```
set.seed(1)
rnorm(2, c(0, -1), c(1, 0.5))
```

```
## [1] -0.6264538 -0.9081783
```

y para simular `nsim`:

```
set.seed(1)
nsim <- 5
rx <- matrix(rnorm(2*nsim, c(0, -1), c(1, 0.5)), nrow = nsim, byrow = TRUE)
colnames(rx) <- paste0("X", 1:ncol(rx))
rx
```

```
##              X1         X2
## [1,] -0.6264538 -0.9081783
## [2,] -0.8356286 -0.2023596
## [3,]  0.3295078 -1.4102342
## [4,]  0.4874291 -0.6308376
## [5,]  0.5757814 -1.1526942
```

6.2 El método de aceptación/rechazo

El algoritmo de aceptación-rechazo es el mismo que el del caso univariante descrito en la Sección 4.2, la única diferencia es que las densidades son multidimensionales. Supongamos que la densidad objetivo f y la densidad auxiliar g verifican:

$$f(x_1, x_2, \dots, x_d) \leq c \cdot g(x_1, x_2, \dots, x_d), \ \forall \mathbf{x} = (x_1, x_2, \dots, x_d) \in \mathbb{R}^d.$$

para una constante $c > 0$. El algoritmo sería:

1. Generar $U \sim \mathcal{U}(0,1)$.
2. Generar $\mathbf{T} = (T_1, T_2, \dots, T_d) \sim g$.
3. Si $c \cdot U \cdot g(T_1, T_2, \dots, T_d) \leq f(T_1, T_2, \dots, T_d)$ devolver $\mathbf{X} = \mathbf{T}$.

   En caso contrario volver al paso 1.

Por ejemplo, de forma análoga al caso unidimensional, en el caso de una densidad acotada en un hipercubo (intervalo cerrado multidimensional) siempre podríamos considerar una uniforme como densidad auxiliar.

Ejemplo 6.2 (distribución bidimensional acotada)
Supongamos que estamos interesados en generar valores de una variable aleatoria bidimensional (X, Y) con función de densidad:

$$f(x,y) = \begin{cases} \frac{3}{16}\left(2 - (x^2 + y^2)\right) & \text{si } x \in [-1,1] \text{ e } y \in [-1,1] \\ 0 & \text{en otro caso} \end{cases}$$

Podríamos considerar como densidad auxiliar la uniforme en $[-1,1] \times [-1,1]$:

$$g(x,y) = \begin{cases} \frac{1}{4} & \text{si } x \in [-1,1] \text{ e } y \in [-1,1] \\ 0 & \text{en otro caso} \end{cases}$$

Como $f(x,y) \leq M = f(0,0) = \frac{3}{8}$, tomando $c = \frac{M}{g(x,y)} = \frac{3}{2}$ tendríamos que $f(x,y) \leq cg(x,y) = M$ y el algoritmo sería:

1. Generar $U \sim \mathcal{U}(0,1)$.
2. Generar $T_1, T_2 \sim \mathcal{U}(-1,1)$.
3. Si $M \cdot U \leq f(T_1, T_2)$ devolver $\mathbf{X} = (T_1, T_2)$.
4. En caso contrario volver al paso 1.

En este caso, la condición de aceptación del paso 3 simplificada sería: $U \leq 1 - \left(T_1^2 + T_2^2\right)/2$.

Ejercicio 6.1 (distribución bidimensional acotada)
Escribir una función que implemente el algoritmo del ejemplo anterior, emplearla para generar 1000 observaciones y representar gráficamente su distribución (puede resultar de interés la función `sm::sm.density()` o las herramientas del paquete `ks`).

Ejemplo 6.3 (distribución uniforme en la esfera)
Supongamos que el objetivo es simular puntos uniformemente distribuidos sobre la "esfera" unitaria d-dimensional (ver Figura D.1):

$$C_d = \left\{(x_1, x_2, \dots, x_d) \in \mathbb{R}^d : x_1^2 + x_2^2 + \dots + x_d^2 \leq 1\right\}.$$

Denotando por $V_d(1)$, el "volumen" (la medida) de la esfera d-dimensional de radio 1 (en general, la de radio r verifica $V_d(r) = r^d V_d(1)$), se tiene:

$$f(x_1, x_2, \dots, x_d) = \begin{cases} \frac{1}{V_d(1)} & \text{si } (x_1, x_2, \dots, x_d) \in C_d \\ 0 & \text{si } (x_1, x_2, \dots, x_d) \notin C_d \end{cases}$$

Para simular valores en $\mathbb{R}^d$, con densidad f, podemos utilizar como distribución auxiliar una $\mathcal{U}\left([-1,1]\times[-1,1]\times\overset{\text{d.}}{\cdots}\times[-1,1]\right)=\mathcal{U}\left([-1,1]^d\right)$, dada por:

$$g\left(x_1,x_2,\ldots,x_d\right)=\begin{cases}\frac{1}{2^d} & \text{si } x_i\in[-1,1]\,,\ \text{para todo } i=1,2,\ldots,d\\ 0 & \text{en otro caso}\end{cases}$$

La constante c óptima para la utilización del método de aceptación/rechazo es:

$$c=\max_{\{\mathbf{x}:g(\mathbf{x})>0\}}\frac{f\left(\mathbf{x}\right)}{g\left(\mathbf{x}\right)}=\frac{\frac{1}{V_d(1)}}{\frac{1}{2^d}}=\frac{2^d}{V_d\left(1\right)}$$

y la condición de aceptación $cUg\left(\mathbf{T}\right)\leq f\left(\mathbf{T}\right)$ se convierte en:

$$\frac{2^d}{V_d\left(1\right)}U\frac{1}{2^d}1_{[-1,1]^d}\left(\mathbf{T}\right)\leq\frac{1}{V_d\left(1\right)}1_{C_d}\left(\mathbf{T}\right),$$

o, lo que es lo mismo, $U1_{[-1,1]^d}\left(\mathbf{T}\right)\leq 1_{C_d}\left(\mathbf{T}\right)$. Dado que el número aleatorio U está en el intervalo $(0,1)$ y que las funciones indicadoras valen 0 ó 1, esta condición equivale a que $1_{[-1,1]^d}\left(\mathbf{T}\right)=1_{C_d}\left(\mathbf{T}\right)$, es decir, a que $\mathbf{T}\in C_d$, por tanto, a que se verifique:

$$T_1^2+T_2^2+\cdots+T_d^2\leq 1.$$

Por otra parte, la simulación de $T\sim\mathcal{U}\left([-1,1]^d\right)$ puede hacerse trivialmente mediante $T_i\sim\mathcal{U}\left(-1,1\right)$ para cada $i=1,2,\ldots,d$, ya que las componentes son independientes. Como el valor de U es superfluo en este caso, el algoritmo queda:

1. Simular $V_1,V_2,\ldots,V_d\sim\mathcal{U}\left(0,1\right)$ independientes.
2. Para $i=1,2,\ldots,d$ hacer $T_i=2V_i-1$.
3. Si $T_1^2+T_2^2+\cdots+T_d^2>1$ entonces volver al paso 1.
4. Devolver $\mathbf{X}=\left(T_1,T_2,\ldots,T_d\right)^t$.

Ver el Ejercicio 1.1 para el caso de $d=2$.

Usando las fórmulas del "volumen" de una "esfera" d-dimensional:

$$V_d\left(r\right)=\begin{cases}\dfrac{\pi^{d/2}r^d}{(d/2)!} & \text{si } d \text{ es par}\\[2ex] \dfrac{2^{\lfloor\frac{d}{2}\rfloor+1}\pi^{\lfloor\frac{d}{2}\rfloor}r^d}{1\cdot3\cdot5\cdots d} & \text{si } d \text{ es impar}\end{cases}$$

puede verse que el número medio de iteraciones del algoritmo, dado por la constante $c=\frac{2^d}{V_d(1)}$, puede llegar a ser enormemente grande. Así, si $d=2$ se tiene $c=1.27$, si $d=3$ se tiene $c=1.91$, si $d=4$ entonces $c=3.24$ y para $d=10$ resulta $c=401.5$ que es un valor que hace que el algoritmo sea tremendamente lento en dimensión 10. Esto está relacionado con la *maldición de la dimensionalidad* (curse of dimensionality), a medida que aumenta el número de dimensiones el volumen de la "frontera" crece exponencialmente (ver p.e. Fernández-Casal et al, 2021, Sección 1.4).

6.3 Factorización de la matriz de covarianzas

Teniendo en cuenta que si $Cov(\mathbf{X})=I$, entonces:

$$Cov(A\mathbf{X})=AA^t.$$

La idea de este tipo de métodos es simular datos independientes y transformarlos linealmente de modo que el resultado tenga la covarianza deseada $\Sigma=AA^t$.

Este método se emplea principalmente para la simulación de una normal multivariante, aunque también es válido para muchas otras distribuciones como la t-multivariante.

En el caso de normalidad, el resultado general es el siguiente.

Proposición 6.1
Si $\mathbf{X} \sim \mathcal{N}_d(\mu, \Sigma)$ y A es una matriz $p \times d$, de rango máximo, con $p \leq d$, entonces:

$$A\mathbf{X} \sim \mathcal{N}_p(A\mu, A\Sigma A^t).$$

Partiendo de $\mathbf{Z} \sim \mathcal{N}_d(\mathbf{0}, I_d)$, se podrían considerar distintas factorizaciones de la matriz de covarianzas:

- Factorización espectral: $\Sigma = H\Lambda H^t = H\Lambda^{1/2}(H\Lambda^{1/2})^t$, donde H es una matriz ortogonal (i.e. $H^{-1} = H^t$), cuyas columnas son los autovectores de la matriz Σ, y Λ es una matriz diagonal, cuya diagonal esta formada por los correspondientes autovalores (positivos). De donde se deduce que:

$$\mathbf{X} = \mu + H\Lambda^{1/2}\mathbf{Z} \sim \mathcal{N}_d(\mu, \Sigma).$$

- Factorización de Cholesky: $\Sigma = LL^t$, donde L es una matriz triangular inferior (fácilmente invertible), por lo que:

$$\mathbf{X} = \mu + L\mathbf{Z} \sim \mathcal{N}_d(\mu, \Sigma).$$

Desde el punto de vista de la eficiencia computacional la factorización de Cholesky sería la preferible. Pero en ocasiones, para evitar problemas numéricos (por ejemplo, en el caso de matrices definidas positivas, i.e. con autovalores nulos) puede ser más adecuado emplear la factorización espectral. En el primer caso el algoritmo sería el siguiente:

Algoritmo 6.2 (de simulación de una normal multivariante)

1. Obtener la factorización de Cholesky $\Sigma = LL^t$.
2. Simular $\mathbf{Z} = (Z_1, Z_2, \ldots, Z_d)$ i.i.d. $\mathcal{N}(0, 1)$.
3. Hacer $\mathbf{X} = \mu + L\mathbf{Z}$.
4. Repetir los pasos 2 y 3 las veces necesarias.

Nota: Hay que tener en cuenta el resultado del algoritmo empleado para la factorización de Cholesky. Por ejemplo si se obtiene $\Sigma = U^tU$, hará que emplear $L = U^t$.

Ejemplo 6.4 (simulación de datos funcionales o temporales)
Supongamos que el objetivo es generar una muestra de tamaño `nsim` de la variable funcional:

$$X(t) = \sin(2\pi t) + \varepsilon(t)$$

con $0 \leq t \leq 1$ y $Cov(\varepsilon(t_1), \varepsilon(t_2)) = e^{-\|t_1 - t_2\|}$, considerando 100 puntos de discretización (se puede pensar también que es un proceso temporal).

```
nsim <- 20
n <- 100
t <- seq(0, 1, length = n)
# Media
mu <- sin(2*pi*t)
# Covarianzas
t.dist <- as.matrix(dist(t))
x.cov <- exp(-t.dist)
```

Para la factorización de la matriz de covarianzas emplearemos la función `chol` del paquete base de R (si las dimensiones fueran muy grandes podría ser preferible emplear otros paquetes, e.g. `spam::chol.spam`), pero al devolver la matriz triangular superior habrá que transponer el resultado:

```
U <- chol(x.cov)
L <- t(U)
```

Si queremos simular una realización:

```
set.seed(1)
head(mu + L %*% rnorm(n))
```

```
##          [,1]
## 1 -0.6264538
## 2 -0.5307633
## 3 -0.5797968
## 4 -0.2844357
## 5 -0.1711797
## 6 -0.2220796
```

y para simular `nsim`:

```
z <- matrix(rnorm(nsim * n), nrow = n)
x <- mu + L %*% z

matplot(t, x, type = "l", ylim = c(-3.5, 3.5))
lines(t, mu, lwd = 2)
```

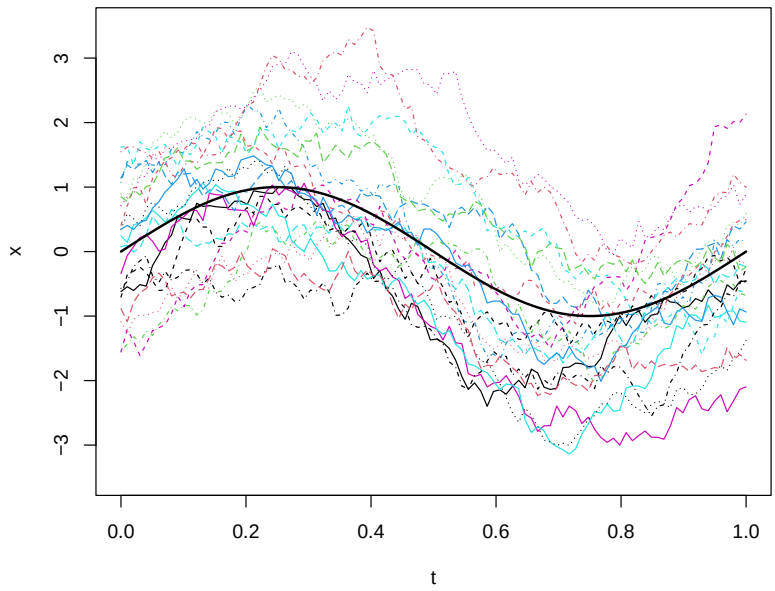


Figura 6.2: Realizaciones del proceso funcional del Ejemplo 6.4, obtenidas a partir de la factorización de Cholesky.

Alternativamente se podría emplear, por ejemplo, la funcion `mvrnorm` del paquete `MASS` que emplea la factorización espectral (`eigen`) (y que tiene en cuenta una tolerancia relativa para correguir autovalores negativos próximos a cero):

```
library(MASS)
mvrnorm
```

```
## function (n = 1, mu, Sigma, tol = 1e-06, empirical = FALSE, EISPACK = FALSE)
## {
```

```
##     p <- length(mu)
##     if (!all(dim(Sigma) == c(p, p)))
##         stop("incompatible arguments")
##     if (EISPACK)
##         stop("'EISPACK' is no longer supported by R", domain = NA)
##     eS <- eigen(Sigma, symmetric = TRUE)
##     ev <- eS$values
##     if (!all(ev >= -tol * abs(ev[1L])))
##         stop("'Sigma' is not positive definite")
##     X <- matrix(rnorm(p * n), n)
##     if (empirical) {
##         X <- scale(X, TRUE, FALSE)
##         X <- X %*% svd(X, nu = 0)$v
##         X <- scale(X, FALSE, TRUE)
##     }
##     X <- drop(mu) + eS$vectors %*% diag(sqrt(pmax(ev, 0)), p) %*%
##         t(X)
##     nm <- names(mu)
##     if (is.null(nm) && !is.null(dn <- dimnames(Sigma)))
##         nm <- dn[[1L]]
##     dimnames(X) <- list(nm, NULL)
##     if (n == 1)
##         drop(X)
##     else t(X)
## }
## <bytecode: 0x0000000038cfc9b8>
## <environment: namespace:MASS>
```

```
x <- mvrnorm(nsim, mu, x.cov)

matplot(t, t(x), type = "l")
lines(t, mu, lwd = 2)
```

Otros métodos para variables continuas relacionados con la factorización de la matriz de covarianzas son el método FFT (transformada rápida de Fourier; e.g. Davies y Harte, 1987) o el *Circular embedding* (Dietrich and Newsam, 1997), que realmente son el mismo.

6.4 Método de las distribuciones condicionadas

Teniendo en cuenta que:

$$f(x_1, x_2, \dots, x_d) = f_1(x_1) \cdot f_2(x_2|x_1) \cdots f_d(x_d|x_1, x_2, \dots, x_{d-1}),$$

donde las densidades condicionales pueden obtenerse a partir de las correspondientes marginales:

$$f_i(x_i|x_1, x_2, \dots, x_{i-1}) = \frac{f_{1,\dots,i}(x_1, x_2, \dots, x_i)}{f_{1,\dots,i-1}(x_1, x_2, \dots, x_{i-1})},$$

se obtiene el siguiente algoritmo general:

Algoritmo 6.3 (de simulación mediante distribuciones condicionadas)

1. Generar $X_1 \sim f_1$.
2. Desde $i = 2$ hasta d generar $X_i \sim f_i(\cdot|X_1, X_2, \dots, X_{i-1})$.
3. Devolver $\mathbf{X} = (X_1, X_2, \dots, X_d)$.

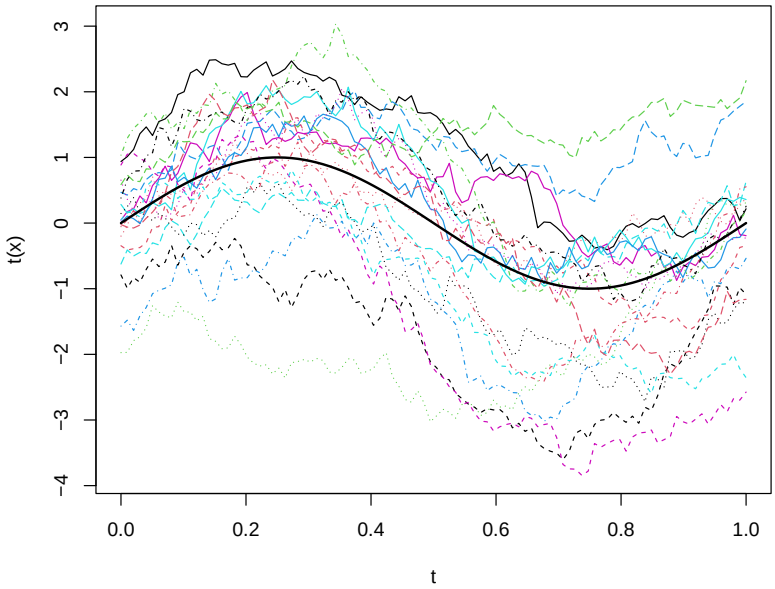


Figura 6.3: Realizaciones del proceso funcional del Ejemplo 6.4, obtenidas empleando la función `MASS::mvrnorm`.

Nota: En las simulaciones unidimensionales se puede tener en cuenta que $f_i\left(x_i|x_1,x_2,\dots,x_{i-1}\right) \propto f_{1,\dots,i}\left(x_1,x_2,\dots,x_i\right)$.

Ejemplo 6.5 (distribución uniforme en el círculo unitario)
Se trata de la distribución bidimensional continua con densidad constante en el círculo:

$$C = \left\{(x_1,x_2) \in \mathbb{R}^2 : x_1^2 + x_2^2 \leq 1\right\}.$$

Su función de densidad viene dada por:

$$f\left(x_1,x_2\right) = \begin{cases} \frac{1}{\pi} & \text{si } \left(x_1,x_2\right) \in C \\ 0 & \text{si } \left(x_1,x_2\right) \notin C \end{cases}$$

La densidad marginal de la primera variable resulta:

$$f_1\left(x_1\right) = \int_{-\sqrt{1-x_1^2}}^{+\sqrt{1-x_1^2}} \frac{1}{\pi} dx_2 = \frac{2\sqrt{1-x_1^2}}{\pi} \text{ si } x_1 \in [-1,1],$$

es decir:

$$f_1\left(x_1\right) = \begin{cases} \frac{2}{\pi}\sqrt{1-x_1^2} & \text{si } x_1 \in [-1,1] \\ 0 & \text{si } x_1 \notin [-1,1] \end{cases}$$

Además:

$$f_2\left(x_2|x_1\right) = \frac{f\left(x_1,x_2\right)}{f_1\left(x_1\right)} = \frac{\frac{1}{\pi}}{\frac{2\sqrt{1-x_1^2}}{\pi}} = \frac{1}{2\sqrt{1-x_1^2}}, \text{ si } x_2 \in \left[-\sqrt{1-x_1^2}, \sqrt{1-x_1^2}\right]$$

valiendo cero en otro caso. Se tiene entonces que:

$$X_2|X_1 \sim \mathcal{U}\left(-\sqrt{1-X_1^2}, \sqrt{1-X_1^2}\right),$$

siempre que $X_1 \in [-1,1]$.

Finalmente, el algoritmo resulta:

1. Simular X_1 con densidad $f_1(x_1) = \frac{2}{\pi}\sqrt{1-x_1^2}1_{\{|x_1|\leq 1\}}$.
2. Simular X_2 con densidad $\mathcal{U}\left(-\sqrt{1-X_1^2}, \sqrt{1-X_1^2}\right)$.
3. Devolver $\mathbf{X} = (X_1, X_2)^t$.

Para el paso 1 puede utilizarse, por ejemplo, el método de aceptación/rechazo, pues se trata de una densidad acotada definida en un intervalo acotado.

Ejemplo 6.6 (distribución normal bidimensional)
En el caso de una distribución normal bidimensional:

$$\mathbf{X} = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix} \sim \mathcal{N}\left(\begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \begin{pmatrix} \sigma_1^2 & \sigma_1\sigma_2\rho \\ \sigma_1\sigma_2\rho & \sigma_2^2 \end{pmatrix}\right)$$

tendríamos que:

$$\begin{aligned} f(x_1,x_2) = & \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \\ & \exp\left(-\frac{1}{2(1-\rho^2)}\left(\frac{(x_1-\mu_1)^2}{\sigma_1^2} + \frac{(x_2-\mu_2)^2}{\sigma_2^2} - \frac{2\rho(x_1-\mu_1)(x_2-\mu_2)}{\sigma_1\sigma_2}\right)\right) \end{aligned}$$

de donde se deduce (ver e.g. Cao, 2002, p.88; o ecuaciones (6.1) y (6.2) en la Sección 6.5.1) que:

$$f_1(x_1) = \frac{1}{\sigma_1\sqrt{2\pi}}\exp\left(-\frac{(x_1-\mu_1)^2}{2\sigma_1^2}\right)$$

$$\begin{aligned} f_2(x_2|x_1) &= \frac{f(x_1,x_2)}{f_1(x_1)} \\ &= \frac{1}{\sigma_2\sqrt{2\pi(1-\rho^2)}}\exp\left(-\frac{\left(x_2-\mu_2-\frac{\sigma_2}{\sigma_1}\rho(x_1-\mu_1)\right)^2}{2\sigma_2^2(1-\rho^2)}\right) \end{aligned}$$

Por tanto:

$$\begin{aligned} X_1 &\sim \mathcal{N}(\mu_1, \sigma_1^2) \\ X_2|X_1 &\sim \mathcal{N}\left(\mu_2 + \frac{\sigma_2}{\sigma_1}\rho(X_1-\mu_1), \sigma_2^2(1-\rho^2)\right) \end{aligned}$$

y el algoritmo sería el siguiente:

Algoritmo 6.4 (de simulación de una normal bidimensional)

1. Simular $Z_1, Z_2 \sim \mathcal{N}(0,1)$ independientes.
2. Hacer $X_1 = \mu_1 + \sigma_1 Z_1$.
3. Hacer $X_2 = \mu_2 + \sigma_2\rho Z_1 + \sigma_2\sqrt{1-\rho^2}Z_2$.

Este algoritmo es el mismo que obtendríamos con la factorización de la matriz de covarianzas ya que $\Sigma = LL^t$ con:

$$L = \begin{pmatrix} \sigma_1^2 & 0 \\ \sigma_2\rho & \sigma_2\sqrt{1-\rho^2} \end{pmatrix}$$

Además, esta aproximación puede generalizarse al caso multidimensional, ver Sección 6.5.1.

Ejercicio 6.2

Considerando la variable aleatoria bidimensional del Ejemplo 6.2 y teniendo en cuenta que la densidad marginal de la variable X es:

$$f_X(x) = \begin{cases} \frac{1}{8}\left(5-3x^2\right) & \text{si } x \in [-1,1] \\ 0 & \text{en otro caso} \end{cases}$$

Describir brevemente un algoritmo para la simulación del vector aleatorio basado en el método de las distribuciones condicionadas (asumir que se dispone de un algoritmo para generar observaciones de las distribuciones unidimensionales de interés).

6.5 Simulación condicional e incondicional

En ocasiones en inferencia estadística interesa la simulación condicional de nuevos valores de forma que se preserven los datos observados, para lo que se suele emplear el algoritmo anterior partiendo de la muestra observada:

Algoritmo 6.5 (de simulación condicional)

1. Obtener la distribución condicional (correspondiente al punto que se quiere simular) dada la muestra y los valores simulados anteriormente.
2. Simular un valor de la distribución condicional.
3. Agregar este valor al conjunto de datos y volver al paso 1.

En el caso de normalidad, en lugar de simular punto a punto, podemos obtener fácilmente la distribución condicionada para simular los valores de forma conjunta.

6.5.1 Simulación condicional de una normal multivariante

Si $\mathbf{X} \sim \mathcal{N}_d(\mu, \Sigma)$ es tal que $\mathbf{X}$, μ y Σ se particionan de la forma:

$$\mathbf{X} = \begin{pmatrix} \mathbf{X}_1 \\ \mathbf{X}_2 \end{pmatrix}, \mu = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix}, \Sigma = \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix},$$

suponiendo que $\mathbf{X}_1$ se corresponde con los valores observados y $\mathbf{X}_2$ con los que se pretende simular, entonces puede verse (e.g. Ripley, 1987) que la distribución de $\mathbf{X}_2|\mathbf{X}_1$ es normal con:

$$E\left(\mathbf{X}_2|\mathbf{X}_1\right) = \mu_2 + \Sigma_{21}\Sigma_{11}^{-1}\left(\mathbf{X}_1 - \mu_1\right), \tag{6.1}$$

$$Cov\left(\mathbf{X}_2|\mathbf{X}_1\right) = \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}. \tag{6.2}$$

Nota: La ecuación (6.1) coincide con la expresión de la predicción lineal óptima de $\mathbf{X}_2$ a partir de $\mathbf{X}_1$ con media y varianzas conocidas (denominado predictor del kriging simple en estadística espacial, y la diagonal de (6.2) son las correspondientes varianzas kriging).

Ejemplo 6.7 (simulación condicional de datos funcionales o temporales)
Continuando con el Ejemplo 6.4 anterior, consideramos los primeros valores de una simulación incondicional como los datos:

```
# Indice correspondiente a los datos observados
# idata <- t < 0.5 # Extrapolación
idata <- t < 0.2 | t > 0.8 # Interpolación
mu1 <- mu[idata]
n1 <- length(mu1)
cov.data <- x.cov[idata, idata]
U <- chol(cov.data)
# Simulación incondicional:
set.seed(1)
data <- drop(mu1 + t(U) %*% rnorm(n1))
y <- rep(NA, n)
y[idata] <- data
```

Para obtener la simulación condicional en los puntos de predicción, calculamos la correspondiente media y matriz de covarianzas condicionadas:

```
mu2 <- mu[!idata]
n2 <- length(mu2)
cov.pred <- x.cov[!idata, !idata]
cov.preddat <- x.cov[!idata, idata]
# Cálculo de los pesos kriging:
cov.data.inv <- chol2inv(U)
lambda <- cov.preddat %*% cov.data.inv
# La media serán las predicciones del kriging simple:
kpred <- mu2 + drop(lambda %*% (data - mu1))
# Varianza de la distribución condicional
kcov <- cov.pred -  lambda %*% t(cov.preddat)
# (La diagonal serán las varianzas kriging).
```

Finalmente generamos las simulaciones condicionales empleando el método de factorización de la matriz de covarianzas (Figura 6.4):

```
z <- matrix(rnorm(nsim * n2), nrow = n2)
ycond <- kpred + t(chol(kcov)) %*% z
# Representación gráfica:
#   Media teórica
plot(t, mu, type = "l", lwd = 2, ylab = "y", ylim = c(-3.5, 3.5))
#   Datos
lines(t, y) # lines(t[idata], data)
#   Media condicional (predicción kriging)
lines(t[!idata], kpred, lwd = 2, lty = 2)
#   Simulaciones condicionales
matplot(t[!idata], ycond, type = "l", lty = 3, add = TRUE)
```

Ejemplo 6.8 (simulación condicional de datos espaciales)
Consideramos un proceso espacial bidimensional normal $Z(\mathbf{s}) \equiv Z(x, y)$ de media 0 y covariograma exponencial:

$$Cov(Z(\mathbf{s}_1), Z(\mathbf{s}_2)) = C(\|\mathbf{s}_1 - \mathbf{s}_2\|) = e^{-\|\mathbf{s}_1 - \mathbf{s}_2\|}.$$

En primer lugar, obtendremos una simulación del proceso en las posiciones $\{(0,0), (0,1), (1,0), (1,1)\}$ que será considerada posteriormente como los datos observados. Empleando las herramientas del paquete `geoR`, resulta muy fácil obtener una simulación incondicional en una rejilla en el cuadrado unidad mediante la función `grf`:

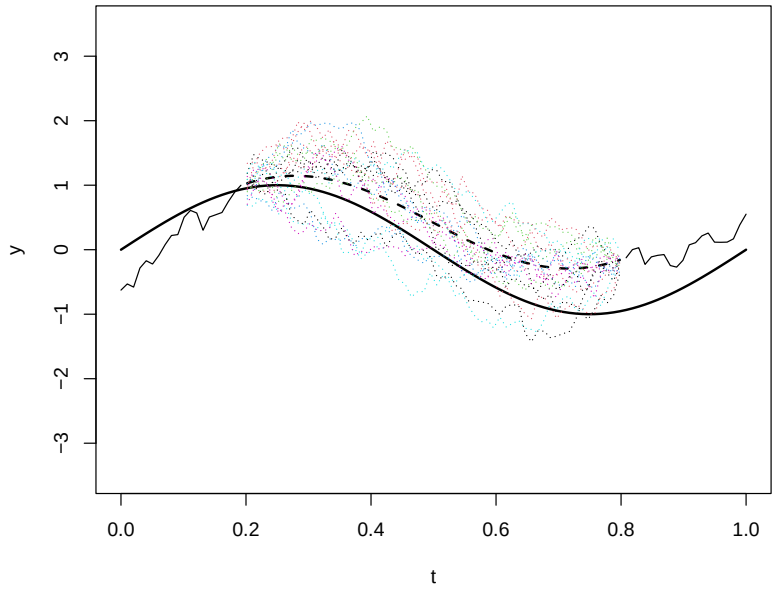


Figura 6.4: Realizaciones condicionales (líneas de puntos) del proceso funcional del Ejemplo 6.7 (datos: línea continua, tendencia/media incondicional: linea gruesa continua, predicción/media condicional: linea gruesa discontinua).

```
library(geoR)
n <- 4
set.seed(1)
z <- grf(n, grid = "reg", cov.pars = c(1, 1))
```

```
## grf: generating grid  2  *  2  with  4  points
## grf: process with  1  covariance structure(s)
## grf: nugget effect is: tausq= 0
## grf: covariance model 1 is: exponential(sigmasq=1, phi=1)
## grf: decomposition algorithm used is:  cholesky
## grf: End of simulation procedure. Number of realizations: 1
```

```
# names(z)
z$coords
```

```
##      x y
## [1,] 0 0
## [2,] 1 0
## [3,] 0 1
## [4,] 1 1
```

```
z$data
```

```
## [1] -0.62645381 -0.05969442 -0.98014198  1.09215113
```

La `grf` función emplea por defecto el método de la factorización de la matriz de covarianzas, sin embargo, si se desean obtener múltiples realizaciones, en lugar de llamar repetidamente a esta función (lo que implicaría factorizar repetidamente la matriz de covarianzas), puede ser preferible emplear un código similar al siguiente (de forma que solo se realiza una vez dicha factorización, y suponiendo además que no es necesario conservar las distintas realizaciones):

```
# Posiciones datos
nx <- c(2, 2)
data.s <- expand.grid(x = seq(0, 1, len = nx[1]), y = seq(0, 1, len = nx[2]))
# plot(data.s, type = "p", pch = 20, asp = 1) # Representar posiciones

# Matriz de varianzas covarianzas
cov.matrix <- varcov.spatial(coords=data.s, cov.pars=c(1,1))$varcov
cov.matrix
```

```
##           [,1]      [,2]      [,3]      [,4]
## [1,] 1.0000000 0.3678794 0.3678794 0.2431167
## [2,] 0.3678794 1.0000000 0.2431167 0.3678794
## [3,] 0.3678794 0.2431167 1.0000000 0.3678794
## [4,] 0.2431167 0.3678794 0.3678794 1.0000000
```

```
# Simular valores
set.seed(1)
L <- t(chol(cov.matrix))

# Bucle simulación
nsim <- 1 # 1000
for (i in 1:nsim) {
  y <- L %*% rnorm(n)
  # calcular estadísticos, errores,...
}
y
```

```
##             [,1]
## [1,] -0.62645381
## [2,] -0.05969442
## [3,] -0.98014198
## [4,]  1.09215113
```

Para generar simulaciones condicionales podemos emplear la función `krige.conv`. Por ejemplo, para generar 4 simulaciones en la rejilla regular 10×10 en el cuadrado unidad $[0,1] \times [0,1]$ condicionadas a los valores generados en el apartado anterior podríamos emplear el siguiente código:

```
# Posiciones simulación condicional
new.nx <- c(20, 20)
new.x <- seq(0, 1, len = new.nx[1])
new.y <- seq(0, 1, len = new.nx[2])
new.s <- expand.grid(x = new.x, y = new.y)
plot(data.s, type = "p", pch = 20, asp = 1)
points(new.s)
```

```
# Simulación condicional
set.seed(1)
nsim.cond <- 4
s.out <- output.control(n.predictive = nsim.cond)
kc <- krige.conv(z, loc = new.s, output = s.out,
                 krige = krige.control(type.krige="SK", beta = 0, cov.pars = c(1, 1)))
```

```
## krige.conv: results will be returned only for prediction locations inside the borders
## krige.conv: model with constant mean
## krige.conv: sampling from the predictive distribution (conditional simulations)
## krige.conv: Kriging performed using global neighbourhood
```

Si las representamos podemos confirmar que los valores en las posiciones $\{(0,0),(0,1),(1,0),(1,1)\}$ coinciden con los generados anteriormente.

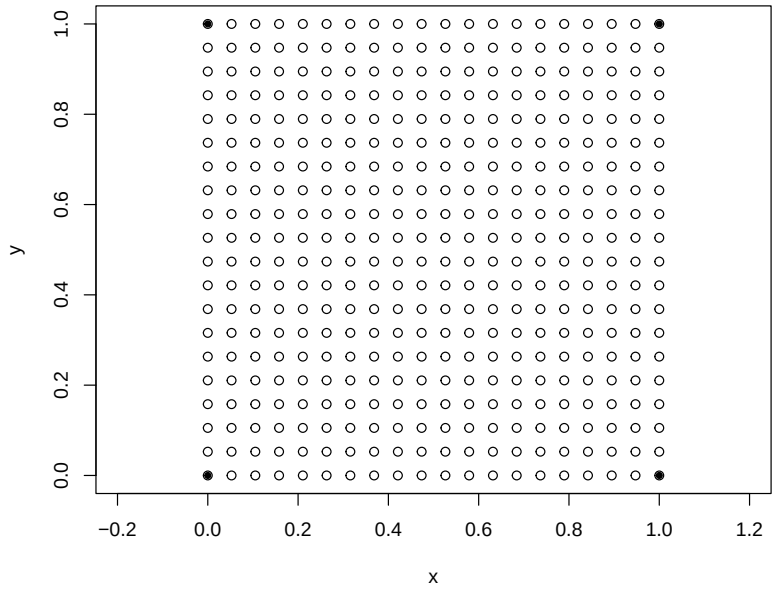
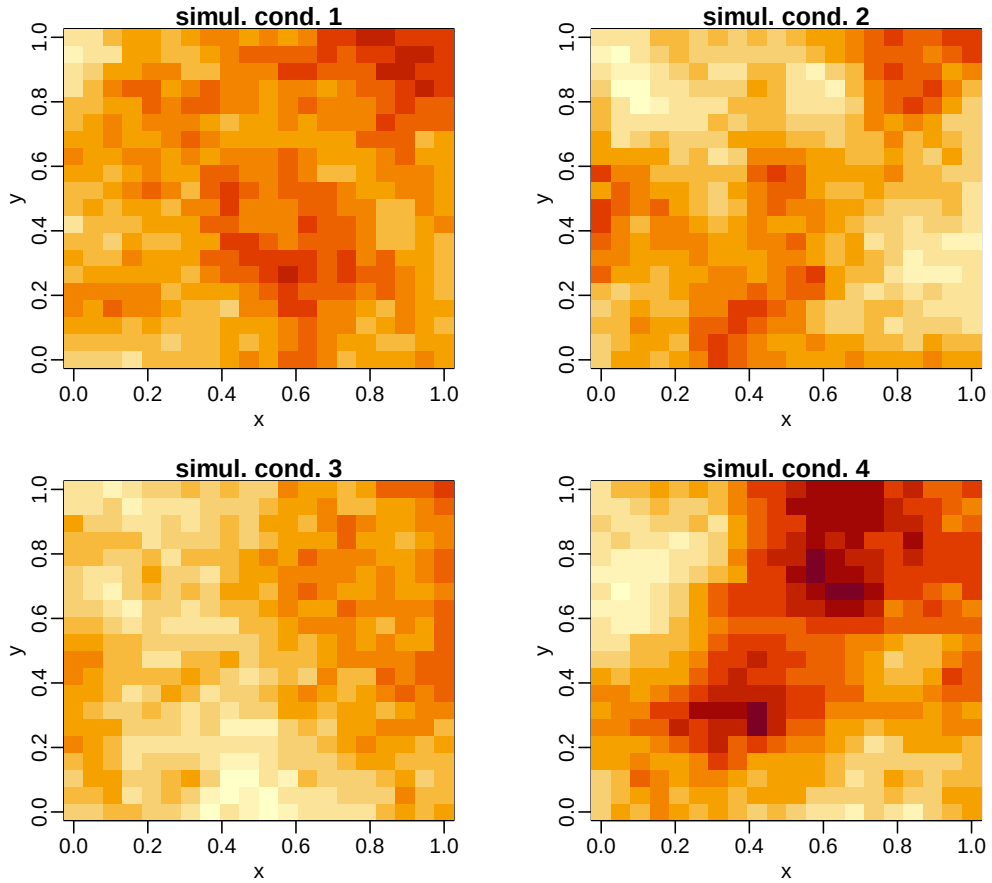


Figura 6.5: Posiciones espaciales de las simulaciones condicionales (y las de los datos).

```
# Generar gráficos
par.old <- par(mfrow = c(2, 2), mar = c(3.5, 3.5, 1, 2), mgp = c(1.5, .5, 0))
zlim <- range(kc$simul)     # Escala común
# La versión actual de geoR::image.kriging() no admite múltiples gráficos en una ventana
# image(kc, val=kc$simul[,1], main="simul. cond. 1", zlim=zlim)
# image(kc, val=kc$simul[,2], main="simul. cond. 2", zlim=zlim)
# image(kc, val=kc$simul[,3], main="simul. cond. 3", zlim=zlim)
# image(kc, val=kc$simul[,4], main="simul. cond. 4", zlim=zlim)
dim(kc$simul) <- c(new.nx, nsim.cond)
image(new.x, new.y, kc$simul[,,1], main="simul. cond. 1",
      xlab = "x", ylab = "y", zlim = zlim)
image(new.x, new.y, kc$simul[,,2], main="simul. cond. 2",
      xlab = "x", ylab = "y", zlim = zlim)
image(new.x, new.y, kc$simul[,,3], main="simul. cond. 3",
      xlab = "x", ylab = "y", zlim = zlim)
image(new.x, new.y, kc$simul[,,4], main="simul. cond. 4",
      xlab = "x", ylab = "y", zlim = zlim)
```




```
par(par.old)
```

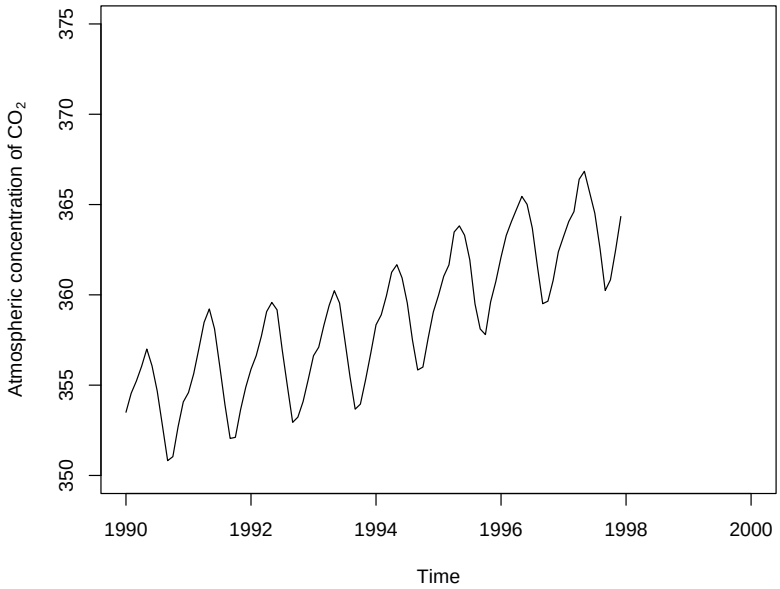
6.5.2 Simulación condicional a partir de un modelo ajustado

En la práctica normalmente se ajusta un modelo a los datos observados y posteriormente se obtienen las simulaciones condicionadas empleando el modelo ajustado (esto también se denomina bootstrap paramétrico condicional).

En R se incluye una función genérica[1] `simulate()` que permite generar respuestas a partir de modelos ajustados (siempre que esté implementado el método correspondiente al tipo de modelo). Los métodos para modelos lineales y modelos lineales generalizamos están implementados en el paquete base `stats`. Muchos otros paquetes que proporcionan modelos adicionales, implementan también los correspondientes métodos `simulate()`. Por ejemplo, en el caso de series de tiempo, el paquete `forecast` permite ajustar distintos tipos de modelos y generar simulaciones a partir de ellos:

```
library(forecast)
data <- window(co2, 1990) # datos de co2 desde 1990
plot(data, ylab = expression("Atmospheric concentration of CO"[2]),
     xlim = c(1990, 2000), ylim = c(350, 375))
```

[1]Se pueden implementar métodos específicos para cada tipo (clase) de objeto; en este caso para cada tipo de modelo ajustado y podemos mostrar los disponibles mediante el comando `methods(simulate)`.




```
# Se podrían ajustar distintos tipos de modelos
fit <- ets(data)
# fit <- auto.arima(data)
```

Podemos obtener predicciones (media de la distribución condicional) e intervalos de predicción:

```
pred <- forecast(fit, h = 24, level = 95)
pred
```

```
##          Point Forecast    Lo 95    Hi 95
## Jan 1998       365.1118 364.5342 365.6894
## Feb 1998       366.1195 365.4572 366.7819
## Mar 1998       367.0161 366.2786 367.7535
## Apr 1998       368.2749 367.4693 369.0806
## May 1998       368.9282 368.0596 369.7968
## Jun 1998       368.2240 367.2967 369.1513
## Jul 1998       366.5823 365.5997 367.5649
## Aug 1998       364.4895 363.4546 365.5244
## Sep 1998       362.6586 361.5738 363.7434
## Oct 1998       362.7805 361.6479 363.9130
## Nov 1998       364.2045 363.0262 365.3829
## Dec 1998       365.5250 364.3025 366.7476
## Jan 1999       366.6002 365.3349 367.8654
## Feb 1999       367.6078 366.3013 368.9144
## Mar 1999       368.5044 367.1578 369.8510
## Apr 1999       369.7633 368.3777 371.1488
## May 1999       370.4165 368.9930 371.8400
## Jun 1999       369.7124 368.2519 371.1728
## Jul 1999       368.0706 366.5741 369.5671
## Aug 1999       365.9778 364.4461 367.5096
## Sep 1999       364.1469 362.5806 365.7131
## Oct 1999       364.2688 362.6688 365.8688
## Nov 1999       365.6929 364.0597 367.3260
## Dec 1999       367.0134 365.3477 368.6790
```

Para análisis adicionales nos puede interesar generar simulaciones (por defecto de la distribución condicional, `future = TRUE`):

```
set.seed(321)
sim.cond <- simulate(fit, 24)

plot(pred)
lines(sim.cond, lwd = 2, col = "red")
```

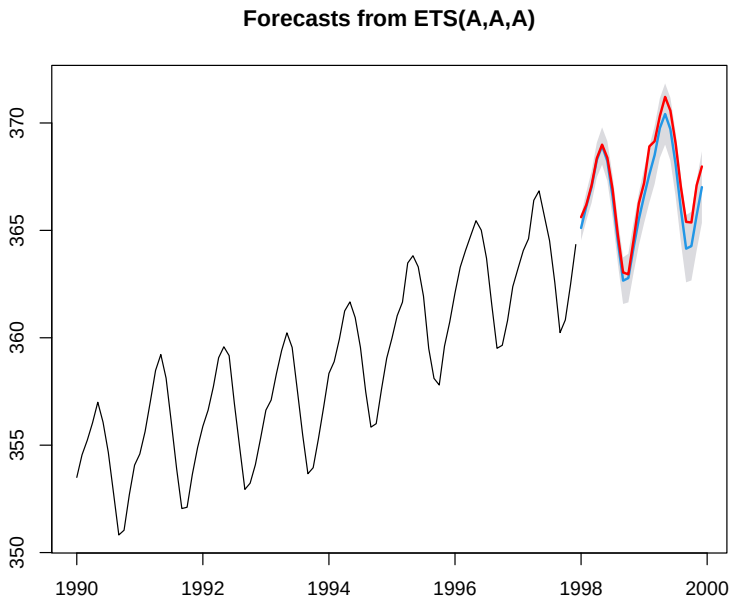


Figura 6.6: Ejemplo de una serie de tiempo (datos observados de co2 en el observatorio Mauna Loa), predicciones futuras (en azul; media distribución condicional) y simulación condicional (en rojo) obtenidas a partir de un modelo ajustado.

Para más detalles ver Hyndman y Athanasopoulos (2018, secciones 4.3 y 11.4).

6.6 Simulación basada en cópulas

Una cópula es una función de distribución multidimensional con distribuciones marginales uniformes (e.g. Nelsen, 2006; Hofert, 2018). Se emplean principalmente para la construcción de distribuciones multivariantes a partir de distribuciones marginales (también en análisis de dependencia y medidas de asociación). La idea es que la estructura de dependencia no depende de las distribuciones marginales.

Por simplicidad nos centraremos en el caso bidimensional. El teorema central en la teoría de cópulas es el teorema de Sklar (1959), que en este caso es:

Teorema 6.1 (de Sklar caso bidimensional)

Si (X, Y) es una variable aleatoria bidimensional con función de distribución conjunta $F(\cdot,\cdot)$ y distribuciones marginales $F_1(\cdot)$ y $F_2(\cdot)$ respectivamente, entonces existe una cópula $C(\cdot,\cdot)$ tal que:

$$F(x, y) = C\left(F_1(x), F_2(y)\right), \quad \forall x, y \in \mathbb{R}.$$

Además, si $F_1(\cdot)$ y $F_2(\cdot)$ son continuas entonces $C(\cdot,\cdot)$ es única. Siendo el recíproco también cierto.

6.6.1 Cópulas Arquimedianas

Además de las cópulas Gausianas, es una de las familias de cópulas más utilizadas. Son de la forma:

$$C(x_1, x_2, \dots, x_d) = \Psi^{-1}\left(\sum_{i=1}^{d} \Psi\left(F_i(x_i)\right)\right),$$

siendo Ψ su función generadora.

Una condición suficiente para que sea una cópula multidimensional válida es que $\Psi(1) = 0$, $\lim_{x \to 0} \Psi(x) = \infty$, $\Psi'(x) < 0$ y $\Psi''(x) > 0$.

Ejemplos:

- Cópula producto o independiente: $\Psi(x) = -\ln(x)$,

$$F(x, y) = F_1(x)F_2(y).$$

- Cópula de Clayton: $\Psi(x) = \frac{1}{\alpha}\left(x^{-\alpha} - 1\right); \alpha > 0$,

$$F(x, y) = (F_1(x)^{-\alpha} + F_2(y)^{-\alpha} - 1)^{-1/\alpha}.$$

- Cópula de Gumbel: $\Psi(x) = (-\ln(x))^{\alpha}; \alpha \geq 1$

6.6.2 Simulación

Las cópulas pueden facilitar notablemente la simulación de la distribución conjunta. Si $(U, V) \sim C(\cdot, \cdot)$ (marginales uniformes):

$$(F_1^{-1}(U), F_2^{-1}(V)) \sim F(\cdot, \cdot)$$

En la mayoría de los casos se dispone de expresiones explicitas de $C_u(v) \equiv C_2\left(v|\, u\right)$ y de su inversa $C_u^{-1}(w)$, por lo que se puede generar (U, V) fácilmente mediante el método secuencial de distribuciones condicionadas descrito en la Sección 6.4.

Algoritmo 6.6 (de simulación bidimensional mediante cópulas)

1. Generar $U, W \sim \mathcal{U}(0, 1)$
2. Obtener $V = C_U^{-1}(W)$
3. Devolver $\left(F_1^{-1}(U), F_2^{-1}(V)\right)$

Ejemplo 6.9 (Cópula bidimensional de Clayton)

Consideramos una variable aleatoria bidimensional con distribuciones marginales uniformes y distribución bidimensional determinada por la cópula de Clayton.

Teniendo en cuenta que en este caso:

$$C_u^{-1}(w) \equiv \left(u^{-\alpha}\left(w^{-\frac{\alpha}{\alpha+1}} - 1\right) + 1\right)^{-\frac{1}{\alpha}},$$

la siguiente función permitiría generar una muestra de tamaño n de esta distribución:

```
rcclayton <- function(alpha, n) {
  val <- cbind(runif(n), runif(n))
  val[, 2] <- (val[, 1]^(-alpha) *
               (val[, 2]^(-alpha/(alpha + 1)) - 1) + 1)^(-1/alpha)
  return(val)
}
```

Utilizando esta función generamos una muestra de tamaño 10000 y representamos gráficamente los valores obtenidos [Figura 6.7]:

```
set.seed(54321)
rcunif <- rcclayton(2, 10000)
plot(rcunif, xlab = "u", ylab = "v")
```

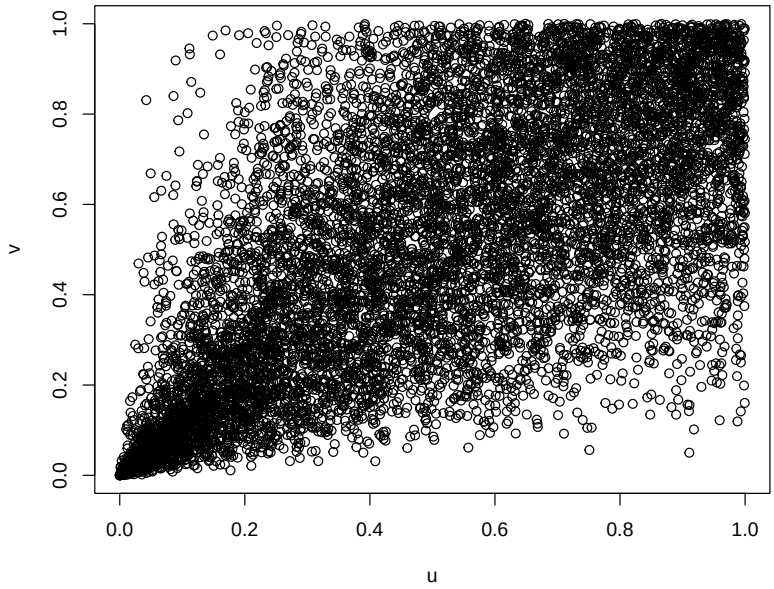


Figura 6.7: Gráfico de dispersión de los valores generados con distribución bidimensional de Clayton.

Podemos representar la densidad conjunta (con `sm::sm.density()`) [Figura 6.8]:

```
# if(!require(sm)) stop('Required package `sm` not installed.')
sm::sm.density(rcunif, xlab = "u", ylab = "v", zlab = "Density")
```

y las distribuciones marginales [Figura 6.9]:

```
par.old <- par(mfrow = c(1, 2))
hist(rcunif[,1], freq = FALSE, xlab = "u")
abline(h = 1)
hist(rcunif[,2], freq = FALSE, xlab = "v")
abline(h = 1)
```

```
par(par.old)
```

Empleando el paquete *copula* [Figuras: 6.10, 6.11]:

```
if(!require(copula)) stop('Required pakage `copula` not installed.')
clayton.cop <- claytonCopula(2, dim = 2) # caso bidimensional
y <- rCopula(10000, clayton.cop)
plot(y, xlab = "u", ylab = "v")
```

```
clayton.cop <- claytonCopula(2, dim = 3) # caso tridimensional
y <- rCopula(10000, clayton.cop)
# scatterplot3d::scatterplot3d(y)
plot3D:::points3D(y[,1], y[,2], y[, 3], colvar = NULL,
                  xlab = "u1", ylab = "u2", zlab = "u3")
```

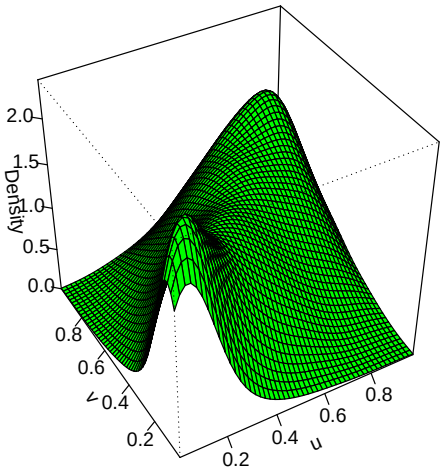


Figura 6.8: Densidad conjunta de los valores generados con distribución bidimensional de Clayton

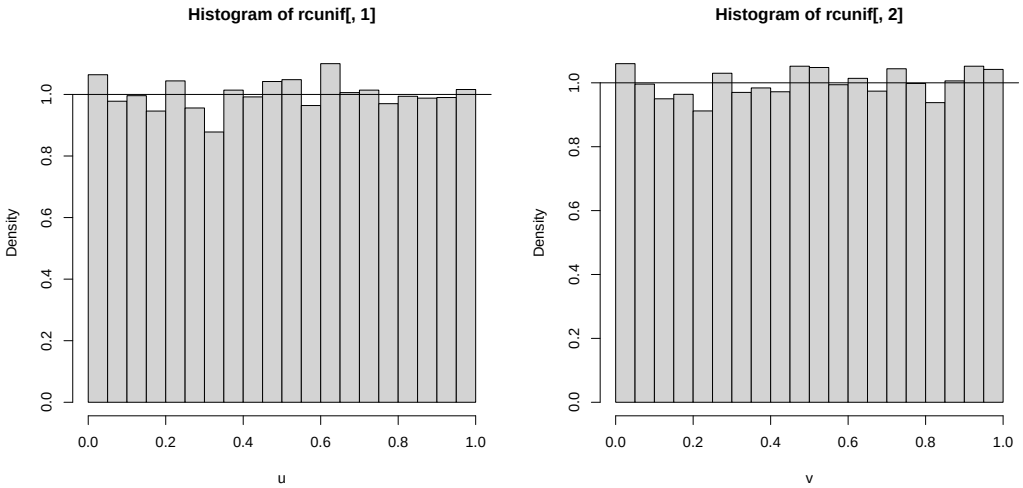


Figura 6.9: Distribuciones marginales de los valores generados con distribución bidimensional de Clayton.

Por ejemplo, podemos generar una muestra de una variable aleatoria bidimensional con distribuciones marginales exponenciales de parámetros 1 y 2, respectivamente (y distribución bidimensional determinada por la cópula de Clayton), transformando la muestra anterior [Figuras: 6.12, 6.13]:

```
rcexp <- cbind(qexp(rcunif[,1], 1), qexp(rcunif[,2], 2))
plot(rcexp, xlab = "exp1", ylab = "exp2")
```

```
# Distribuciones marginales
par.old <- par(mfrow = c(1, 2))
hist(rcexp[,1], freq = FALSE, xlab = "exp1")
curve(dexp(x, 1), add = TRUE)
```

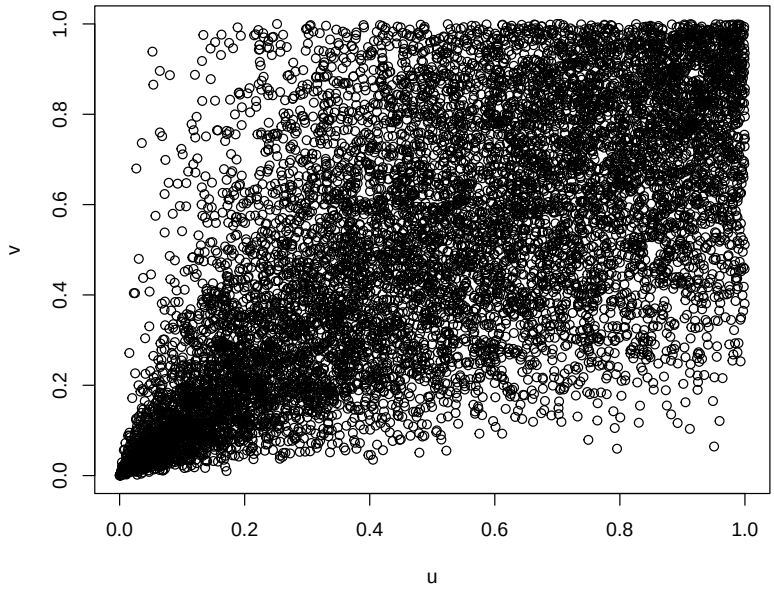


Figura 6.10: Gráfico de dispersión de los valores generados con distribución bidimensional de Clayton empleando el paquete ‘copula‘.

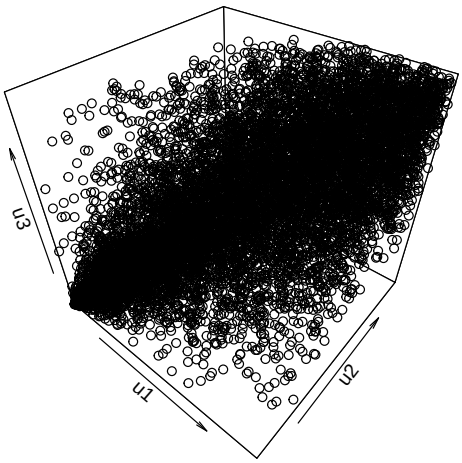


Figura 6.11: Gráfico de dispersión de los valores generados con distribución trididimensional de Clayton empleando el paquete ‘copula‘.

```
hist(rcexp[,2], freq = FALSE, xlab = "exp2")
curve(dexp(x, 2), add = TRUE)
```

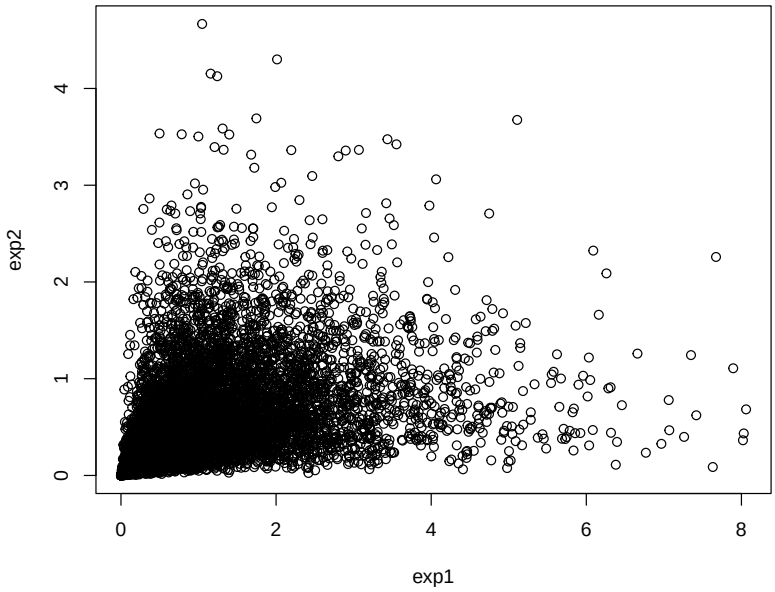


Figura 6.12: Gráfico de dispersión de los valores generados con distribución exponencial y dependencia definida por la cópula de Clayton.

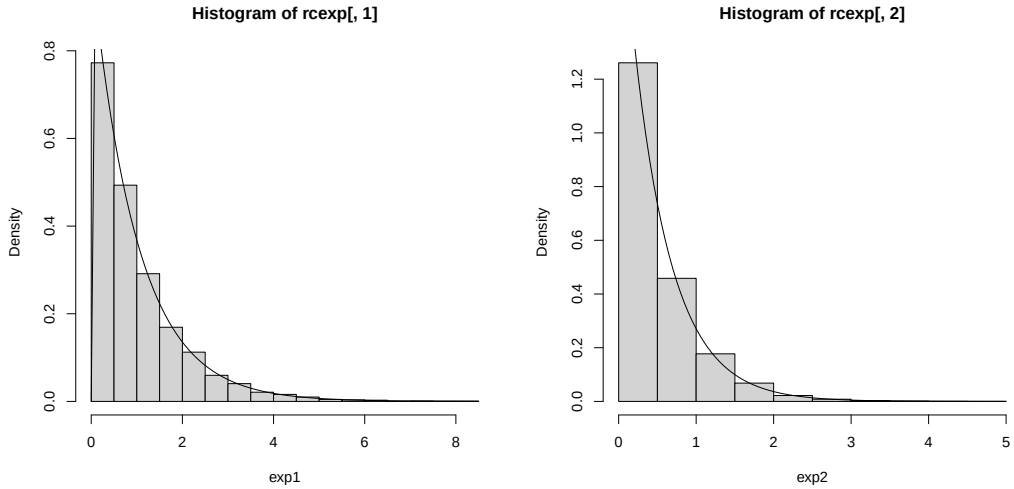


Figura 6.13: Distribuciones marginales exponenciales de los valores generados con dependencia definida por la cópula de Clayton.

```
par(par.old)
```

6.7 Simulación de distribuciones multivariantes discretas

En el caso de una distribución d-dimensional discreta el procedimiento habitual es simular una variable aleatoria discreta unidimensional equivalente. Este tipo de procedimientos son conocidos como métodos de etiquetado o codificación y la idea básica consistiría en construir un indice unidimensional equivalente al indice multidimensional, mediante una función de etiquetado $l(\mathbf{i}) = l\left(i_1, i_2, \ldots, i_d\right) \in \mathbb{N}$.

Si la variable discreta multidimensional tiene soporte finito, este tipo de recodificación se puede hacer de forma automática en R cambiando simplemente el indexado[2] (empleando la función `as.vector()` para cambiar a un indexado unidimensional y posteriormente las funciones `as.matrix()`, o `as.array()`, para reconstruir el indexado multidimensional).

Como ejemplo ilustrativo, en el caso bidimensional, podríamos emplear el siguiente código:

```
xy <- outer(10*1:4, 1:2, "+")
xy
```

```
##      [,1] [,2]
## [1,]   11   12
## [2,]   21   22
## [3,]   31   32
## [4,]   41   42
```

```
z <- as.vector(xy)
z
```

```
## [1] 11 21 31 41 12 22 32 42
```

```
xy <- matrix(z, ncol = 2)
xy
```

```
##      [,1] [,2]
## [1,]   11   12
## [2,]   21   22
## [3,]   31   32
## [4,]   41   42
```

Suponiendo que la primera componente toma I valores distintos, la función de etiquetado para calcular el índice unidimensional a partir de uno bidimensional sería:

$$l(i,j) = I(j-1) + i$$

```
nx <- nrow(xy)
l <- function(i, j) nx*(j-1) + i
z[l(2, 1)]
```

```
## [1] 21
```

```
z[l(3, 2)]
```

```
## [1] 32
```

Para recuperar el índice bidimensional se emplearía la inversa de la función de etiquetado:

$$l^{-1}(k) = \left((k-1) \bmod I + 1, \left\lfloor \frac{k-1}{I} \right\rfloor + 1\right)$$

```
linv <- function(k) cbind((k - 1) %% nx + 1, floor((k - 1)/nx) + 1)
# Para el caso multidimensional (1d => md): imd <- arrayInd(i1d, dim(xy))
xy[linv(2)]
```

```
## [1] 21
```

```
xy[linv(7)]
```

```
## [1] 32
```

[2] En R podemos obtener el índice multidimensional empleando la función `arrayInd(ind, .dim, ...)`, siendo `ind` un vector de índices unidimensionales.

Realmente lo que ocurre internamente en R es que un objeto `matrix` o `array` está almacenado como un vector y admite un indexado multidimensional si está presente un atributo `dim`:

```
dim(z) <- c(4, 2)
z
```

```
##      [,1] [,2]
## [1,]   11   12
## [2,]   21   22
## [3,]   31   32
## [4,]   41   42
```

```
dim(z) <- c(2, 2, 2)
z
```

```
## , , 1
##
##      [,1] [,2]
## [1,]   11   31
## [2,]   21   41
##
## , , 2
##
##      [,1] [,2]
## [1,]   12   32
## [2,]   22   42
```

```
dim(z) <- NULL
z
```

```
## [1] 11 21 31 41 12 22 32 42
```

Si la variable discreta multidimensional no tiene soporte finito, se podrían emplear métodos de codificación más avanzados (ver Cao, 2002, Sección 6.3), aunque no se podría guardar la función de masa de probabilidad en una tabla. No obstante, se podría emplear el indexado anterior si todas las componentes menos una (la correspondiente al índice `j` en el ejemplo anterior) toman un número finito de valores.

6.7.1 Simulación de una variable discreta bidimensional

Consideramos datos recogidos en un estudio de mejora de calidad en una fábrica de semiconductores. Se obtuvo una muestra de obleas que se clasificaron dependiendo de si se encontraron partículas en la matriz que producía la oblea y de si la calidad de oblea era buena (para más detalles ver Hall, 1994).

```
n <- c(320, 14, 80, 36)
particulas <- gl(2, 1, 4, labels = c("no", "si"))
calidad <- gl(2, 2, labels = c("buena", "mala"))
df <- data.frame(n, particulas, calidad)
df
```

```
##     n particulas calidad
## 1 320         no   buena
## 2  14         si   buena
## 3  80         no    mala
## 4  36         si    mala
```

En lugar de estar en el formato de un conjunto de datos (`data.frame`) puede que los datos estén en formato de tabla (`table`, `matrix`):

```
tabla <- xtabs(n ~ calidad + particulas)
tabla
```

```
##        particulas
## calidad  no  si
##   buena 320  14
##   mala   80  36
```

Lo podemos convertir directamente a `data.frame`:

```
as.data.frame(tabla)
```

```
##   calidad particulas Freq
## 1   buena         no  320
## 2    mala         no   80
## 3   buena         si   14
## 4    mala         si   36
```

En este caso estimamos[3] las probabilidades a partir de las frecuencias:

```
df$p <- df$n/sum(df$n)
df
```

```
##     n particulas calidad          p
## 1 320         no   buena 0.71111111
## 2  14         si   buena 0.03111111
## 3  80         no    mala 0.17777778
## 4  36         si    mala 0.08000000
```

En formato tabla:

```
pij <- tabla/sum(tabla)
pij
```

```
##        particulas
## calidad         no         si
##   buena 0.71111111 0.03111111
##   mala  0.17777778 0.08000000
```

Para simular la variable bidimensional consideramos una variable unidimensional de índices:

```
z <- 1:nrow(df)
z
```

```
## [1] 1 2 3 4
```

Con probabilidades:

```
pz <- df$p
pz
```

```
## [1] 0.71111111 0.03111111 0.17777778 0.08000000
```

Si las probabilidades estuviesen en una matriz, las convertiríamos a un vector con:

```
as.vector(pij)
```

```
## [1] 0.71111111 0.17777778 0.03111111 0.08000000
```

Si simulamos la variable unidimensional:

```
set.seed(1)
nsim <- 20
rz <- sample(z, nsim, replace = TRUE, prob = pz)
```

Podríamos obtener simulaciones bidimensionales, por ejemplo:

[3] Como ya se comentó, la simulación empleando un modelo estimado también se denomina bootstrap paramétrico.

```
etiquetas <- as.matrix(df[c('particulas', 'calidad')])
rxy <- etiquetas[rz, ]
head(rxy)
```

```
##      particulas calidad
## [1,] "no"       "buena"
## [2,] "no"       "buena"
## [3,] "no"       "buena"
## [4,] "si"       "mala"
## [5,] "no"       "buena"
## [6,] "si"       "mala"
```

Alternativamente, si queremos trabajar con data.frames:

```
etiquetas <- df[c('particulas', 'calidad')]
rxy <- etiquetas[rz, ]
head(rxy)
```

```
##     particulas calidad
## 1           no   buena
## 1.1         no   buena
## 1.2         no   buena
## 4           si    mala
## 1.3         no   buena
## 4.1         si    mala
```

```
# Si se quieren eliminar las etiquetas de las filas:
row.names(rxy) <- NULL
head(rxy)
```

```
##   particulas calidad
## 1         no   buena
## 2         no   buena
## 3         no   buena
## 4         si    mala
## 5         no   buena
## 6         si    mala
```

6.7.2 Simulación de tablas de contingencia

Podríamos emplear el código anterior para simular tablas de contingencia. En estos casos se suele mantener fijo el tamaño de la muestra, igual al total de la tabla[4]:

```
nsim <- sum(n)
set.seed(1)
rz <- sample(z, nsim, replace = TRUE, prob = pz)
rtable <- table(rz) # Tabla de frecuencias unidimensional
matrix(rtable, ncol = 2, dimnames = dimnames(tabla)) # Tabla de frecuencias bidimensional
```

```
##        particulas
## calidad  no si
##   buena 321 78
##   mala   15 36
```

Aunque puede ser preferible emplear directamente `rmultinom()` si se van a generar muchas:

[4]También se podría fijar una de las distribuciones de frecuencias marginales y simular las correspondientes distribuciones condicionadas.

```
ntsim <- 1000
rtablas <- rmultinom(ntsim, sum(n), pz)
rtablas[ , 1:5] # Las cinco primeras simulaciones
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  298  329  323  323  307
## [2,]   15   21    5   15   15
## [3,]   92   68   91   77   92
## [4,]   45   32   31   35   36
```

Ejemplo 6.10 (Distribución del estadístico chi-cuadrado de independencia)

En el contraste chi-cuadrado (tradicional) de independencia (ver Sección A.1.4) se emplea una aproximación continua de la distribución del estadístico bajo la hipótesis nula (la distribución asintótica χ^2, con una corrección por continuidad en tablas dos por dos):

```
res <- chisq.test(tabla)
res
```

```
##
##  Pearson's Chi-squared test with Yates' continuity correction
##
## data:  tabla
## X-squared = 60.124, df = 1, p-value = 8.907e-15
```

Sin embargo la distribución exacta del estadístico del contraste es discreta y como alternativa la podríamos aproximar mediante simulación.

Para simular bajo independencia podemos estimar las probabilidades a partir de las frecuencias marginales de la tabla de contingencia:

```
pind <- (rowSums(tabla) %o% colSums(tabla))/(sum(tabla)^2)
matrix(pind, nrow = nrow(tabla), dimnames = dimnames(tabla))
```

```
##        particulas
## calidad        no         si
##   buena 0.6597531 0.08246914
##   mala  0.2291358 0.02864198
```

Empleando el código anterior podemos generar las simulaciones de las tablas de contingencia (bajo independencia):

```
ntsim <- 2000
rtablas <- rmultinom(ntsim, sum(n), pind)
rtablas[ , 1:5] # Las cinco primeras simulaciones
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  292  285  309  303  290
## [2,]   96  105   97   84  113
## [3,]   48   48   36   49   39
## [4,]   14   12    8   14    8
```

A partir de las cuales podemos aproximar por simulación la distribución exacta del estadístico del contraste chi-cuadrado de independencia:

```
sim.stat <- apply(rtablas, 2, function(x) chisq.test(matrix(x, nrow = nrow(tabla)))$statistic)
hist(sim.stat, freq = FALSE, breaks = 'FD')
# lines(density(sim.stat))
# Distribución asintótica (aproximación chi-cuadrado)
curve(dchisq(x, res$parameter), col = 'blue', add = TRUE)
```

Como se mostrará en la Sección 7.4.3, podríamos aproximar el p-valor del contraste de independencia a partir de esta aproximación:

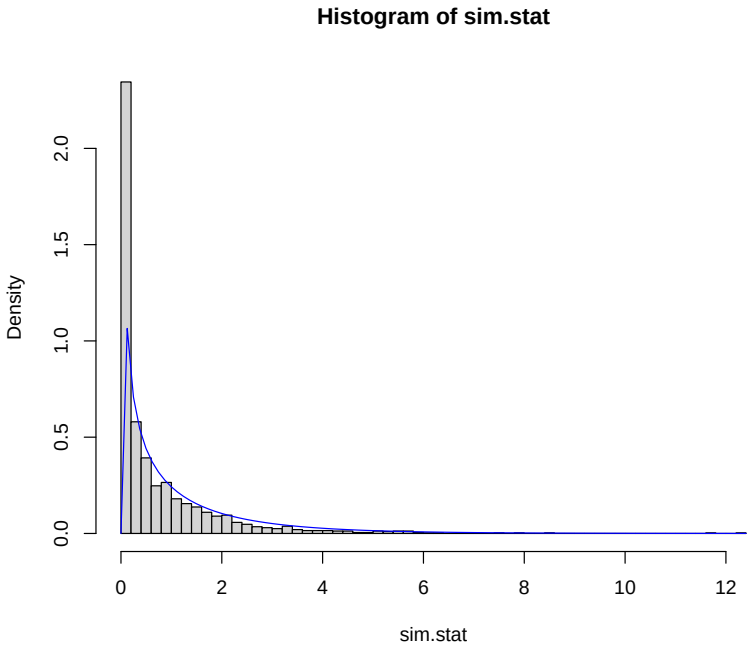


Figura 6.14: Aproximación Monte Carlo de la distribución del estadístico chi-cuadrado bajo independencia.

```
obs.stat <- res$statistic
pvalue.mc <- mean(sim.stat >= obs.stat)
pvalue.mc
```

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

Esto es similar a lo que realiza la función `chisq.test()` con la opción `simulate.p.value = TRUE` (empleando el algoritmo de Patefield, 1981):

```
chisq.test(tabla, simulate.p.value = TRUE, B = 2000)
```

```
##
##  Pearson's Chi-squared test with simulated p-value (based on 2000
##  replicates)
##
## data:  tabla
## X-squared = 62.812, df = NA, p-value = 0.0004998
```

Capítulo 7

Métodos Monte Carlo

Uno de los objetivos habituales en inferencia es la aproximación de una esperanza (o el caso particular de una probabilidad), es decir, se trataría de evaluar una integral, que en ocasiones puede ser compleja. Esto puede ser de interés en otros campos, aunque la integral no esté relacionada con procesos estocásticos. Si el número de dimensiones es pequeño puede ser recomendable emplear métodos numéricos (ver Apéndice B), pero si la dimensión del dominio de la integral es grande, puede ser mucho más eficiente emplear métodos basados en simulación o incluso ser la única aproximación realmente viable. En las secciones 7.1 y 7.2 se describen este tipo de procedimientos, denominados métodos de integración Monte Carlo.

Los métodos Monte Carlo emplean simulación para resolver problemas matemáticos, como la evaluación de expresiones, la aproximación de integrales o la resolución de sistemas, entre muchos otros. Estos problemas son de interés en muchos campos (Física, Economía, Informática...) y pueden ser estrictamente deterministas. Otro ejemplo es la resolución de problemas de optimización. Para evitar problemas de mínimos locales se puede recurrir a herramientas que emplean búsquedas aleatorias de los valores óptimos. En la Sección 7.3 se describen brevemente los principales métodos de optimización Monte Carlo.

En muchos casos, especialmente en Inferencia Estadística, existe una aleatoriedad inherente al modelo empleado para resolver el problema y los métodos Monte Carlo surgen de forma natural. Como se comentó en la Sección 1.1, en ocasiones no se pueden obtener soluciones analíticas a problemas de inferencia, o solo se disponen de resultados asintóticos que pueden no ser suficientemente buenos para muestras finitas, y se puede recurrir a la simulación.

Los estudios Monte Carlo son una herramienta habitual para investigar las ventajas e inconvenientes de un método de inferencia, y para entender mejor su funcionamiento. Por este motivo suelen ser el primer paso en el desarrollo de métodos Monte Carlo (que posteriormente pueden ser objeto de estudios analíticos si producen buenos resultados). También se pueden emplear para tratar de determinar, entre los métodos disponibles, el más adecuado para resolver el problema de interés.

En Estadística Computacional (que incluiría los métodos de Aprendizaje Estadístico/Automático) se emplean métodos de inferencia computacionalmente intensivos y muchos de ellos emplean simulación. Entre las técnicas empleadas destacan los métodos de remuestreo, como el jackknife o el bootstrap, que trataremos en capítulos siguientes. Como ya se comentó, la simulación empleando un modelo estimado también se denomina bootstrap paramétrico y la mayoría de los métodos Monte Carlo de inferencia estadística los podríamos clasificar como métodos de remuestreo.

7.1 Integración Monte Carlo

La integración Monte Carlo se emplea principalmente para aproximar integrales multidimensionales:

$$I = \int \cdots \int_D s\left(x_1, \ldots, x_n\right) dx_1 \cdots dx_n$$

donde puede presentar ventajas respecto a los métodos tradicionales de integración numérica (ver Apéndice B), ya que la velocidad de convergencia no depende del número de dimensiones.

La idea es reescribir la expresión de la integral, encontrando una función de densidad f definida sobre D, de forma que:

$$I = \int_D s(\mathbf{x})d\mathbf{x} = \int h(\mathbf{x})f(\mathbf{x})d\mathbf{x} = E\left(h(\mathbf{X})\right)$$

donde $\mathbf{X} \sim f$ (y preferiblemente fácil de simular).

7.1.1 Integración Monte Carlo clásica

En el caso de que el dominio D sea acotado, la aproximación más simple consiste en considerar una distribución uniforme en D (i.e. $f(\mathbf{x}) = 1_D(\mathbf{x})/|D|$ y $h(\mathbf{x}) = |D|s(\mathbf{x})$).

Por simplicidad nos centraremos en el caso unidimensional (el orden de convergencia es independiente del número de dimensiones). Supongamos que nos interesa aproximar:

$$I = \int_0^1 s(x)dx$$

Si $x_1, x_2, \dots, x_n$ *i.i.d.* $\mathcal{U}(0,1)$ entonces:

$$I = E\left(s\left(\mathcal{U}(0,1)\right)\right) \approx \frac{1}{n}\sum_{i=1}^{n} s\left(x_i\right)$$

Si el intervalo de integración es genérico:

$$I = \int_a^b s(x)dx = (b-a)\int_a^b s(x)\frac{1}{(b-a)}dx = (b-a)E\left(s\left(\mathcal{U}(a,b)\right)\right).$$

Si $x_1, x_2, \dots, x_n$ *i.i.d.* $\mathcal{U}(a,b)$:

$$I \approx \frac{b-a}{n}\sum_{i=1}^{n} s\left(x_i\right)$$

Ejemplo 7.1 (integración Monte Carlo clásica)
Como primera aproximación para implementar la integración Monte Carlo clásica para aproximar integrales del tipo:

$$I = \int_a^b s(x)dx,$$

podríamos considerar la siguiente función:

```
mc.integral0 <- function(fun, a, b, n) {
  # Integración Monte Carlo de `fun()` entre `a` y `b` utilizando una muestra
  # (pseudo) aleatoria de tamaño `n`. Se asume que `fun()` es una función de
  # una sola variable (no vectorial), `a < b` y `n` entero positivo.
  # ------------------------
  x <- runif(n, a, b)
  fx <- sapply(x, fun) # Si fun fuese vectorial bastaría con: fx <- fun(x)
  return(mean(fx) * (b - a))
}
```

Como ejemplo la empleamos para aproximar:

$$\int_0^1 4x^4 dx = \frac{4}{5},$$

```
fun <- function(x) ifelse((x > 0) & (x < 1), 4 * x^4, 0)
# return(4 * x^4)
curve(fun, 0, 1)
abline(h = 0, lty = 2)
abline(v = c(0, 1), lty = 2)
```

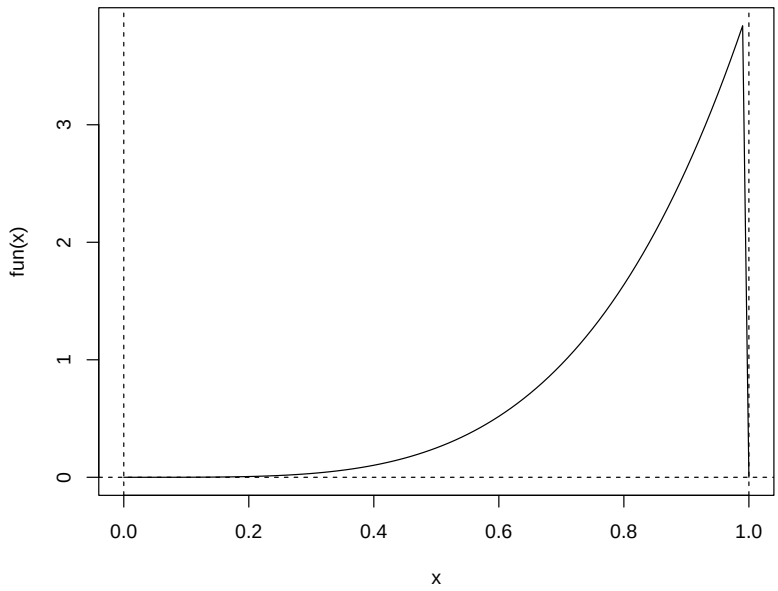


Figura 7.1: Ejemplo de integral en dominio acotado.

```
set.seed(1)
mc.integral0(fun, 0, 1, 20)
```

```
## [1] 0.977663
```

```
mc.integral0(fun, 0, 1, 100)
```

```
## [1] 0.7311169
```

```
mc.integral0(fun, 0, 1, 100)
```

```
## [1] 0.8304299
```

La función `mc.integral0` no es adecuada para analizar la convergencia de la aproximación por simulación. Una alternativa más eficiente para representar gráficamente la convergencia está implementada en la función `mc.integral()` del paquete `simres` (fichero *mc.plot.R*):

```
library(simres)
mc.integral
```

```
## function(fun, a, b, n, level = 0.95, plot = TRUE, ...) {
##   fx <- sapply(runif(n, a, b), fun) * (b - a)
##   result <- if (plot) conv.plot(fx, level = level, ...) else {
##     q <- qnorm((1 + level)/2)
##     list(approx = mean(fx), max.error = q * sd(fx)/sqrt(n))
##   }
##   return(result)
## }
```

```
## <bytecode: 0x000000003b78c0f8>
## <environment: namespace:simres>
```

```
set.seed(1)
mc.integral(fun, 0, 1, 5000, ylim = c(0.2, 1.4))
```

```
## $approx
## [1] 0.8142206
##
## $max.error
## [1] 0.03028194
```

```
abline(h = 4/5, lty = 2, col = "blue")
```

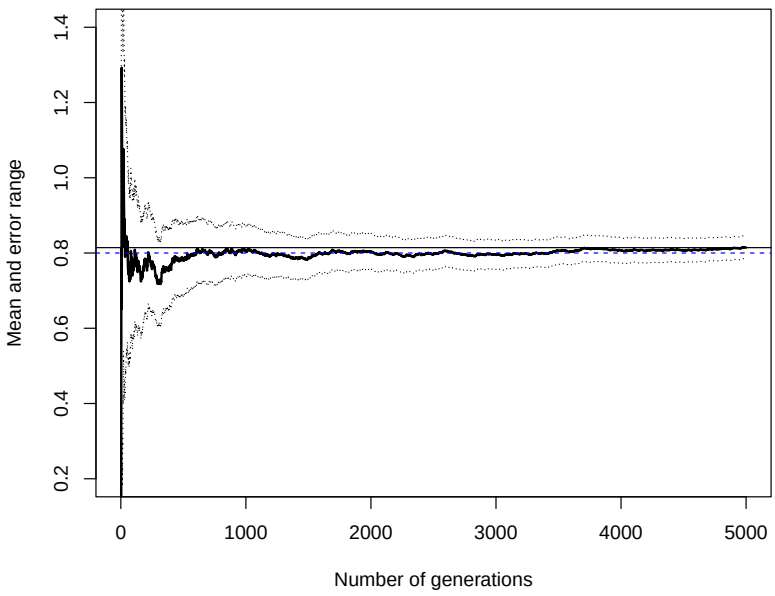


Figura 7.2: Convergencia de la aproximación de la integral mediante simulación.

Si sólo interesa la aproximación:

```
set.seed(1)
mc.integral(fun, 0, 1, 5000, plot = FALSE)
```

```
## $approx
## [1] 0.8142206
##
## $max.error
## [1] 0.03028194
```

Nota: Es importante tener en cuenta que la función `mc.integral()` solo es válida para dominio finito.

7.1.2 Caso general

En lo que resta de esta sección (y en las siguientes) asumiremos que nos interesa aproximar una esperanza:

$$\theta = E\left(h\left(X\right)\right) = \int h\left(x\right) f(x) dx$$

siendo $X \sim f$. Entonces, si $x_1, x_2, \ldots, x_n$ *i.i.d.* X:

$$\theta \approx \frac{1}{n} \sum_{i=1}^{n} h(x_i)$$

Por ejemplo, como en el ejercicio anterior se considera de una función de densidad, se correspondería con el caso general de $h(x) = x$ y $f(x) = 4x^3$ para $0 < x < 1$. La idea es que, en lugar de considerar una distribución uniforme, es preferible generar más valores donde hay mayor "área" (ver Figura 7.1).

Los pasos serían simular `x` con densidad f y aproximar la integral por `mean(h(x))`. En este caso podemos generar valores de la densidad objetivo fácilmente mediante el método de inversión, ya que $F(x) = x^4$ si $0 < x < 1$.

```
rfun <- function(nsim) runif(nsim)^(1/4) # Método de inversión
nsim <- 5000
set.seed(1)
x <- rfun(nsim)
# h <- function(x) x
# res <- mean(h(x)) # Aproximación por simulación
res <- mean(x)
res
```

```
## [1] 0.7967756
```

```
# error <- 2*sd(h(x))/sqrt(nsim)
error <- 2*sd(x)/sqrt(nsim)
error
```

```
## [1] 0.004728174
```

Esta forma de proceder permite aproximar integrales impropias en las que el dominio de integración no es acotado.

Ejemplo 7.2 (integración Monte Carlo con dominio no acotado)
Aproximar:

$$\phi(t) = \int_t^{\infty} \frac{1}{\sqrt{2\pi}} e^{-\frac{x^2}{2}} dx,$$

para $t = 4.5$, empleando integración Monte Carlo (aproximación tradicional con dominio infinito).

```
# h <- function(x) x > 4.5
# f <- function(x) dnorm(x)
set.seed(1)
nsim <- 10^3
x <- rnorm(nsim)
mean(x > 4.5) # mean(h(x))
```

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

```
pnorm(-4.5)  # valor teórico P(X > 4.5)
```

```
## [1] 3.397673e-06
```

De esta forma es difícil que se generen valores (en este caso ninguno) en la región que interesaría para la aproximación de la integral:

```
any(x > 4.5)
```

```
## [1] FALSE
```

Como ya se comentó anteriormente, sería preferible generar más valores donde hay mayor "área", pero en este caso f concentra la densidad en una región que no resulta de utilidad. Por ese motivo puede ser preferible recurrir a una densidad auxiliar que solvente este problema.

7.2 Muestreo por importancia

Para aproximar la integral:

$$\theta = E\left(h\left(X\right)\right) = \int h\left(x\right) f(x) dx,$$

puede ser preferible generar observaciones de una densidad g que tenga una forma similar al producto hf.

Si $Y \sim g$:

$$\theta = \int h\left(x\right) f(x) dx = \int \frac{h\left(x\right) f(x)}{g(x)} g(x) dx = E\left(q\left(Y\right)\right).$$

siendo $q\left(x\right) = \frac{h(x)f(x)}{g(x)}$.

Si $y_1, y_2, \ldots, y_n$ *i.i.d.* $Y \sim g$:

$$\theta \approx \frac{1}{n} \sum_{i=1}^{n} q\left(y_i\right) = \frac{1}{n} \sum_{i=1}^{n} w(y_i) h\left(y_i\right) = \hat{\theta}_g$$

con $w(y) = \frac{f(y)}{g(y)}$.

En este caso $Var(\hat{\theta}_g) = Var\left(q\left(Y\right)\right)/n$, pudiendo reducirse significativamente respecto al método clásico si:

$$g(x) \underset{aprox.}{\propto} |h(x)| f(x),$$

ya que en ese caso $|q(x)|$ sería aproximadamente constante (puede demostrarse fácilmente que la varianza es mínima si esa relación es exacta).

Para garantizar la convergencia de la aproximación por simulación, la varianza del estimador $\hat{\theta}_g$ debería ser finita, i.e.:

$$E\left(q^2\left(Y\right)\right) = \int \frac{h^2\left(x\right) f^2(x)}{g(x)} dx = E\left(h^2\left(X\right) \frac{f(X)}{g(X)}\right) < \infty.$$

La idea básica es que si la densidad g tiene colas más pesadas que la densidad f con mayor facilidad puede dar lugar a "simulaciones" con varianza finita (podría emplearse en casos en los que no existe $E\left(h^2\left(X\right)\right)$; ver Sección 3.1).

La distribución de los pesos $w(y_i)$ debería ser homogénea para evitar datos influyentes (que introducirían inestabilidad en la aproximación).

Ejemplo 7.3

Podríamos aproximar la integral del Ejemplo 7.2 anterior empleando muestreo por importancia considerando como densidad auxiliar una exponencial de parámetro $\lambda = 1$ truncada en t:

$$g(x) = \lambda e^{-\lambda(x-t)},\ x > t,$$

(podemos emplear `dexp(y - t)` para evaluar esta densidad y `rexp(n) + t` para generar valores).

En primer lugar comparamos $h(x)f(x)$ con la densidad auxiliar reescalada, $g(x)f(4.5)$, para comprobar si es una buena elección:

```
curve(dnorm(x), 4.5, 6, ylab = "dnorm(x) y dexp(x-4.5)*k")
abline(v = 4.5)
abline(h = 0)
escala <- dnorm(4.5)  # Reescalado para comparación...
curve(dexp(x - 4.5) * escala, add = TRUE, lty = 2)
```

Generamos valores de la densidad auxiliar y calculamos los pesos:

```
set.seed(1)
nsim <- 10^3
y <- rexp(nsim) + 4.5    #  Y ~ g
w <- dnorm(y)/dexp(y - 4.5)
```

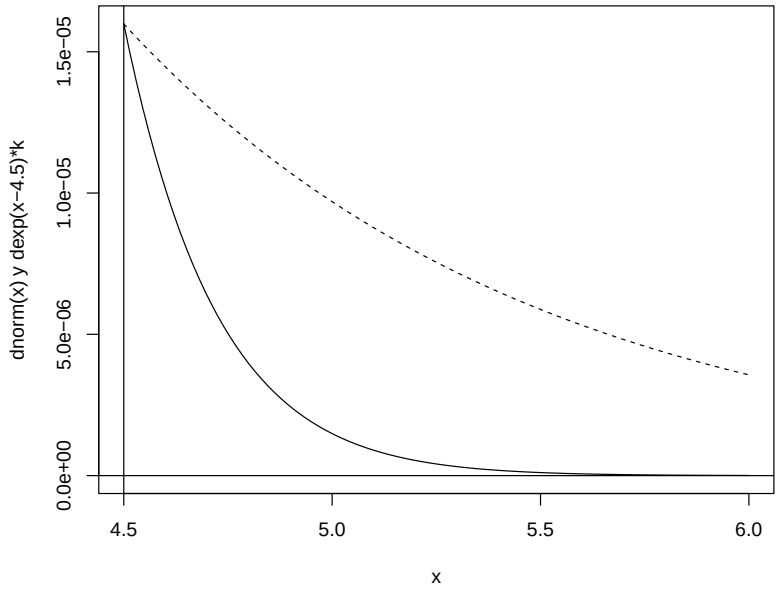


Figura 7.3: Objetivo a integrar (densidad objetivo truncada) y densidad auxiliar reescalada.

La aproximación por simulación sería `mean(w * h(y))`:

```
# h(x) <- function(x) x > 4.5  # (1 si x > 4.5 => h(y) = 1)
mean(w) # mean(w*h(y))
```

```
## [1] 3.144811e-06
```

```
pnorm(-4.5)  # valor teórico
```

```
## [1] 3.397673e-06
```

Representamos gráficamente la aproximación en función del número de simulaciones:

```
plot(cumsum(w)/1:nsim, type = "l", ylab = "Aproximación", xlab = "Iteraciones")
abline(h = pnorm(-4.5), lty = 2)
```

El error estándar de la aproximación sería `sqrt(var(w * h(y))/nsim)`:

```
sqrt(var(w)/nsim) # sd(w*h(y))/sqrt(nsim)
```

```
## [1] 1.371154e-07
```

Mientras que empleando la aproximación tradicional:

```
est <- mean(rnorm(nsim) > 4.5)
est
```

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

```
sqrt(est * (1 - est)/nsim)
```

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

Ejemplo 7.4 (muestreo por importancia con mala densidad auxiliar)

Supongamos que se pretende aproximar $P(2 < X < 6)$ siendo $X \sim Cauchy(0, 1)$ empleando muestreo por importancia y considerando como densidad auxiliar la normal estándar $Y \sim N(0, 1)$. Representaremos gráficamente la aproximación y estudiaremos los pesos $w(y_i)$.

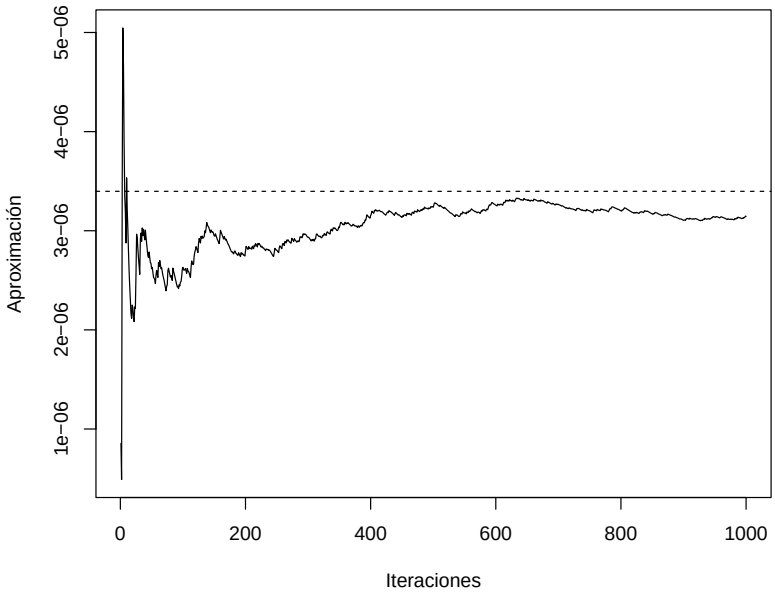


Figura 7.4: Convergencia de la aproximación de la integral mediante muestreo por importancia.

Nota: En este caso van a aparecer problemas (la densidad auxiliar debería tener colas más pesadas que la densidad objetivo; sería adecuado si intercambiáramos las distribuciones objetivo y auxiliar, como en el Ejemplo 7.5 siguiente).

Se trata de aproximar `pcauchy(6) - pcauchy(2)`, i.e. `f(y) = dcauchy(y)` y `h(y) = (y > 2) * (y < 6)`, empleando muestreo por importancia con `g(y) = dnorm(y)`.

```
nsim <- 10^5
set.seed(4321)
y <- rnorm(nsim)
w <- dcauchy(y)/dnorm(y) # w <- w/sum(w) si alguna es una cuasidensidad
```

La aproximación por simulación es `mean(w(y) * h(y))`:

```
mean(w * (y > 2) * (y < 6))
```

```
## [1] 0.09929348
```

```
pcauchy(6) - pcauchy(2)  # Valor teórico
```

```
## [1] 0.09501516
```

Si se estudia la convergencia:

```
plot(cumsum(w * (y > 2) * (y < 6))/1:nsim, type = "l", ylab = "Aproximación", xlab = "Iteraciones")
abline(h = pcauchy(6) - pcauchy(2), lty = 2)
```

Lo que indica es una mala elección de la densidad auxiliar.

La distribución de los pesos debería ser homogénea. Por ejemplo, si los reescalamos para que su suma sea el número de valores generados, deberían tomar valores en torno a uno:

```
boxplot(nsim * w/sum(w))
```

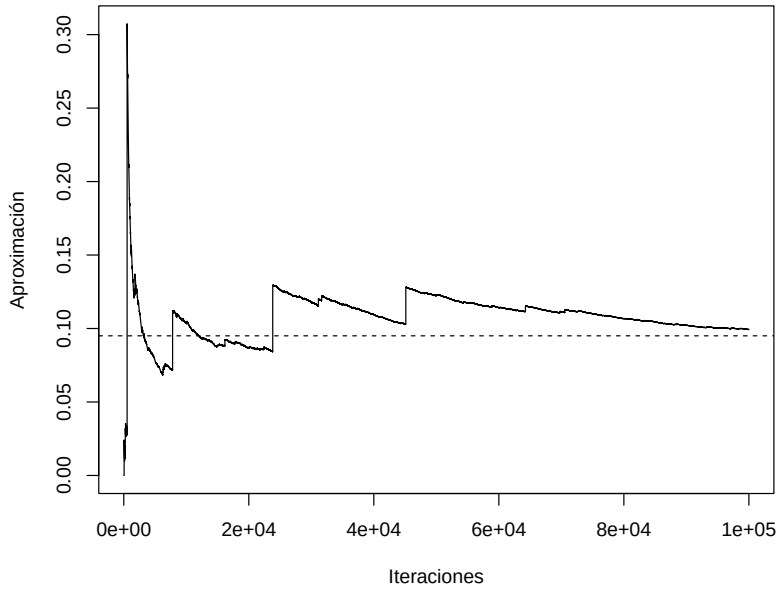


Figura 7.5: Gráfico de convergencia de la aproximación mediante muestreo por importancia con mala densidad auxiliar.

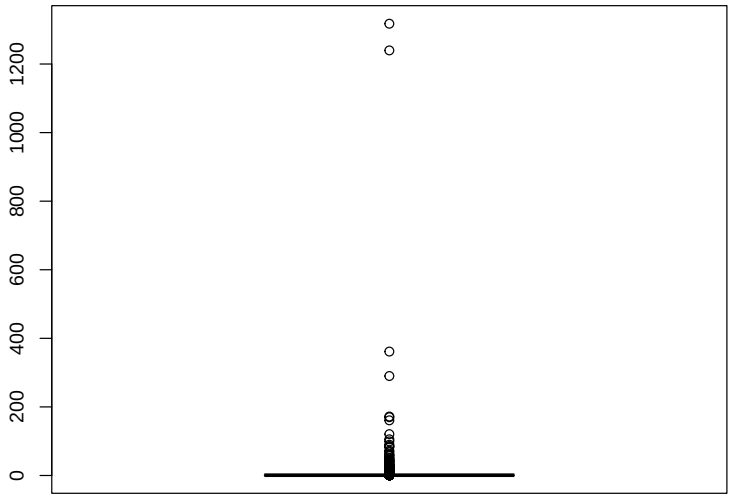


Figura 7.6: Gráfico de cajas de los pesos del muestreo por importancia reescalados (de forma que su media es 1).

7.2.1 Remuestreo (del muestreo) por importancia

Cuando f y/o g son cuasi-densidades, para evitar calcular constantes normalizadoras, se emplea como aproximación:

$$\theta \approx \frac{\sum_{i=1}^{n} w(y_i) h(y_i)}{\sum_{i=1}^{n} w(y_i)}.$$

De hecho este estimador es empleado muchas veces en lugar del anterior ya que, aunque en general no es insesgado, puede ser más eficiente si $w(Y)$ y $w(Y)h(Y)$ están altamente correlacionadas (e.g. Liu, 2004, p.35).

Adicionalmente, puede verse que con un muestreo de $\{y_1, y_2, \dots, y_n\}$ ponderado por $w(y_i)$ (prob. $= w(y_i) \big/ \sum_{i=1}^{n} w(y_i)$) se obtiene una simulación aproximada de f (*Sample importance resampling*, Rubin, 1987).

Ejemplo 7.5 (simulación de normal estándar a partir de Cauchy; Sampling Importance Resampling)
Generamos 1000 simulaciones de una distribución (aprox.) $N(0,1)$ (densidad objetivo) mediante remuestreo del muestreo por importancia de 10^5 valores de una $Cauchy(0,1)$ (densidad auxiliar).

Nota: En este caso `f(y) = dnorm(y)` y `g(y) = dcauchy(y)`, al revés del Ejemplo 7.4 anterior.

```
# Densidad objetivo
# f <- dnorm # f <- function(x) ....

nsim <- 10^3
# El nº de simulaciones de la densidad auxiliar debe ser mucho mayor:
nsim2 <- 10^5
set.seed(4321)
y <- rcauchy(nsim2)
w <- dnorm(y)/dcauchy(y) # w <- w/sum(w) si alguna es una cuasidensidad

# Si se pidiera aproximar una integral
# h(y) = y si es la media # h <- function(y) y
# mean(w * h(y))
```

Sampling Importance Resampling:

```
rx <- sample(y, nsim, replace = TRUE, prob = w/sum(w))
hist(rx, freq = FALSE, breaks = "FD", ylim = c(0, 0.5))
lines(density(rx))
curve(dnorm, col = "blue", add = TRUE)
```

Nota: Si f o g fuesen cuasidensidades y se pidiese aproximar la integral, habría que reescalar los pesos `w <- f(y)/g(y)` en la aproximación por simulación, resultando `sum(w * h(y))/sum(w)` (media ponderada) y en el análisis de convergencia se emplearía `cumsum(w * h(y))/cumsum(w)`.

Ejercicio 7.1 (propuesto)
Consideramos una variable aleatoria con densidad:

$$f(x) \propto e^{-x} \cos^2(x), \text{ si } x > 0.$$

a. Aproximar mediante integración Monte Carlo la media de esta distribución ($h(x) = x$) empleando muestreo de importancia con distribución auxiliar una exponencial de parámetro $\lambda = 1$ a partir de 10000 simulaciones (OJO: se conoce la cuasi-densidad de la variable aleatoria de interés, emplear la aproximación descrita en apuntes).

b. Generar 500 simulaciones (aprox.) de la distribución de interés mediante remuestreo del muestreo por importancia.

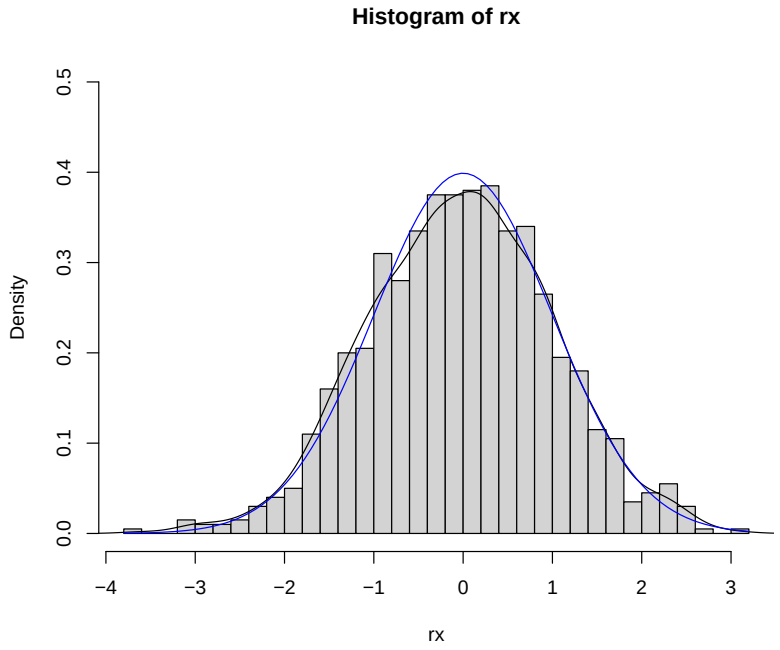


Figura 7.7: Distribución de los valores generados mediante remuestreo por importancia y densidad objetivo.

NOTA: En el último apartado, para comprobar que los valores generados proceden de la distribución objetivo, si representamos la cuasidensidad $f^*(x) = e^{-x}\cos^2(x)$ junto con el histograma (en escala de densidades, `freq = FALSE`), hay que tener en cuenta que faltaría dividir la cuasidensidad por una constante normalizadora para poder compararlos directamente. Si no se reescala la cuasidensidad, podríamos compobar si la forma es similar (si la distribución de los valores generados es proporcional a la cuasidensidad, con mayor concentración donde la cuasidensidad se aleja de 0). En este caso (como g es una densidad) podríamos estimar la constante normalizadora ($f(x) = \frac{1}{c}f^*(x)$) a partir de los pesos del muestreo por importancia (`c.approx <- mean(w)`; en este caso concreto $c = \frac{3}{5}$).

7.3 Optimización Monte Carlo

Supongamos que estamos interesados en la minimización de una función:

$$\min_{\mathbf{x}\in D} f(\mathbf{x}).$$

Hay una gran cantidad de algoritmos numéricos para resolver problemas de optimización no lineal multidimensional, por ejemplo los basados en el método de Newton-Raphson (implementados en la función `nlm`, entre otras).

La idea original consiste en buscar los ceros de su primera derivada (o del gradiente) empleando una aproximación iterativa:

$$\mathbf{x}_{i+1} = \mathbf{x}_i - [Hf(\mathbf{x}_i)]^{-1}\nabla f(\mathbf{x}_i),$$

donde $Hf(\mathbf{x}_i)$ es el hessiano de la función (matriz de segundas derivadas) y $\nabla f(\mathbf{x}_i)$ el gradiente (vector de primeras derivadas). Estos métodos normalmente funcionan muy bien cuando la función objetivo no tiene mínimos locales (ideal f cuadrática). Los resultados obtenidos pueden ser muy malos en caso contrario (especialmente en el caso multidimensional) y dependen en gran medida del punto inicial[1] Un ejemplo donde es habitual que aparezcan este tipo de problemas es en la estimación por máxima verosimilitud (la función objetivo puede ser multimodal).

[1] Este tipo de algoritmos se denominan *codiciosos* o *voraces*, porque buscan la mejor opción a "corto plazo".

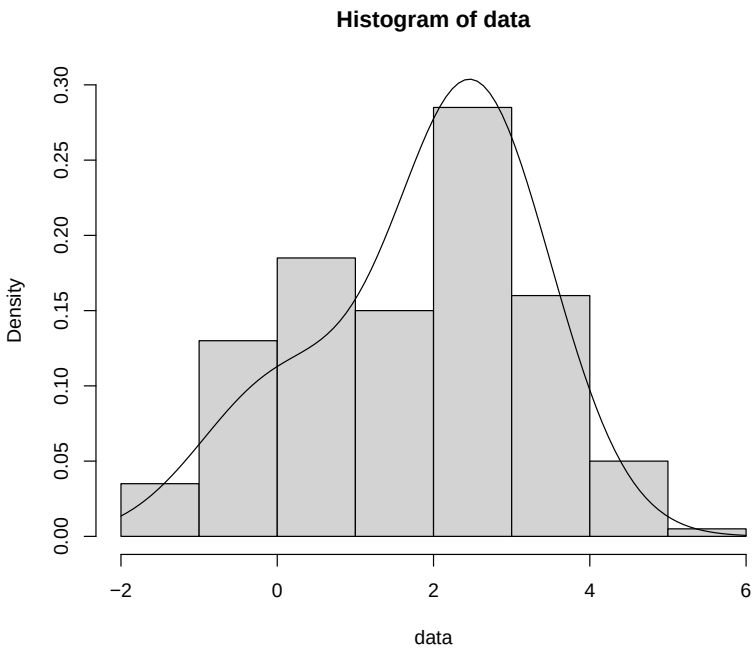
Ejemplo 7.6 (Estimación por máxima verosimilitud mediante un algoritmo de Newton) La mixtura de distribuciones normales:

$$\frac{1}{4}N(\mu_1, 1) + \frac{3}{4}N(\mu_2, 1),$$

tiene una función de verosimilitud asociada bimodal. Generaremos una muestra de 200 valores de esta distribución con $\mu_1 = 0$ y $\mu_2 = 2.5$, construiremos la correspondiente función de verosimilitud y la representaremos gráficamente.

Obtención de la muestra (simulación mixtura dos normales):

```
nsim <- 200
mu1 <- 0
mu2 <- 2.5
sd1 <- sd2 <- 1

set.seed(12345)
p.sim <- rbinom(nsim, 1, 0.25)
data <- rnorm(nsim, mu1*p.sim + mu2*(1-p.sim), sd1*p.sim + sd2*(1-p.sim))

hist(data, freq = FALSE, breaks = "FD", ylim = c(0, 0.3))
curve(0.25 * dnorm(x, mu1, sd1) + 0.75 * dnorm(x, mu2, sd2), add = TRUE)
```



Podemos obtener la estimación por máxima verosimilitud de los parámetros empleando la rutina `nlm` para minimizar el logaritmo (negativo) de la función de verosimilitud:

```
like <- function(mu)
  -sum(log((0.25 * dnorm(data, mu[1], sd1) + 0.75 * dnorm(data, mu[2], sd2))))
  # NOTA: Pueden aparecer NA/Inf por log(0)
```

Si queremos capturar los valores en los que se evalúa esta función, podemos proceder de forma similar a como se describe en el capítulo Function operators de la primera edición del libro "Advanced R" de Hadley Wickham: "Behavioural FOs leave the inputs and outputs of a function unchanged, but add some extra behaviour".

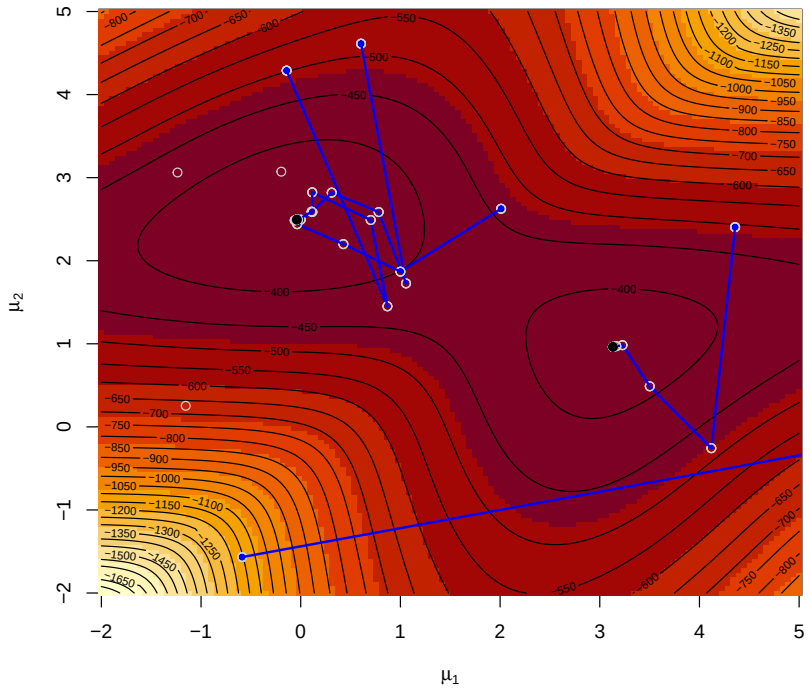
```
tee <- function(f) {
  function(...) {
    input <- if(nargs() == 1) c(...) else list(...)
    output <- f(...)
    # Hacer algo ...
    # ... con output e input
    return(output)
  }
}
```

En este caso queremos representar los puntos en los que el algoritmo de optimización evalúa la función objetivo (especialmente como evoluciona el valor óptimo)

```
tee.optim2d <- function(f) {
  best.f <- Inf   # Suponemos que se va a minimizar (opción por defecto)
  best.par <- c(NA, NA)
  function(...) {
    input <- c(...)
    output <- f(...)
    ## Hacer algo ...
    points(input[1], input[2], col = "lightgray")
    if(best.f > output) {
      lines(rbind(best.par, input), lwd = 2, col = "blue")
      best.f <<- output
      best.par <<- input
      # points(best.par[1], best.par[2], col = "blue", pch = 20)
      # cat("par = ", best.par, "value = ", best.f, "\n")
    }
    ## ... con output e input
    return(output)
  }
}
```

Representar la superficie del logaritmo de la verosimilitud, los puntos iniciales y las iteraciones en la optimización numérica con `nlm`:

```
mmu1 <- mmu2 <- seq(-2, 5, length = 128)
lli <- outer(mmu1, mmu2, function(x,y) apply(cbind(x,y), 1, like))

par(mar = c(4, 4, 1, 1))
image(mmu1, mmu2, -lli, xlab = expression(mu[1]), ylab = expression(mu[2]))
contour(mmu1, mmu2, -lli, nlevels = 50, add = TRUE)

# Valores iniciales aleatorios
nstarts <- 5
set.seed(1)
starts <- matrix(runif(2*nstarts, -2, 5), nrow = nstarts)
points(starts, col = "blue", pch = 19)

# Minimización numérica con nlm
for (j in 1:nstarts){
  # Normalmente llamaríamos a nlm(like, start)
  res <- nlm(tee.optim2d(like), starts[j, ]) # nlm(like, starts[j, ])
  points(res$estimate[1],res$estimate[2], pch = 19)
  cat("par = ", res$estimate, ", value =", res$minimum, "\n")
}
```




```
## par =  -0.03892511 2.494589 , value = 361.5712
## par =  -0.03892501 2.494589 , value = 361.5712
## par =  -0.03892507 2.494589 , value = 361.5712
## par =  3.132201 0.9628536 , value = 379.3739
## par =  20.51013 1.71201 , value = 474.1414
```

7.3.1 Algoritmos de optimización Monte Carlo

Una alternativa sería tratar de generar valores aleatoriamente de forma que las regiones donde la función objetivo es menor tuviesen mayor probabilidad y menor probabilidad las regiones donde la función objetivo es mayor. Por ejemplo, se podría pensar en generar valores de acuerdo a una densidad (tranformación Boltzman-Gibbs):

$$g(x) \propto \exp\left(-f(x)/T\right),$$

donde $T > 0$ es un parámetro (denominado temperatura) seleccionado de forma que se garantice la integrabilidad.

Entre los métodos de optimización Monte Carlo podríamos destacar:

- Métodos con gradiente aleatorio.
- Temple simulado.
- Algoritmos genéticos.
- Monte Carlo EM.
- ...

7.3.2 Temple simulado

Método inspirado en el templado de un metal (se calienta el metal a alta temperatura y se va enfriando lentamente). En cada paso se reemplaza la aproximación actual por un valor aleatorio "cercano", elegido con una probabilidad que depende de la mejora en la función objetivo y de un parámetro T (denominado temperatura) que disminuye gradualmente durante el proceso.

- Cuando la temperatura es grande los cambios son bastante probables en cualquier dirección.
- Al ir disminuyendo la temperatura los cambios tienden a ser siempre "cuesta abajo".

Al tener una probabilidad no nula de aceptar una modificación "cuesta arriba" se trata de evitar quedar atrapado en un óptimo local (ver Figura 7.8).

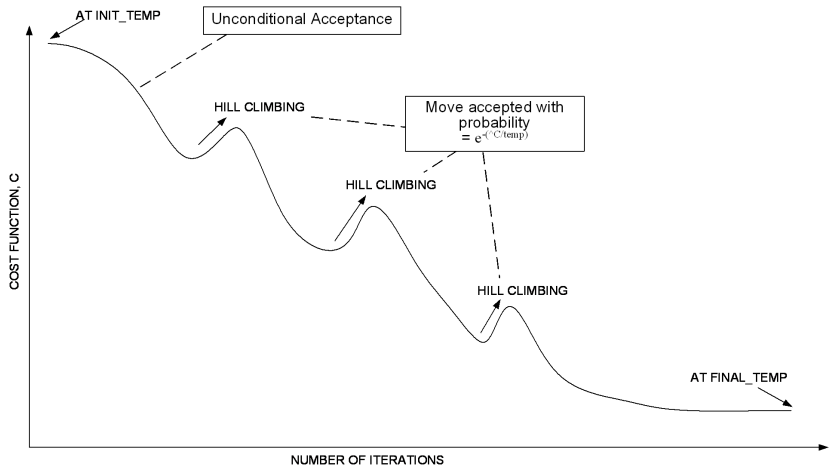


Figura 7.8: Fuente: Premchand Akella ([ppt](https://www.presentica.com/doc/11473134/simulated-annealing-pdf-document)).

Este procedimiento se puede ver como una adaptación del método de Metropolis-Hastings que se tratará en el Capítulo XX (Introducción a los métodos de cadenas de Markov Monte Carlo).

Algoritmo

```
temp <- temp.ini
par <- par.ini
fun.par <- FUN(par)
iter <- 1
while(temp > temp.final && iter < iter.max) {
  iter.temp <- 1
  while(iter.temp < iter.temp.max)) { # iteraciones con temperatura constante
    par.new <- PERTURB(par, temp)
    fun.new <- FUN(par.new)
    fun.inc <- fun.new - fun.par
    if ((fun.inc < 0) || (runif(1) > exp(-(fun.inc/temp)))) break
    iter.temp <- iter.temp + 1
  }
  iter <- iter + iter.temp
  par <- par.new
  fun.par <- fun.new
  temp <- SCHEDULE(temp)
}

FUN <- function(par, ...) {...}
SCHEDULE <- function(temp, temp.ini, iter)
  temp.ini / log(iter + exp(1) - 1)
  # temp.ini / log(((temp - 1) %/% tmax)*tmax + exp(1))
PERTURB <- function(par, temp, scale = 1/temp.ini)
  rnorm(length(par), par, 1/(scale*temp))
```

Una versión de este método está implementado[2] en la función `optim()`:

```
optim(par, fn, gr = NULL, ..., method = "SANN", control = list(maxit = 10000, temp = 10, tmax = 10
```

El argumento `gr` permite especificar la función para generar posiciones candidatas (por defecto nú-

[2]En el fichero fuente optim.c.

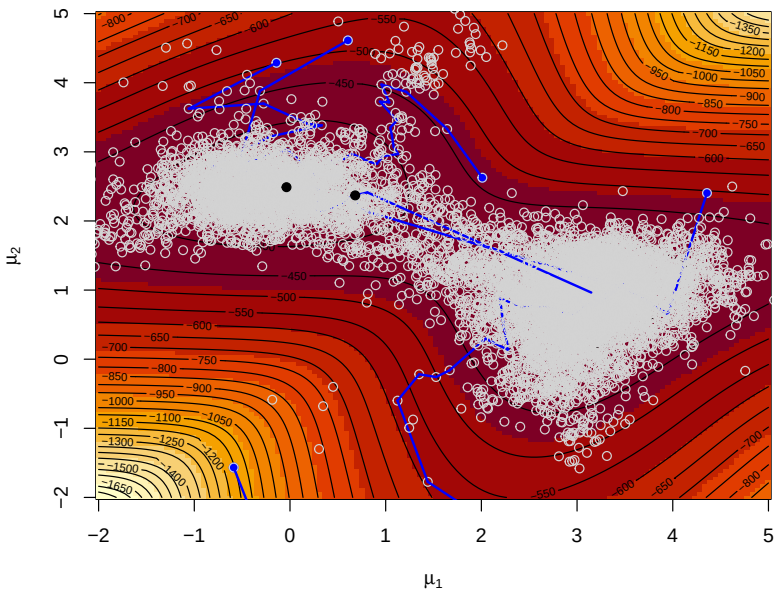
cleo gausiano con escala proporcional a la temperatura actual) y permitiría resolver problemas de optimización combinatoria. El argumento `control` permite establecer algunas opciones adicionales:

- `maxit`: número total de evaluaciones de la función (único criterio de parada), por defecto 10000.
- `temp`: temperatura inicial, por defecto 10.
- `tmax`: número de evaluaciones de la función para cada temperatura, por defecto 10.

Ejemplo 7.7 (Estimación máximo-verosimil empleando temple simulado)

Repetimos el Ejemplo 7.6 anterior empleando el método "SANN" de la función `optim()`:

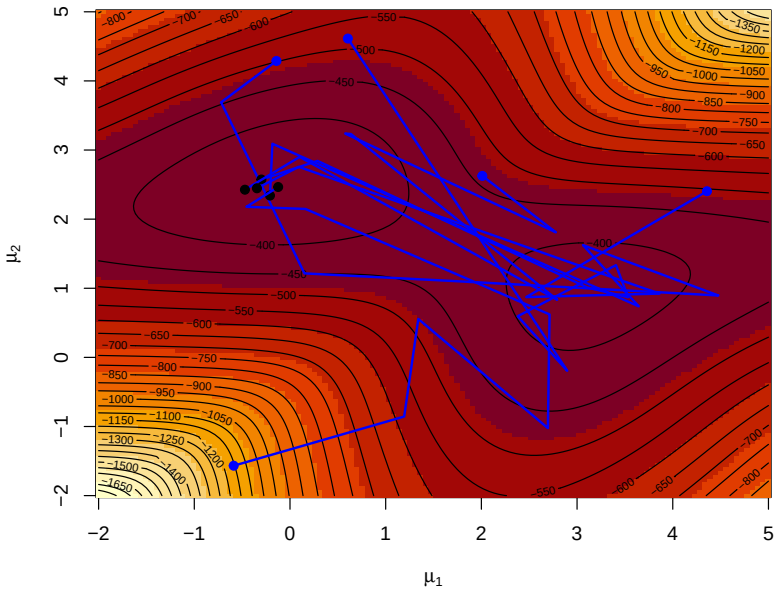
```
# Representar la superficie del logaritmo de la verosimilitud
image(mmu1, mmu2, -lli, xlab = expression(mu[1]), ylab = expression(mu[2]))
contour(mmu1, mmu2, -lli, nlevels = 50, add = TRUE)
points(starts, col = "blue", pch = 19)

set.seed(1)
for (j in 1:nstarts){
  # Normalmente llamaríamos a optim(start, like, method = "SANN")
  # Note that the "SANN" method depends critically on the settings of the control parameters.
  # For "SANN" maxit gives the total number of function evaluations: there is no other stopping crit
  # Defaults to 10000.
  res <- optim(starts[j, ], tee.optim2d(like), method = "SANN", control = list(temp = 100, maxit = 20
  points(res$par[1],res$par[2], pch = 19)
  cat("par = ", res$par, ", value =", res$value, "\n")
}
```




```
## par =  0.0002023461 2.473437 , value = 361.6372
## par =  -0.182735 2.45585 , value = 362.0255
## par =  -0.0281341 2.484467 , value = 361.5801
## par =  -0.03642928 2.488626 , value = 361.5732
## par =  0.6814165 2.370026 , value = 374.839
```

Como alternativa podríamos emplear la siguiente función basada en el algoritmo del Ejemplo 5.9 de Robert y Casella (2010):

```
SA <- function(fun, pini, lower = -Inf, upper = Inf, tolerance = 1e-04, factor = 1) {
  temp <- scale <- iter <- dif <- 1
  npar <- length(pini)
  par <- matrix(pini, ncol = npar)
  curfun <- hval <- fun(pini)
  while (dif > tolerance) {
    prop <- par[iter, ] + rnorm(npar) * scale[iter]
    # Se decide si se acepta la propuesta
    if (any(prop < lower) || any(prop > upper) ||
        (temp[iter] * log(runif(1)) > curfun - fun(prop)))  prop <- par[iter, ]
    curfun <- fun(prop)
    hval <- c(hval, curfun)
    par <- rbind(par, prop)
    iter <- iter + 1
    temp <- c(temp, 1/log(iter + 1))  # Actualizar la temperatura
    # Se controla el número de perturbaciones aceptadas
    ace <- length(unique(par[(iter/2):iter, 1]))
    if (ace == 1)
      # si es muy pequeño se disminuye la escala de la perturbación
      factor <- factor/10
    if (2 * ace > iter)
      # si es muy grande se aumenta
      factor <- factor * 10
    scale <- c(scale, max(2, factor * sqrt(temp[iter])))  # Actualizar la escala de la perturbaci
    dif <- (iter < 100) + (ace < 2) + (max(hval) - max(hval[1:(iter/2)]))
  }
  list(par = par, value = hval, iter = iter)
}

# Representar la superficie del logaritmo de la verosimilitud
image(mmu1, mmu2, -lli, xlab = expression(mu[1]), ylab = expression(mu[2]))
contour(mmu1, mmu2, -lli, nlevels = 50, add = TRUE)
points(starts, col = "blue", pch = 19)

set.seed(1)
for (j in 1:nstarts) {
  sar <- SA(like, starts[j, ])
  with(sar, lines(par[, 1], par[, 2], lwd = 2, col = "blue"))
  with(sar, points(par[iter, 1], par[iter, 2], pch = 19))
  with(sar, cat("par = ", par[iter, ], ", value =", value[iter], "\n"))
}
```




```
## par =  -0.2091332 2.341469 , value = 363.0035
## par =  -0.2986682 2.573345 , value = 363.6607
## par =  -0.4708455 2.425984 , value = 365.3277
## par =  -0.3454382 2.446332 , value = 363.5074
## par =  -0.1236326 2.464842 , value = 361.7403
```

7.3.3 Algoritmos genéticos

Los algoritmos genéticos tratan de encontrar la mejor solución (entre un conjunto de soluciones posibles) imitando los procesos de evolución biológica:

- **Población**: formada por n individuos $\mathbf{x}_i$ codificados en **cromosomas**.
- $f(\mathbf{x}_i)$ ajuste/capacidad/**adaptación** del individuo $\mathbf{x}_i$.
- **Selección**: los individuos mejor adaptados tienen mayor probabilidad de ser **padres**.
- **Cruzamiento**: los cromosomas de dos padres se combinan para generar hijos.
- **Mutación**: modificación al azar del cromosoma de los hijos (variabilidad).
- **Elitismo**: el mejor individuo pasa a la siguiente generación.

Los paquetes de R `DEOptim` y `gafit` implementan algunos de estos tipos de algoritmos.

Ejemplo 7.8 (Estimación máximo-verosimil empleando un algoritmo genético)

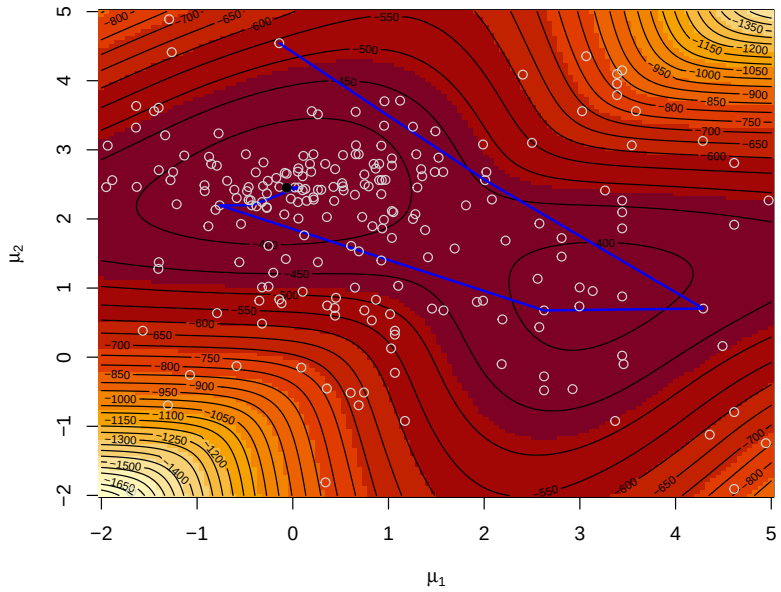
Repetimos el ejemplo anterior empleando el algoritmo genético implementado en la función `DEoptim::DEOptim()`:

```
require(DEoptim)

# Representar la superficie del logaritmo de la verosimilitud
image(mmu1, mmu2, -lli, xlab = expression(mu[1]), ylab = expression(mu[2]))
contour(mmu1, mmu2, -lli, nlevels = 50, add = TRUE)
# Estos algoritmos no requieren valores iniciales (los generan al azar en el rango)

lower <- c(-2, -2)
upper <- c(5, 5)
```

```
set.seed(1)
# DEoptim(like, lower, upper)
der <- DEoptim(tee.optim2d(like), lower, upper, DEoptim.control(itermax = 10))
```

```
## Iteration: 1 bestvalit: 373.132461 bestmemit:   -0.764103    2.196961
## Iteration: 2 bestvalit: 367.580379 bestmemit:   -0.430095    2.196961
## Iteration: 3 bestvalit: 367.580379 bestmemit:   -0.430095    2.196961
## Iteration: 4 bestvalit: 367.580379 bestmemit:   -0.430095    2.196961
## Iteration: 5 bestvalit: 361.906887 bestmemit:    0.058951    2.455186
## Iteration: 6 bestvalit: 361.906887 bestmemit:    0.058951    2.455186
## Iteration: 7 bestvalit: 361.906887 bestmemit:    0.058951    2.455186
## Iteration: 8 bestvalit: 361.657986 bestmemit:   -0.064005    2.452184
## Iteration: 9 bestvalit: 361.657986 bestmemit:   -0.064005    2.452184
## Iteration: 10 bestvalit: 361.657986 bestmemit:   -0.064005    2.452184
```

```
# Por defecto fija el tamaño de la población a NP = 10*npar = 20
# Puede ser mejor dejar el valor por defecto itermax = 200
points(der$optim$bestmem[1], der$optim$bestmem[2], pch = 19)
```



7.4 Métodos Monte Carlo en Inferencia Estadística

*** Work in progress ***: Esta sección es muy preliminar y variará en siguientes versiones

Como ya se comentó en la introducción muchas de las aplicaciones de la simulación serían de utilidad en Estadística:

- Distribución de estimadores puntuales/estadísticos:
  - Aproximación de la distribución.
    - Aproximación de características de la distribución.
    - Valided de la distribución asintótica.
  - Comparación de estimadores.
- Estimación por intervalo de confianza:

  - Obtención de intervalos/bandas de confianza (probabilidad).
  - Análisis de un estimador por intervalo de confianza.
- Contrastes de hipótesis:
  - Aproximación del p-valor.
  - Análisis de un contraste de hipótesis.
  - Validación teoría.
- Métodos de remuestro bootstrap.
- Inferencia Bayesiana
- ...

En esta sección nos centraremos en estudios de simulación Monte Carlo en algunas de estas aplicaciones y daremos algún ejemplo de métodos Monte Carlo para inferencia estadística. La mayoría de los métodos Monte Carlo los podríamos clasificar como métodos de remuestreo y se tratarán con mayor profundidad en capítulos siguientes.

Observación: En esta sección se obtendrán simulaciones de estadísticos a partir de muestras (podemos pensar que se parte de generaciones de una variable multivariante). En la mayoría de los ejemplos se generan todas las muestras de una vez, se guardan y se procesan vectorialmente (normalmente empleando la función `apply`). Como ya se comentó en el Capítulo 1.3, en problemas mas complejos, en los que no es necesario almacenar todas las muestras, puede ser preferible emplear un bucle para generar y procesar las muestras iterativamente.

7.4.1 Distribución en el muestreo

Ejercicio 7.2 (Distribución de la media muestral)

Si $X_1, \dots, X_n$ es una muestra aleatoria simple de una variable aleatoria $X \sim N(\mu, \sigma)$, la distribución en el muestreo de:

$$\hat{\mu} = \overline{X} = \frac{1}{n}\sum_{i=1}^{n} X_i$$

es:

$$\overline{X} \sim N\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$$

Confirmar este resultado mediante simulación, para ello:

a) Crear un conjunto de datos `muestras` con 500 muestras de tamaño $n = 10$ de una $N(1, 2)$. Añadir al conjunto de datos las estimaciones de la media y desviación típica obtenidas con cada una de las muestras.

   Valores iniciales:

```
set.seed(54321) # Fijar semilla para reproducibilidad
nsim <- 500
nx <- 10
```

   Valores teóricos:

```
mux <- 1
sdx <- 2
```

   Simulación de las muestras (al estilo `Rcmdr`):

```
muestras <- as.data.frame(matrix(rnorm(nsim*nx, mean=mux, sd=sdx), ncol=nx))
rownames(muestras) <- paste("muestra", 1:nsim, sep="")
colnames(muestras) <- paste("obs", 1:nx, sep="")
str(muestras)
```

```
## 'data.frame':    500 obs. of  10 variables:
##  $ obs1 : num  0.642 -0.856 -0.568 -2.301 0.184 ...
##  $ obs2 : num  3.483 2.216 1.1 4.305 0.677 ...
##  $ obs3 : num  1.24 -1.51 -3.98 2.29 2.46 ...
##  $ obs4 : num  3.286 0.947 0.953 -1.663 2.623 ...
##  $ obs5 : num  3.77 -1.34 1.61 -2.46 1.11 ...
##  $ obs6 : num  -2.044 0.32 3.046 0.136 3.555 ...
##  $ obs7 : num  0.6186 -1.8614 4.3386 0.0996 0.8334 ...
##  $ obs8 : num  -0.829 2.202 -1.688 1.534 -0.114 ...
##  $ obs9 : num  0.4904 -0.6713 0.5451 -0.6517 0.0168 ...
##  $ obs10: num  2.79 2.84 1.27 3.93 2.17 ...
```

Estimaciones:

```
muestras$mean <- rowMeans(muestras[,1:nx])
muestras$sd <- apply(muestras[,1:nx], 1, sd)
```

La fila `muestras[i,]` contiene las observaciones de la i-ésima muestra y la correspondiente media y desviación típica.

```
muestras[1,]
```

```
##               obs1     obs2     obs3    obs4     obs5     obs6      obs7
## muestra1 0.6421985 3.482661 1.242483 3.28559 3.766896 -2.04443 0.6186323
##                obs8      obs9    obs10     mean       sd
## muestra1 -0.8293636 0.4903819 2.790091 1.344514 1.951292
```

Normalmente emplearemos sin embargo una ordenación por columnas (cada fila se corresponderá con una generación).

b) Generar el histograma (en escala de densidades) de las medias muestrales y compararlo con la densidad teórica.

Distribución de la media muestral:

```
hist(muestras$mean, freq = FALSE, breaks = "FD",
     xlab = "Medias", ylab = "Densidad")
# Densidad observada (estimación)
lines(density(muestras$mean))
# Densidad teórica (bajo normalidad)
curve(dnorm(x, mux, sdx/sqrt(nx)), lwd = 2, col = "blue", add = TRUE)
# Aproximación del valor esperado de la media muestral mediante simulación
abline(v = mean(muestras$mean), lty = 2)
# Valor esperado de la media muestral (teórico)
abline(v = mux, col = "blue")
```

Ejercicio 7.3 (Distribución de la media muestral continuación)

Si $X_1, \dots, X_n$ es una m.a.s. de una variable aleatoria X (cualquiera) con $E(X) = \mu$ y $Var(X) = \sigma^2$, por el Teorema Central del Límite, la distribución en el muestreo de $\hat{\mu} = \overline{X}$ se aproxima a la normalidad:

$$\overline{X} \underset{n\to\infty}{\longrightarrow} N\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$$

Típicamente se suele considerar que esta aproximación es buena para tamaños muestrales $n > 30$, aunque dependerá de las características de la distribución de X.

a) Repetir el Ejercicio 7.2 anterior considerando muestras de una $Exp(1)$ (tener en cuenta que $X \sim Exp(\lambda) \Rightarrow \mu_X = \sigma_X = 1/\lambda$). ¿Qué ocurre con la distribución de la media muestral?

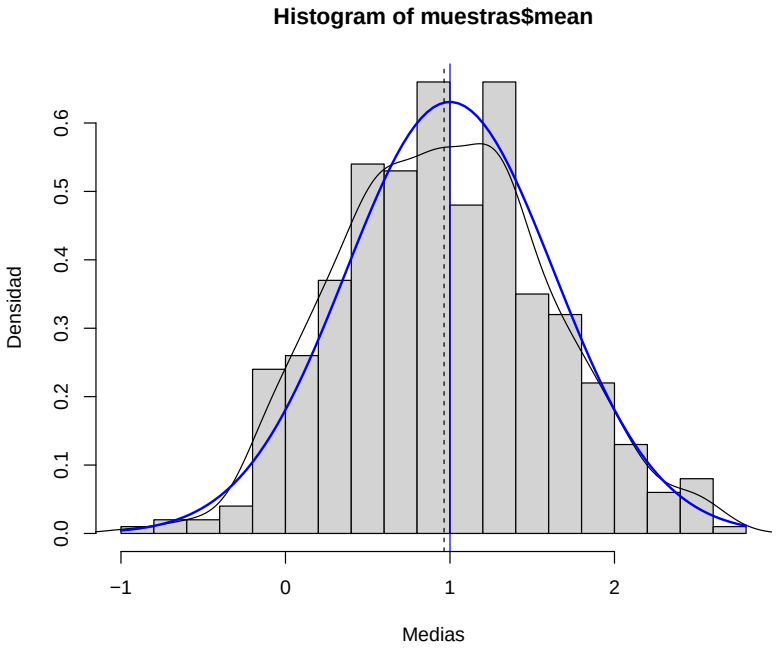


Figura 7.9: Distribución de la media muestral de una distribución normal.

```
set.seed(54321) # Fijar semilla para reproducibilidad
nsim <- 500
nx <- 10
# nx <- 50
```

Valores teóricos:

```
lambda <- 1
muexp <- 1/lambda
sdexp <- muexp
```

Simulación de las muestras:

```
muestras2 <- as.data.frame(matrix(rexp(nsim*nx, rate=lambda), ncol=nx))
rownames(muestras2) <- paste("muestra", 1:nsim, sep="")
colnames(muestras2) <- paste("obs", 1:nx, sep="")
```

Estimaciones:

```
muestras2$mean <- rowMeans(muestras2[,1:nx])
muestras2$sd <- apply(muestras2[,1:nx], 1, sd)
```

Distribución de la media muestral:

```
hist(muestras2$mean, xlim = c(-0.1, 2.5), freq = FALSE, breaks = "FD",
     xlab = "Medias", ylab = "Densidad")
# Densidad observada (estimación)
lines(density(muestras2$mean))
# Distribución asintótica (TCL)
curve(dnorm(x,muexp,sdexp/sqrt(nx)), lwd=2, col="blue", add=TRUE)
# Aproximación del valor esperado de la media muestral mediante simulación
abline(v=mean(muestras2$mean),lty=2)
# Valor esperado de la media muestral (teórico)
```

```
abline(v=muexp, col="blue")
```

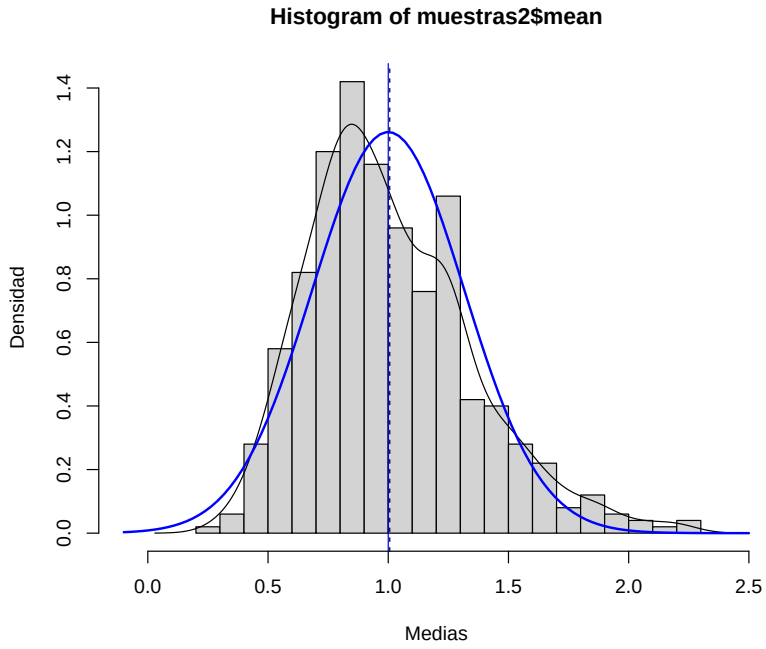


Figura 7.10: Distribución de la media muestral de una distribución exponencial y distribución asintótica.

b) Aumentar el tamaño muestral a 50. ¿Se aproxima más la distribución de las medias muestrales a la teórica bajo normalidad?

   Ejecutar el código del apartado anterior fijando `nx <- 50`.

7.4.2 Intervalos de confianza

Ejercicio 7.4 (Intervalo de confianza para la media)

A partir del enunciado del Ejercicio 7.2, se deduce que el intervalo de confianza (de nivel $1-\alpha$) para la media μ de una población normal con varianza conocida es:

$$IC_{1-\alpha}(\mu) = \left(\overline{X} - z_{1-\alpha/2}\frac{\sigma}{\sqrt{n}},\ \overline{X} + z_{1-\alpha/2}\frac{\sigma}{\sqrt{n}}\right).$$

La idea es que el $100(1-\alpha)\%$ de los intervalos así construidos contentrán el verdadero valor del parámetro.

a) Utilizando el conjunto de datos `muestras` del ejercicio 1 (500 muestras de tamaño $n = 10$ de una $N(1, 2)$), añadir en dos nuevas variables los extremos del intervalo de confianza para la media con varianza conocida al conjunto de datos. Analizar la cobertura de estas estimaciones por IC.

   IC para la media con varianza conocida (bajo normalidad):

```
alfa <- 0.05
z <- qnorm(1 - alfa/2)
muestras$ici <- muestras$mean - z*sdx/sqrt(nx)
muestras$ics <- muestras$mean + z*sdx/sqrt(nx)
```

   Cobertura de las estimaciones por IC:

```
muestras$cob <- (muestras$ici < mux) & (mux < muestras$ics)
ncob <- sum(muestras$cob) # Nº de intervalos que contienen la verdadera media
ncob
```

```
## [1] 480
```

```
100*ncob/nsim     # Proporción de intervalos
```

```
## [1] 96
```

```
100*(1 - alfa)    # Proporción teórica bajo normalidad
```

```
## [1] 95
```

Como ejemplo ilustrativo, generamos el gráfico de los primeros 50 intervalos:

```
m <- 50
tmp <- muestras[1:m,]
attach(tmp)
color <- ifelse(cob,"blue","red")
plot(1:m, mean, col = color, ylim = c(min(ici),max(ics)),
     xlab = "Muestra", ylab = "IC")
arrows(1:m, ici, 1:m, ics, angle = 90, length = 0.05, code = 3, col = color)
abline(h = mux, lty = 3)
```

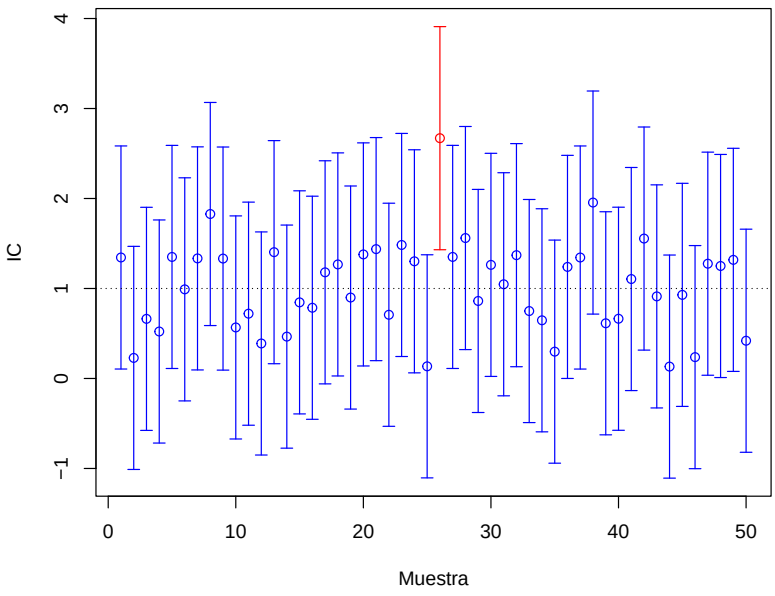


Figura 7.11: Cobertura de las estimaciones por IC.

```
detach(tmp)
```

b) Repetir el apartado anterior considerando muestras de una $Exp(1)$. ¿Qué ocurre con la cobertura del intervalo de confianza obtenido bajo normalidad?

Ejecutar el código del apartado a) del ejercicio 2.

IC para la media con varianza conocida (bajo normalidad)

```
alfa <- 0.05
z <- qnorm(1 - alfa/2)
muestras2$ici <- muestras2$mean - z*sdexp/sqrt(nx)
muestras2$ics <- muestras2$mean + z*sdexp/sqrt(nx)
```

Cobertura de las estimaciones por IC:

```
muestras2$cob <- (muestras2$ici < muexp) & (muexp < muestras2$ics)
ncob <- sum(muestras2$cob) # Nº de intervalos que contienen la verdadera media
ncob
```

```
## [1] 469
```

```
100*ncob/nsim     # Proporción de intervalos
```

```
## [1] 93.8
```

```
100*(1 - alfa)    # Proporción teórica bajo normalidad
```

```
## [1] 95
```

Como ejemplo ilustrativo, generamos el gráfico de los primeros 100 intervalos:

```
m <- 100
tmp <- muestras2[1:m,]
attach(tmp)
color <- ifelse(cob,"blue","red")
plot(1:m, mean, col = color, ylim = c(min(ici),max(ics)),
     xlab = "Muestra", ylab = "IC")
arrows(1:m, ici, 1:m, ics, angle = 90, length = 0.05, code = 3, col = color)
abline(h = muexp, lty = 3)
```

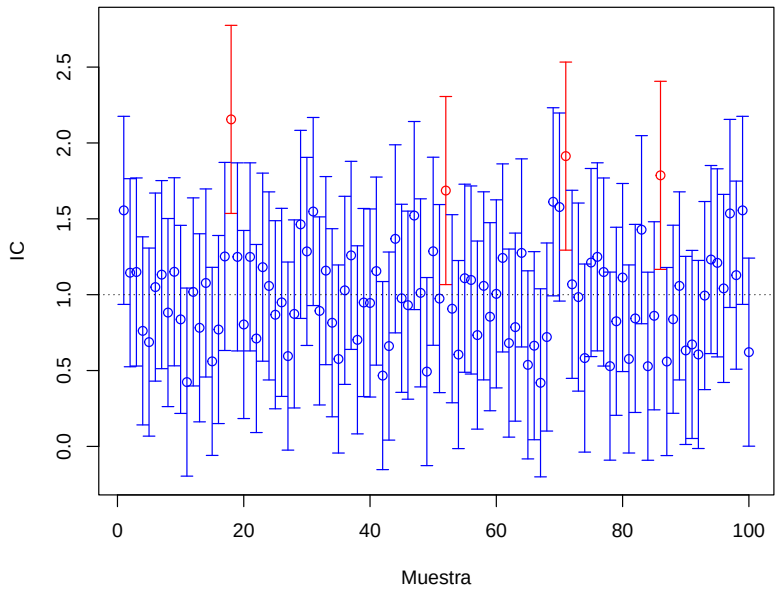


Figura 7.12: Cobertura de las estimaciones por IC (bajo normalidad).

```
detach(tmp)
```

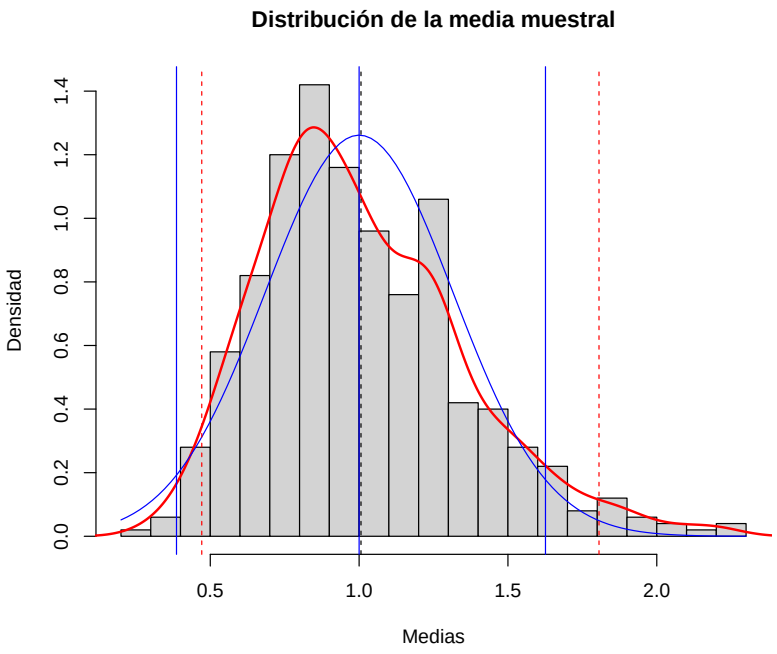
c) ¿Qué ocurre si aumentamos el tamaño muestral a 50?

Ejecutar el código del ejercicio anterior fijando `nx <- 50` y el del apartado anterior.

En los apartados b) y c) podíamos considerar bootstrap descrito en siguientes capítulos.

Podemos aproximar por simulación los intervalos de probabilidad de la media muestral (tendríamos una idea del valor esperado de lo que obtendríamos con el bootstrap percentil; en este caso el estimador es insesgado...):

```
# Distribución de la media muestral
hist(muestras2$mean, freq=FALSE, breaks="FD",
     main="Distribución de la media muestral", xlab="Medias", ylab="Densidad")
# Densidad observada (estimación)
lines(density(muestras2$mean), lwd=2, col='red')
# Densidad teórica (bajo normalidad)
curve(dnorm(x,muexp,sdexp/sqrt(nx)), col="blue", add=TRUE)
# Aproximación por simulación del valor esperado de la media muestral
abline(v=mean(muestras2$mean), lty=2)
# Valor esperado de la media muestral (teórico)
abline(v=muexp, col="blue")
# IP bajo normalidad
ic.aprox <- apply(muestras2[ ,c('ici','ics')], 2, mean)
## ic.aprox
##       ici       ics
## 0.3865199 1.6261099
# Intervalo de probabilidad para la media muestral aproximado bajo normalidad
abline(v = ic.aprox, col='blue')

# Intervalo de probabilidad para la media muestral (aproximado por simulación)
ic.sim <- quantile(muestras2$mean, c(alfa/2, 1 - alfa/2))
## ic.sim
##      2.5%     97.5%
## 0.4714233 1.8059094
# IP (aprox.)
abline(v=ic.sim, lty=2, col='red')
```



Nota: Estimaciones puntuales, por intervalo de confianza y contrastes de hipótesis para la media con varianza desconocida bajo normalidad se pueden obtener con la función `t.test`.

Ejercicio 7.5 (Intervalo de confianza Agresti-Coull para una proporción)

El Intervalo de confianza para una proporción construido usando la aproximación normal tiene un mal comportamiento cuando el tamaño de la muestra es pequeño. Una simple y efectiva mejora consiste en añadir a la muestra $2a$ elementos, a exitos y a fracasos. Así el intervalo de confianza al $(1-\alpha)\,100\%$ para una proporción mejorado es:

$$IC^{a}_{1-\alpha}(p) = \left(\tilde{p} - z_{1-\alpha/2}\sqrt{\frac{\tilde{p}(1-\tilde{p})}{\tilde{n}}}\,,\, \tilde{p} + z_{1-\alpha/2}\sqrt{\frac{\tilde{p}(1-\tilde{p})}{\tilde{n}}}\right),$$

$$\text{siendo } \tilde{n} = n + 2a,\ \tilde{p} = \frac{np + a}{\tilde{n}}.$$

En el caso de $a = 2$ se denomina IC Agresti-Coull.

(Los apartados a) y b) están basados en los ejemplos 1.5 y 1.6 de Suess y Trumbo, 2010)

a) Teniendo en cuenta que la variable aleatoria $X = n\hat{p} \sim \mathcal{B}(n, p)$, obtener y representar gráficamente la cobertura teórica del intervalo de confianza estándar ($a = 0$) de una proporción para una muestra de tamaño $n = 30$, $\alpha = 0.05$ y distintos valores de p (`p.teor <- seq(1/n, 1 - 1/n, length = 1000)`).

Parámetros:

```
n <- 30
alpha <- 0.05
adj <- 0  # (adj <- 2 para Agresti-Coull)
```

Probabilidades teóricas:

```
m <- 1000
p.teor <- seq(1/n, 1 - 1/n, length = m)
```

Posibles resultados:

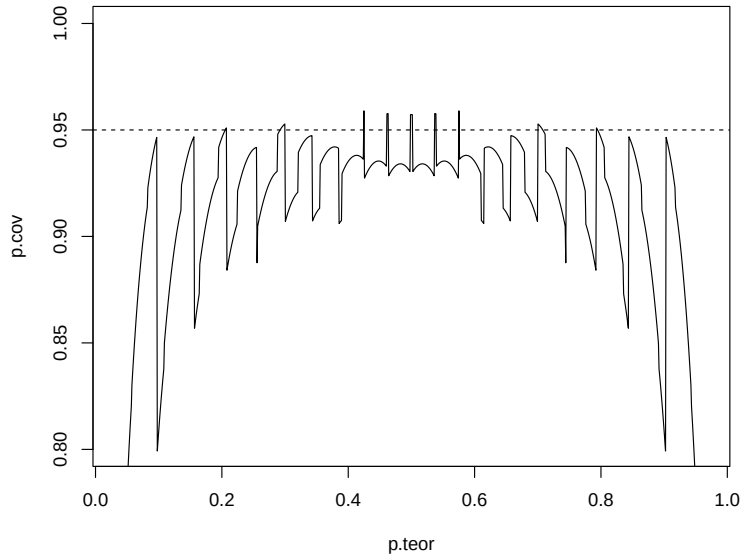
```
x <- 0:n
p.est <- (x + adj)/(n + 2 * adj)
ic.err <- qnorm(1 - alpha/2) * sqrt(p.est * (1 - p.est)/(n + 2 * adj))
lcl <- p.est - ic.err
ucl <- p.est + ic.err
```

Recorrer prob. teóricas:

```
p.cov <- numeric(m)
for (i in 1:m) {
  # cobertura de los posibles intervalos
  cover <- (p.teor[i] >= lcl) & (p.teor[i] <= ucl)
  # prob. de los posibles intervalos
  p.rel <- dbinom(x[cover], n, p.teor[i])
  # prob. total de cobertura
  p.cov[i] <- sum(p.rel)
}
```

Gráfico coberturas:

```
plot(p.teor, p.cov, type = "l", ylim = c(1 - 4 * alpha, 1))
abline(h = 1 - alpha, lty = 2)
```

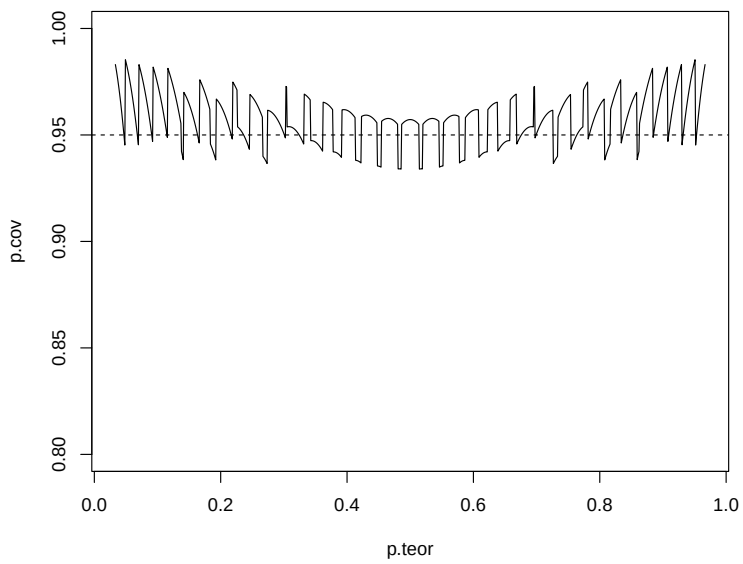


Fuente Suess y Trumbo (2010).

b) Repetir el apartado anterior considerando intervalos de confianza Agresti-Coull ($a = 2$).

Parámetros:

```
n <- 30
alpha <- 0.05
adj <- 2  # Agresti-Coull

# Probabilidades teóricas:
m <- 1000
p.teor <- seq(1/n, 1 - 1/n, length = m)
# Posibles resultados:
x <- 0:n
p.est <- (x + adj)/(n + 2 * adj)
ic.err <- qnorm(1 - alpha/2) * sqrt(p.est * (1 - p.est)/(n + 2 * adj))
lcl <- p.est - ic.err
ucl <- p.est + ic.err
# Recorrer prob. teóricas:
p.cov <- numeric(m)
for (i in 1:m) {
  # cobertura de los posibles intervalos
  cover <- (p.teor[i] >= lcl) & (p.teor[i] <= ucl)
  # prob. de los posibles intervalos
  p.rel <- dbinom(x[cover], n, p.teor[i])
  # prob. total de cobertura
  p.cov[i] <- sum(p.rel)
}
# Gráfico coberturas:
plot(p.teor, p.cov, type = "l", ylim = c(1 - 4 * alpha, 1))
abline(h = 1 - alpha, lty = 2)
```



c) Repetir el apartado anterior empleando simulación para aproximar la cobertura.

Parámetros:

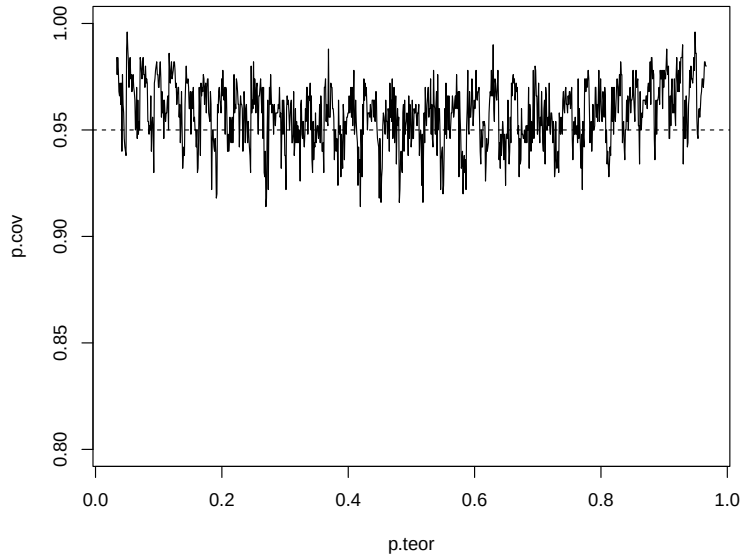
```
n <- 30
alpha <- 0.05
adj <- 2  #' (2 para Agresti-Coull)

set.seed(54321)
nsim <- 500
# Probabilidades teóricas:
m <- 1000
p.teor <- seq(1/n, 1 - 1/n, length = m)
```

Recorrer prob. teóricas:

```
# m <- length(p.teor)
p.cov <- numeric(m)
for (i in 1:m) {
  # Equivalente a simular nsim muestras de tamaño n
  # ry <- matrix(rbinom(n*nsim, 1, p.teor[i]), ncol=n)
  # rx <- apply(ry, 1, sum)
  rx <- rbinom(nsim, n, p.teor[i])
  p.est <- (rx + adj)/(n + 2 * adj)
  ic.err <- qnorm(1 - alpha/2) * sqrt(p.est * (1 - p.est)/(n + 2 * adj))
  p.cov[i] <- mean( abs(p.est - p.teor[i]) < ic.err )
}
```

Representar:

```
plot(p.teor, p.cov, type = "l", ylim = c(1 - 4 * alpha, 1))
abline(h = 1 - alpha, lty = 2)
```



Como ya se comentó, el caso de ajustar un modelo a los datos y realizar simulaciones a partir de ese modelo ajustado para aproximar las características de interés de un estadístico, se denomina también bootstrap paramétrico. Para más detalles ver por ejemplo la Sección 3.1 de Cao y Fernández-Casal (2020). En este libro, en las secciones 4.6.2 y B.3.2, se incluyen ejemplos adicionales de estudios de simulación.

7.4.3 Contrastes de hipótesis

Ver Capítulo 5 de Cao y Fernández-Casal (2020).

Ejercicio 7.6 (Test de Kolmogorov-Smirnov)

En la Sección 2.3 del Tema 2 se propuso el análisis de la bondad de ajuste de un generador de números pseudo-aleatorios mediante el test de Kolmogorov-Smirnov (ver Sección A.1.5). Sin embargo, si H_0 es compuesta (los parámetros desconocidos se estiman por máxima verosimilitud y se trabaja con $\widehat{F}_0$) los cuantiles de la distribución (asintótica) de D_n pueden ser demasiado conservativos y sería preferible utilizar la distribución exacta.

a) Analizar el comportamiento del contraste de Kolmogorov-Smirnov para contrastar normalidad empleando repetidamente este test, considerando 1000 pruebas con muestras de tamaño 30 de una $\mathcal{N}(0,1)$. Comparar gráficamente el ajuste de la distribución del p-valor a la de referencia (estudiar el tamaño del contraste).

   Valores iniciales:

```
set.seed(54321)
nx <- 30
mx <- 0
sx <- 1
nsim <- 1000
estadistico <- numeric(nsim)
pvalor <- numeric(nsim)
```

   Realizar contrastes

```
for(isim in 1:nsim) {
  rx <- rnorm(nx, mx, sx)
  tmp <- ks.test(rx, "pnorm", mean(rx), sd(rx))
```

```
  estadistico[isim] <- tmp$statistic
  pvalor[isim] <- tmp$p.value
}
```

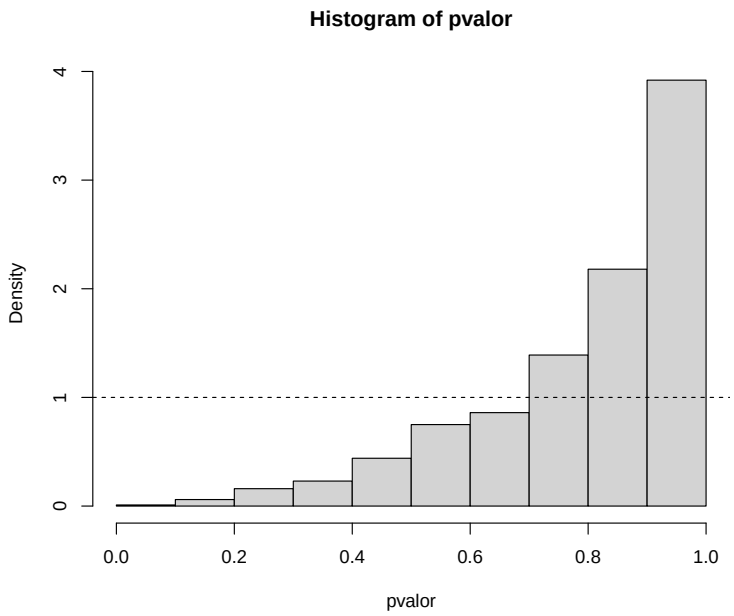
Proporción de rechazos:

```
{
  cat("\nProporción de rechazos al 1% =", mean(pvalor < 0.01), "\n")
  cat("Proporción de rechazos al 5% =", mean(pvalor < 0.05), "\n")
  cat("Proporción de rechazos al 10% =", mean(pvalor < 0.1), "\n")
}
```

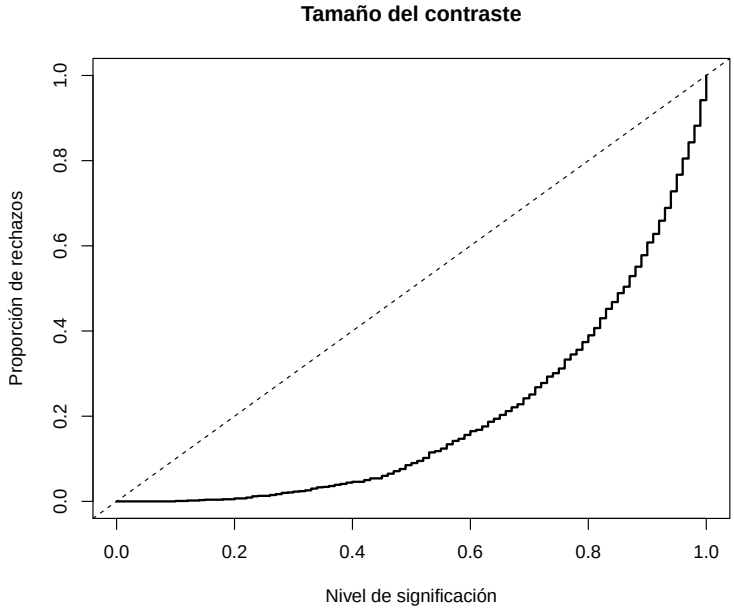
```
##
## Proporción de rechazos al 1% = 0
## Proporción de rechazos al 5% = 0
## Proporción de rechazos al 10% = 0.001
```

Análisis de los p-valores:

```
hist(pvalor, freq=FALSE)
abline(h=1, lty=2)   # curve(dunif(x,0,1), add=TRUE)
```




```
# Distribución empírica
curve(ecdf(pvalor)(x), type = "s", lwd = 2,
      main = 'Tamaño del contraste', ylab = 'Proporción de rechazos',
      xlab = 'Nivel de significación')
abline(a=0, b=1, lty=2)   # curve(punif(x, 0, 1), add = TRUE)
```



b) Repetir el apartado anterior considerando el test de Lilliefors (rutina `lillie.test` del paquete `nortest`).

```
library(nortest, quietly = TRUE)
```

Valores iniciales:

```
set.seed(54321)
nx <- 30
mx <- 0
sx <- 1
nsim <- 1000
estadistico <- numeric(nsim)
pvalor <- numeric(nsim)
```

Realizar contrastes

```
for(isim in 1:nsim) {
  rx <- rnorm(nx, mx, sx)
  # tmp <- ks.test(rx, "pnorm", mean(rx), sd(rx))
  tmp <- lillie.test(rx)
  estadistico[isim] <- tmp$statistic
  pvalor[isim] <- tmp$p.value
}
```

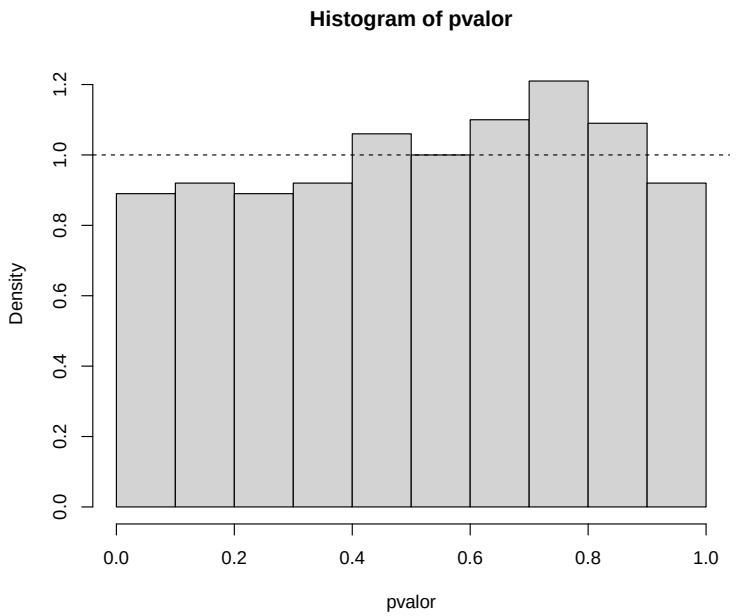
Proporción de rechazos:

```
{
  cat("\nProporción de rechazos al 1% =", mean(pvalor < 0.01), "\n")
  cat("Proporción de rechazos al 5% =", mean(pvalor < 0.05), "\n")
  cat("Proporción de rechazos al 10% =", mean(pvalor < 0.1), "\n")
}
```

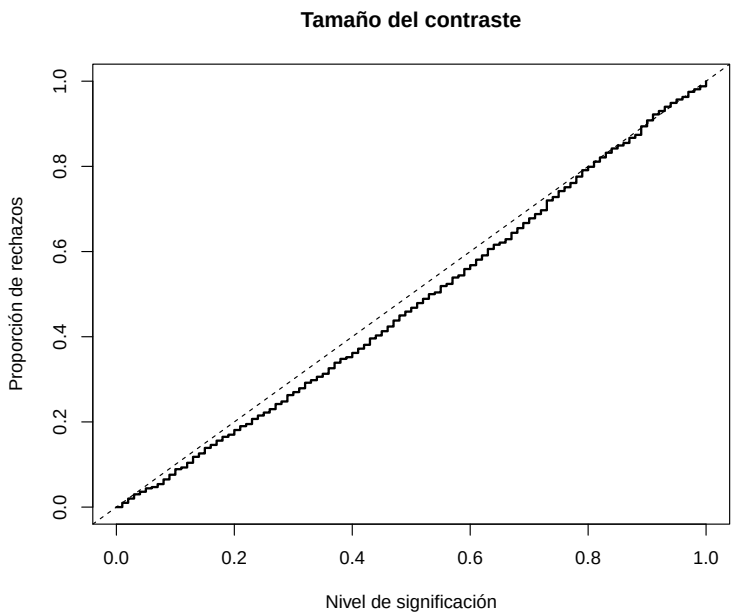
```
##
## Proporción de rechazos al 1% = 0.01
## Proporción de rechazos al 5% = 0.044
## Proporción de rechazos al 10% = 0.089
```

Análisis de los p-valores:

```
hist(pvalor, freq=FALSE)
abline(h=1, lty=2)    # curve(dunif(x,0,1), add=TRUE)
```




```
# Distribución empírica
curve(ecdf(pvalor)(x), type = "s", lwd = 2, main = 'Tamaño del contraste',
      ylab = 'Proporción de rechazos', xlab = 'Nivel de significación')
abline(a=0, b=1, lty=2)    # curve(punif(x, 0, 1), add = TRUE)
```



c) Repetir el apartado a) contrastando una distribución exponencial y considerando 500 pruebas con muestras de tamaño 30 de una $Exp(1)$.

Valores iniciales:

```
set.seed(54321)
nx <- 30
```

```
ratex <- 1
nsim <- 500
estadistico <- numeric(nsim)
pvalor <- numeric(nsim)
```

Realizar contrastes

```
for(isim in 1:nsim) {
  rx <- rexp(nx, ratex)
  tmp <- ks.test(rx, "pexp", 1/mean(rx))
  estadistico[isim] <- tmp$statistic
  pvalor[isim] <- tmp$p.value
}
```

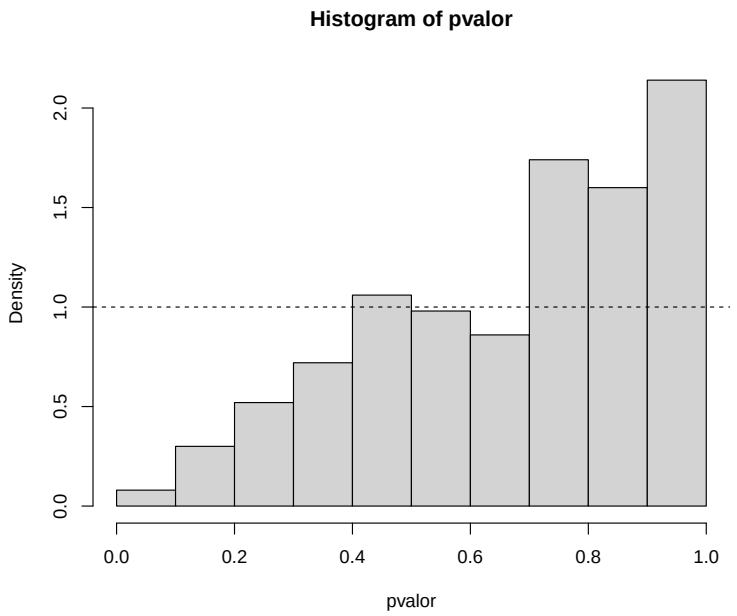
Proporción de rechazos:

```
{
  cat("\nProporción de rechazos al 1% =", mean(pvalor < 0.01), "\n")
  cat("Proporción de rechazos al 5% =", mean(pvalor < 0.05), "\n")
  cat("Proporción de rechazos al 10% =", mean(pvalor < 0.1), "\n")
}
```

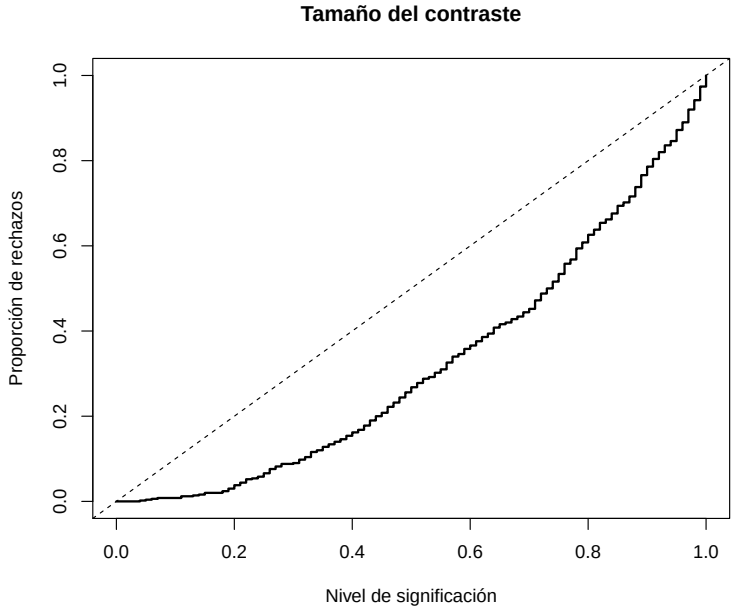
```
##
## Proporción de rechazos al 1% = 0
## Proporción de rechazos al 5% = 0.004
## Proporción de rechazos al 10% = 0.008
```

Análisis de los p-valores:

```
hist(pvalor, freq=FALSE)
abline(h=1, lty=2)    # curve(dunif(x,0,1), add=TRUE)
```




```
# Distribución empírica
curve(ecdf(pvalor)(x), type = "s", lwd = 2,
      main = 'Tamaño del contraste', ylab = 'Proporción de rechazos',
      xlab = 'Nivel de significación')
abline(a=0, b=1, lty=2)    # curve(punif(x, 0, 1), add = TRUE)
```



d) Diseñar una rutina que permita realizar el contraste KS de bondad de ajuste de una variable exponencial aproximando el p-valor por simulación y repetir el apartado anterior empleando esta rutina.

```
ks.exp.sim <- function(x, nsim = 10^3) {
  DNAME <- deparse(substitute(x))
  METHOD <- "Kolmogorov-Smirnov Test of pexp by simulation"
  n <- length(x)
  RATE <- 1/mean(x)
  ks.exp.stat <- function(x, rate=1/mean(x)) {
    DMinus <- pexp(sort(x), rate=rate) - (0:(n - 1))/n
    DPlus <- 1/n - DMinus
    Dn = max(c(DMinus, DPlus))
  }
  STATISTIC <- ks.exp.stat(x, rate = RATE)
  names(STATISTIC) <- "Dn"
  # PVAL <- 0
  # for(i in 1:nsim) {
  #   rx <- rexp(n, rate = RATE)
  #   if (STATISTIC <= ks.exp.stat(rx)) PVAL <- PVAL+1
  # }
  # PVAL <- PVAL/nsim
  # PVAL <- PVAL/(nsim + 1)
  # PVAL <- (PVAL + 1)/(nsim + 2)
  rx <- matrix(rexp(n*nsim, rate = RATE), ncol=n)
  PVAL <- mean(STATISTIC <= apply(rx, 1, ks.exp.stat))
  return(structure(list(statistic = STATISTIC, alternative = "two.sided",
                        p.value = PVAL, method = METHOD, data.name = DNAME),
                        class = "htest"))
}
```

Simulación:

```
set.seed(54321)
nx <- 30
ratex <- 1
```

```
nsim <- 500
estadistico <- numeric(nsim)
pvalor <- numeric(nsim)
```

Realizar contrastes

```
for(isim in 1:nsim) {
  rx <- rexp(nx, ratex)
  # tmp <- ks.test(rx, "pexp", 1/mean(rx))
  tmp <- ks.exp.sim(rx, nsim = 200)
  estadistico[isim] <- tmp$statistic
  pvalor[isim] <- tmp$p.value
}
```

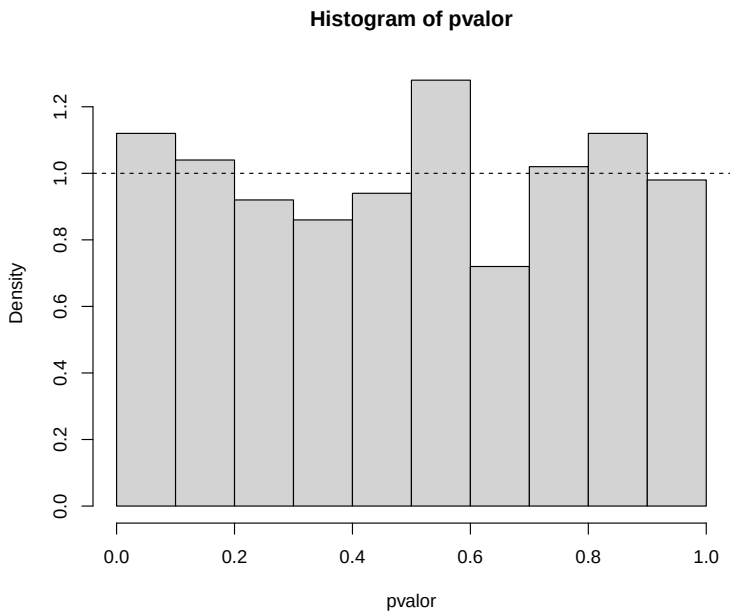
Proporción de rechazos:

```
{
  cat("\nProporción de rechazos al 1% =", mean(pvalor < 0.01), "\n")
  cat("Proporción de rechazos al 5% =", mean(pvalor < 0.05), "\n")
  cat("Proporción de rechazos al 10% =", mean(pvalor < 0.1), "\n")
}
```

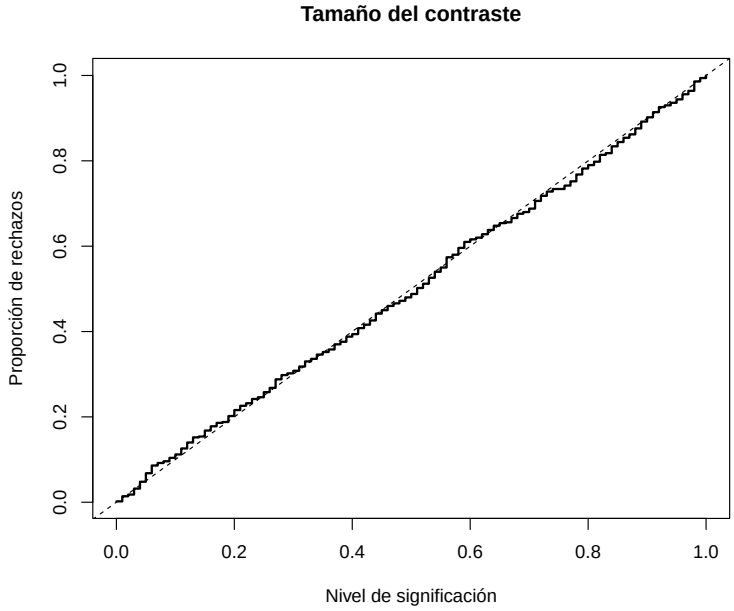
```
##
## Proporción de rechazos al 1% = 0.008
## Proporción de rechazos al 5% = 0.058
## Proporción de rechazos al 10% = 0.106
```

Análisis de los p-valores:

```
hist(pvalor, freq=FALSE)
abline(h=1, lty=2)    # curve(dunif(x,0,1), add=TRUE)
```




```
# Distribución empírica
curve(ecdf(pvalor)(x), type = "s", lwd = 2,
      main = 'Tamaño del contraste', ylab = 'Proporción de rechazos',
      xlab = 'Nivel de significación')
abline(a=0, b=1, lty=2)    # curve(punif(x, 0, 1), add = TRUE)
```

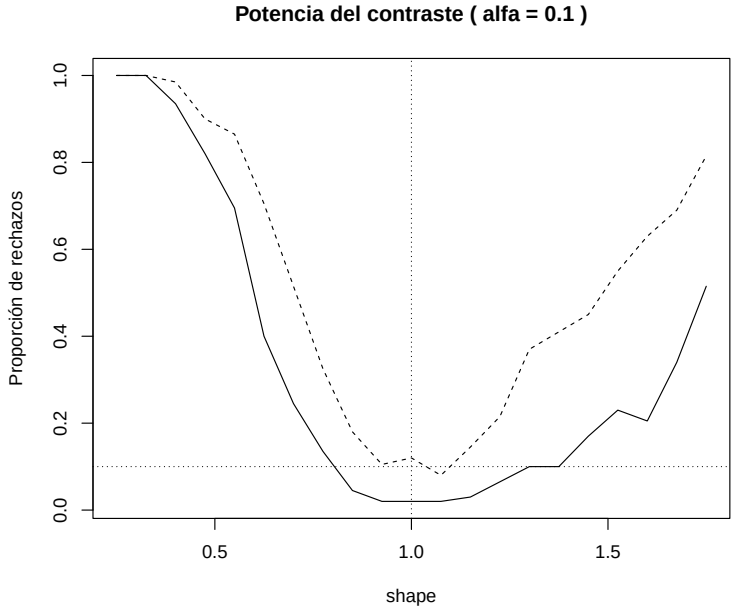


e) Estudiar la potencia de los contrastes de los apartados c) y d), considerando como alternativa una distribución Weibull.

La distribución exponencial es un caso particular de la Weibull: `dexp(x, ratex) == dweibull(x, 1, 1/ratex)`. Estudiamos lo que ocurre al desplazar `dweibull(x, shape, 1/ratex)` con `0 < shape < 2`.

CUIDADO: las simulaciones pueden requerir de mucho tiempo de computación (consideramos valores pequeños de `nx` y `nsim` en datos y en `ks.exp.sim`).

```
set.seed(54321)
nx <- 20
ratex <- 1     # Puede ser interesante representarlo variando rate
nsim <- 200
alfa <- 0.1    # Puede ser interesante representarlo variando alfa


shapex <- seq(0.25, 1.75, len=21)
preject <- numeric(length(shapex)) # Porporciones de rechazos con ks.test
ks.test.p <- function(x) ks.test(x, "pexp", 1/mean(x))$p.value
preject2 <- preject # Porporciones de rechazos con ks.exp.sim
ks.exp.sim.p <- function(x) ks.exp.sim(x, 200)$p.value

for (i in seq_along(shapex)) {
  rx <- matrix(rweibull(nx*nsim, shape = shapex[i], scale = 1/ratex), ncol=nx)
  preject[i] <- mean( apply(rx, 1, ks.test.p) <= alfa )
  preject2[i] <- mean( apply(rx, 1, ks.exp.sim.p) <= alfa )
}

plot(shapex, preject, type="l", main = paste("Potencia del contraste ( alfa =", alfa, ")"),
     xlab = "shape", ylab = "Proporción de rechazos")
lines(shapex, preject2, lty = 2)
abline(h = alfa, v = 1, lty = 3)
```



El estadístico de Kolmogorov-Smirnov `Dn = max(c(DMinus, DPlus))` tiene ventajas desde el punto de vista teórico, pero puede no ser muy potente para detectar diferencias entre la distribución bajo la hipótesis nula y la distribución de los datos. La ventaja de la aproximación por simulación es que no estamos atados a resultados teóricos y podemos emplear el estadístico que se considere oportuno (la principal desventaja es el tiempo de computación). Por ejemplo, podríamos pensar en utilizar como estadístico la suma de los errores en valor absoluto del correspondiente gráfico PP, y solo habría que cambiar el estadístico `Dn` en la función `ks.exp.sim` por `Dn = sum(abs( (1:n - 0.5)/n - pexp(sort(x), rate=rate) ))`.

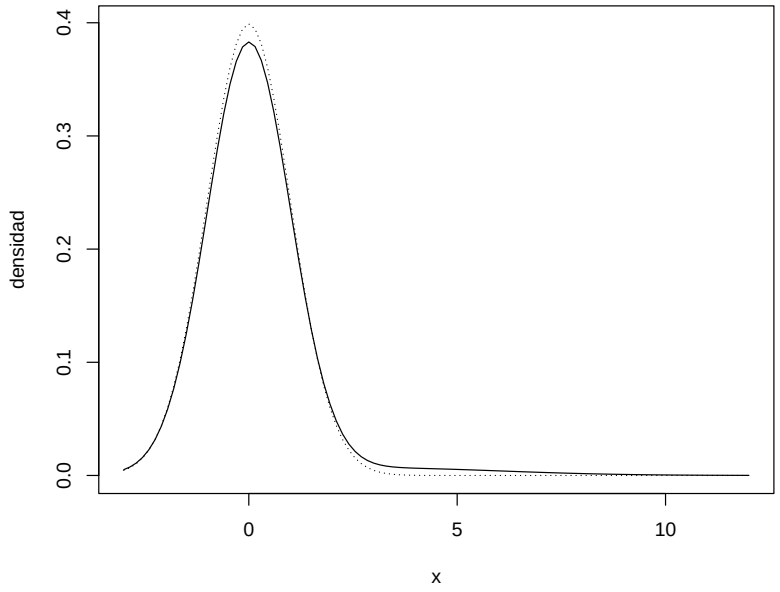
7.4.4 Comparación de estimadores

Ejercicio 7.7 (Comparación de la eficiencia de la media muestral y de la mediana bajo contaminación) Supongamos que estamos interesados en estudiar el efecto de datos atípicos en la estimación de la media teórica mediante la media y la mediana muestrales. Consideramos una variable aleatoria con distribución normal contaminada, en la que una observación procede de una $N(0,1)$ con probabilidad 0.95 y de una $N(3,3^2)$ con probabilidad 0.05 (mixtura). Se puede generar una muestra de esta variable (mixtura) mediante el método de composición descrito en la Sección 4.4, por ejemplo empleando el siguiente código:

```
p.sim <- rbinom(n, 1, 0.05)
dat.sim <- rnorm(n, 3*p.sim, 1+2*p.sim)
```

Podemos comparar la densidad objetivo con la de los valores contaminados:

```
curve(dnorm(x, 0, 1), -3, 12, ylab = 'densidad', lty = 3)
curve(0.95*dnorm(x, 0, 1) + 0.05*dnorm(x, 3, 3), add = TRUE)
```



Nota: Como se comentó en la Sección 4.4, también es habitual simular este tipo de datos generando un porcentaje alto de valores (en este caso un 95%) de la distribución base ($N(0,1)$) y el resto (5%) de la distibución "contaminadora" ($N(3,3^2)$), aunque se suele considerar un porcentaje de contaminación del 1% o inferior (además, como en este caso concreto no va importar el orden, no sería necesario combinar aleatoriamente los valores).

a) Aproximar mediante simulación (500 generaciones) el sesgo y error estándar de la media y la mediana en el caso de una muestra de tamaño $n = 100$ (suponiendo que se pretende estimar la media no contaminada 0).

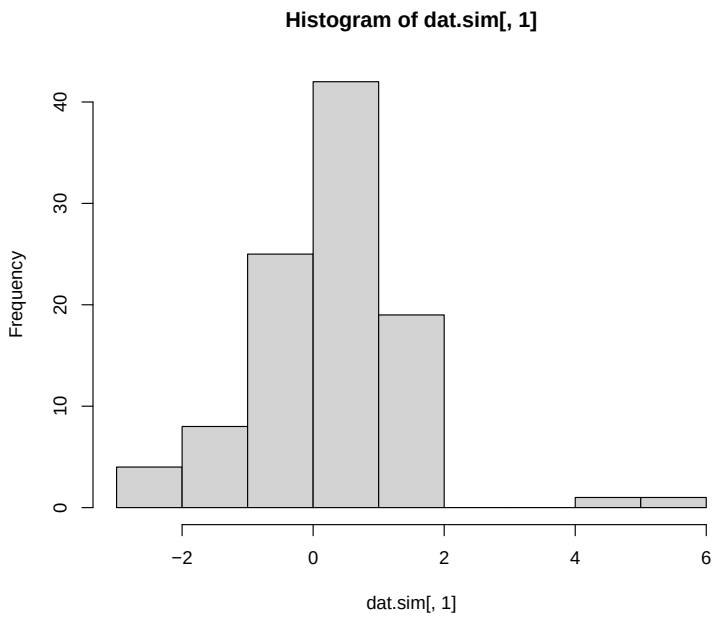
```
# media y mediana
xsd <- 1
xmed <- 0
ndat <- 100
nsim <- 500

# for (isim in 1:nsim) # evitar matrix y apply
set.seed(1)
ntsim <- ndat*nsim
p.sim <- rbinom(ntsim, 1, 0.05)
dat.sim <- rnorm(ntsim, 3*p.sim, 1+2*p.sim)
dat.sim <- matrix(dat.sim, ncol=nsim)
```

Cada columna es una muestra

```
str(dat.sim[,1])
```

```
##  num [1:100] 0.197 -0.42 1.163 -0.406 0.744 ...
```

```
hist(dat.sim[,1])
```



Calculamos los estimadores:

```
mean.sim <- apply(dat.sim, 2, mean)
median.sim <- apply(dat.sim, 2, median)
```

Estimamos sus características:

```
mean(mean.sim) # Coincide con el sesgo (media teórica es 0)
```

```
## [1] 0.1459986
```

```
sd(mean.sim)
```

```
## [1] 0.1349537
```

```
mean(median.sim) # Coincide con el sesgo (media teórica es 0)
```

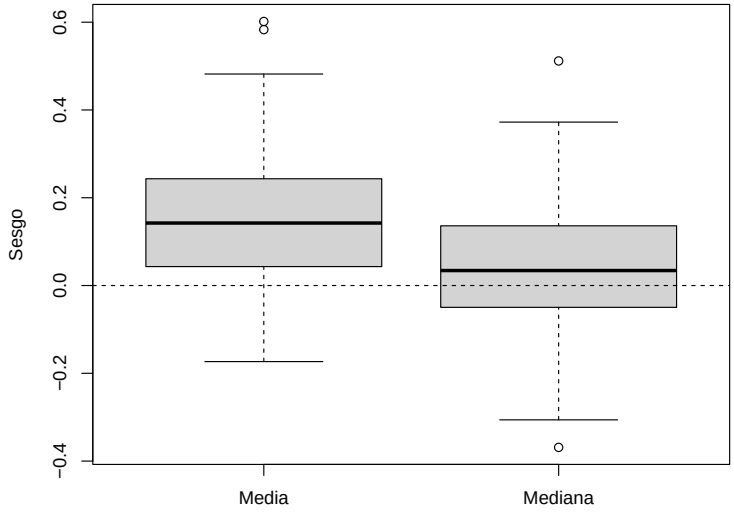
```
## [1] 0.04453509
```

```
sd(median.sim)
```

```
## [1] 0.1300611
```

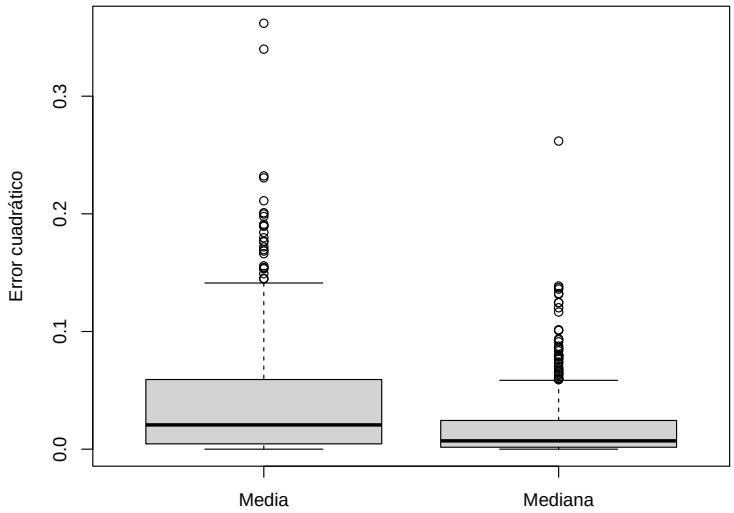
Sesgo:

```
boxplot(mean.sim-xmed, median.sim-xmed,
      names=c("Media","Mediana"), ylab="Sesgo")
abline(h = 0, lty = 2)
```



Error cuadrático:

```
boxplot((mean.sim-xmed)^2, (median.sim-xmed)^2,
      names=c("Media","Mediana"), ylab="Error cuadrático")
```



Estadísticos error cuadrático:

```
# SE media
summary((mean.sim-xmed)^2)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0000005 0.0045072 0.0206272 0.0394917 0.0591531 0.3619587
```

```
# SE mediana
summary((median.sim-xmed)^2)
```

```
##      Min.   1st Qu.    Median      Mean   3rd Qu.      Max.
## 0.0000001 0.0016481 0.0070625 0.0188654 0.0243903 0.2618368
```

Capítulo 8

Métodos de remuestreo

Un par de notas:

- Etimología: se denomina *bootstrap* a la cinta de la bota (oreja lateral o trasera para ayudar a calzarse las botas).
- Modismo anglosajón: "to pull oneself up by one's bootstraps", que podríamos traducir como resolver un problema con medios propios, sin la ayuda de otros. Se cree que esta frase está basada en un libro del siglo XVIII:

  > "The Baron had fallen to the bottom of a deep lake. Just when it looked like all was lost, he thought to pick himself up by his own bootstraps".
  >
  > — Raspe, Rudolph Erich (The Surprising Adventures of Baron Munchausen, 1785)

El bootstrap es un procedimiento estadístico que sirve para aproximar características de la distribución en el muestreo de un estadístico. Para ello se emplea (normalmente) simulación, generando un gran número de muestras mediante algún tipo de remuestreo de la muestra original.

Su ventaja principal es que no requiere hipótesis sobre el mecanismo generador de los datos (aunque los resultados asintóticos requieren de hipótesis generales). Por lo que son de especial utilidad cuando no se dispone la distribución exacta del estadístico y no es posible o adecuado emplear la distribución asintótica.

En este capítulo se incluye una breve introducción al bootstrap desde un punto de vista aplicado. Para información adicional, inluyendo resultados teóricos, ver por ejemplo Davison y Hinkley (1997) o Cao y Fernández-Casal (2020).

En este libro nos centraremos principalmente en los métodos de remuestreo bootstrap, aunque hay otros tipos de remuestreo como el *jackknife*, para la aproximación del sesgo y varianza de un estimador (ver Sección 2.2 de Cao y Fernández-Casal, 2020), o los empleados en contrastes de permutaciones (ver Sección 5.3 de Cao y Fernández-Casal, 2020).

En los siguientes capítulos se tratarán algunas de las principales aplicaciones de los métodos bootstrap. Entre ellas podríamos destacar:

- Aproximación del sesgo y de la varianza de un estimador.
- Construcción de intervalos de confianza.
- Contraste de hipótesis.

También la simulación (condicional) de nuevas observaciones o la estimación de la probabilidad de superar un determinado umbral (probabilidad de riesgo).

8.1 Introducción

En primer lugar nos centraremos en la idea original del bootstrap uniforme (Efron, 1979; también denominado bootstrap no paramétrico o naïve), que se expondrá de manera más formal en la Sección 8.2.

Supongamos que $\mathbf{X} = (X_1, \dots, X_n)$ es una una muestra aleatoria simple (m.a.s.) de una población con distribución F_θ y que estamos interesados en hacer inferencia sobre θ empleando un estimador $\hat{\theta} = T(\mathbf{X})$.

La idea es aproximar características poblacionales por las correspondientes de la distribución empírica de los datos observados. Se trata de imitar el experimento de muestreo en la población real, pero empleando la distribución empírica en lugar de la distribución teórica F_θ desconocida. Al conocer el mecanismo que genera los datos en el universo bootstrap, se puede emplear Monte Carlo para simularlo. En el caso i.i.d. esto puede ser implementado mediante remuestreo, realizando repetidamente **muestreo aleatorio con reemplazamiento del conjunto de datos original** (manteniendo el tamaño muestral).

Para aproximar la distribución en el muestreo por Monte Carlo, se genera un número grande B de réplicas bootstrap:

- $\mathbf{X}^{*(b)} = \left(X_1^{*(b)}, \dots, X_n^{*(b)}\right)$ muestra bootstrap (remuestra), obtenida mediante muestreo con reemplazamiento de $\mathbf{X} = (X_1, \dots, X_n)$.
- $\hat{\theta}^{*(b)} = T\left(\mathbf{X}^{*(b)}\right)$ valor del estadístico en la muestra bootstrap (réplica bootstrap del estadístico).

para $b = 1, \dots, B$.

La idea original (bootstrap natural, Efron) es que la variabilidad de $\hat{\theta}_b^*$ en torno a $\hat{\theta}$ aproxima la variabilidad de $\hat{\theta}$ en torno a θ: **la distribución de** $\hat{\theta}_b^* - \hat{\theta}$ (en el universo bootstrap) **aproxima la distribución de** $\hat{\theta} - \theta$ (en la población).

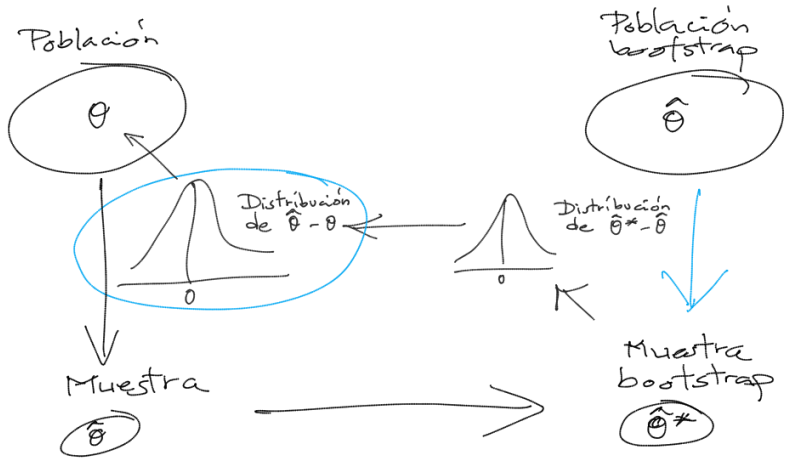


Figura 8.1: Esquema de la idea del boostrap (de Efron).

En general podríamos decir que **la muestra es a la población lo que la muestra bootstrap es a la muestra**.

Ejemplo 8.1 (aproximación bootstrap de la distribución de la media muestral)

Como ejemplo ilustrativo consideramos una muestra simulada de tamaño $n = 100$ de una normal estándar y la media muestral como estimador de la media teórica:

```
set.seed(1)
n <- 100
mean_teor <- 0
```

```
sd_teor <- 1
muestra <- rnorm(n, mean = mean_teor, sd = sd_teor)
```

El valor del estadístico en la muestra es:

```
estadistico <- mean(muestra)
```

Representamos la distribución de la muestra [Figura 8.2]:

```
hist(muestra, freq = FALSE, xlim = c(-3, 3),
     main = '', xlab = 'x', ylab = 'densidad')
abline(v = estadistico, lty = 2)
curve(dnorm, col = "blue", add = TRUE)
abline(v = mean_teor, col = "blue", lty = 2)
```

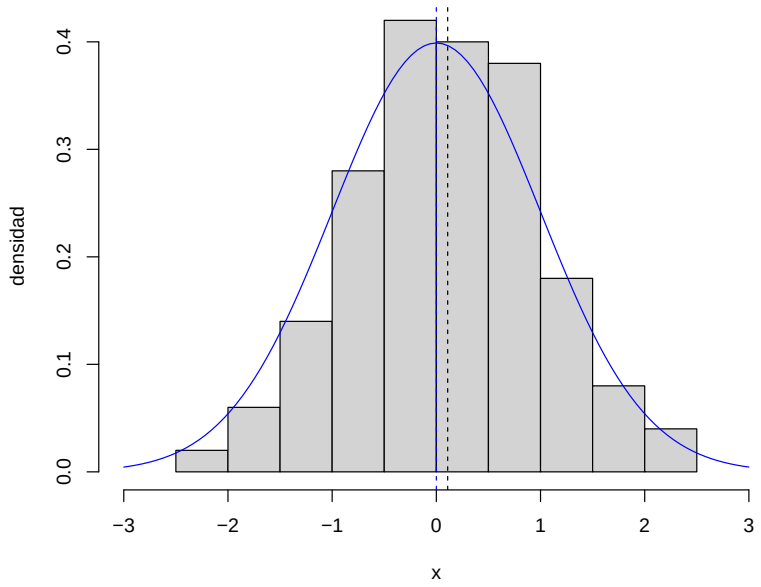


Figura 8.2: Distribución de la muestra simulada (y distribución teórica en azul).

Como aproximación de la distribución poblacional, desconocida en la práctica, siempre podemos considerar la distribución empírica (o una versión suavizada: bootstrap suavizado; Sección 9.3). Alternativamente podríamos asumir un modelo paramétrico y estimar los parámetros a partir de la muestra (bootstrap paramétrico; Sección 9.2 [Figura 8.3].

```
# Distribución bootstrap uniforme
curve(ecdf(muestra)(x), xlim = c(-3, 3), ylab = "F(x)", type = "s")
# Distribución bootstrap paramétrico (asumiendo normalidad)
curve(pnorm(x, mean(muestra), sd(muestra)), lty = 2, add = TRUE)
# Distribución teórica
curve(pnorm, col = "blue", add = TRUE)
legend("bottomright", legend = c("Empírica", "Aprox. paramétrica", "Teórica"),
       lty = c(1, 2, 1), col = c("black","black", "blue"))
```

En este caso (bootstrap uniforme) generamos las réplicas bootstrap empleando la distribución empírica:

```
set.seed(1)
B <- 1000
```

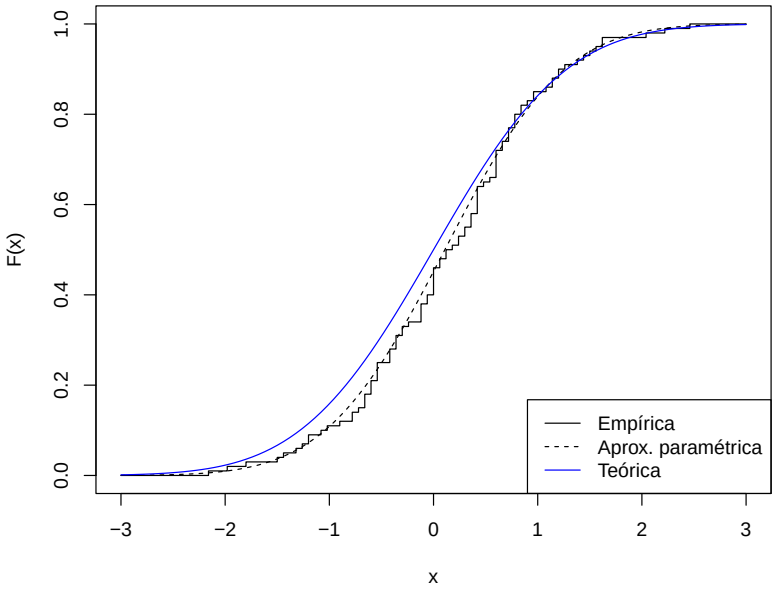


Figura 8.3: Distribución teórica de la muestra simulada y distintas aproximaciones.

```
estadistico_boot <- numeric(B)
for (k in 1:B) {
    remuestra <- sample(muestra, n, replace = TRUE)
    estadistico_boot[k] <- mean(remuestra)
}
```

Podríamos emplear directamente las réplicas bootstrap del estimador para aproximar la distribución en el muestreo de la media muestral (esto es lo que se conoce como bootstrap percentil directo, o simplemente bootstrap percentil):

```
hist(estadistico_boot, freq = FALSE, xlim = c(-0.2, 0.5),
     ylab = "Densidad", main = "")
# Valor esperado bootstrap del estadístico
mean_boot <- mean(estadistico_boot)
abline(v = mean_boot, lwd = 2)
# abline(v = estadistico, col = "blue")

# Distribución poblacional
curve(dnorm(x, mean_teor, sd_teor/sqrt(n)), col = "blue", add = TRUE)
abline(v = 0, col = "blue", lty = 2)
```

Sin embargo, especialmente si el estimador es sesgado, puede ser preferible emplear la distribución de $\widehat{\theta}_b^* - \widehat{\theta}$ como aproximación de la distribución de $\widehat{\theta} - \theta$ (bootstrap natural, básico o percentil básico):

```
hist(estadistico_boot - estadistico, freq = FALSE,
     ylab = "Densidad", main = "")
# Distribución teórica
curve(dnorm(x, 0, sd_teor/sqrt(n)), col = "blue", add = TRUE)
abline(v = 0, col = "blue", lty = 2)
```

Sin embargo, asintóticamente ambos procedimientos son equivalentes[1] y pueden dar lugar a los mismos

[1]Por este motivo en algunas referencias más teóricas no se diferencia entre ambos métodos y se denominan simple-

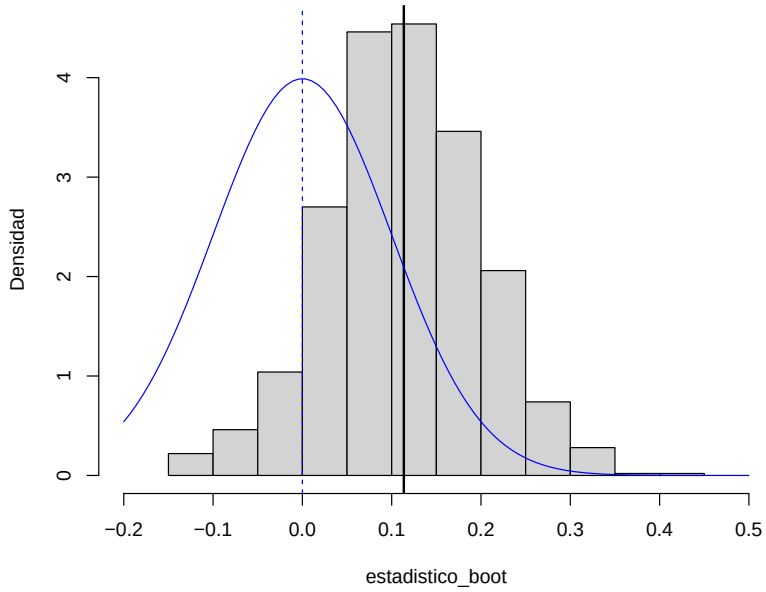


Figura 8.4: Aproximación de la distribución de la media muestral centrada mediante bootstrap percentil (uniforme).

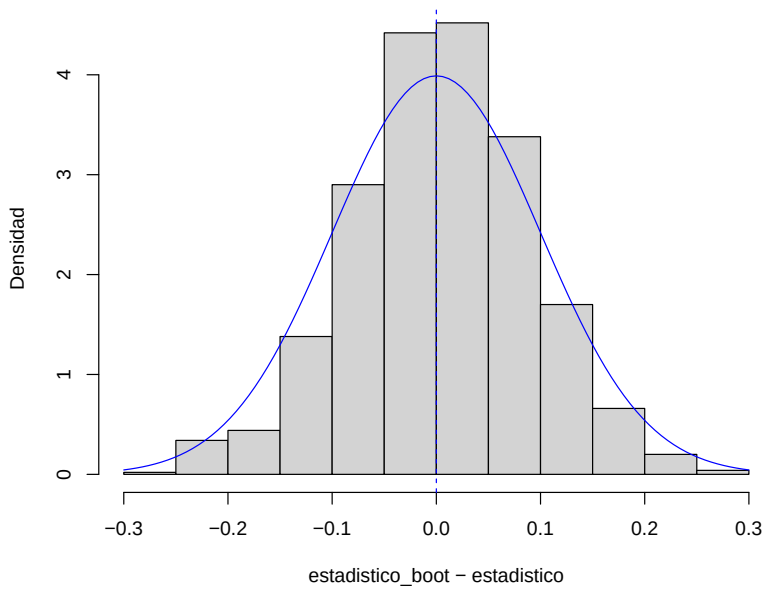


Figura 8.5: Aproximación de la distribución de la media muestral mediante bootstrap natural (uniforme).

resultados en determinados problemas de inferencia. Por ejemplo en la aproximación del sesgo y de la varianza de un estimador (Sección **??**):

mente bootstrap percentil.

```
# Sesgo (teor=0)
mean_boot - estadistico # mean(estadistico_boot - estadistico)
```

```
## [1] 0.004714973
```

```
# Error estándar
sd(estadistico_boot) # sd(estadistico_boot - estadistico)
```

```
## [1] 0.08610306
```

```
# Error estándar teórico
sd_teor/sqrt(n)
```

```
## [1] 0.1
```

8.2 El Bootstrap uniforme

Suponemos que $\mathbf{X} = (X_1, \dots, X_n)$ es una una muestra aleatoria simple (m.a.s.) de una población con distribución F y que estamos interesados en hacer inferencia sobre $\theta = \theta(F)$ empleando un estimador $\hat{\theta} = T(\mathbf{X})$. Para ello nos gustaría conocer la distribución en el muestreo de un estadístico $R(\mathbf{X}, F)$, función del estimador (y por tanto de la muestra) y de la distribución poblacional. Por ejemplo el estimador studentizado:

$$R = R(\mathbf{X}, F) = \frac{\hat{\theta} - \theta}{\sqrt{\widehat{Var}(\hat{\theta})}}.$$

A veces podemos calcular directamente la distribución de $R(\mathbf{X}, F)$, aunque suele depender de cantidades poblacionales, no conocidas en la práctica. Otras veces sólo podemos llegar a aproximar la distribución de $R(\mathbf{X}, F)$ cuando $n \to \infty$. Por ejemplo, bajo normalidad $X_i \overset{i.i.d.}{\sim} \mathcal{N}(\mu, \sigma^2)$, si el parámetro de interés es la media

$$\theta(F) = \mu = \int x \, dF(x) = \int x f(x) \, dx$$

y consideramos como estimador la media muestral

$$\hat{\theta} = T(\mathbf{X}) = \theta(F_n) = \int x \, dF_n(x) = \bar{X}.$$

Como normalmente en la práctica la varianza es desconocida, podríamos considerar el estadístico:

$$R = R(\mathbf{X}, F) = \sqrt{n}\frac{\bar{X} - \mu}{S_{n-1}} \sim t_{n-1},$$

donde S_{n-1}^2 es la cuasivarianza muestral:

$$S_{n-1}^2 = \frac{1}{n-1}\sum_{j=1}^{n}\left(X_j - \bar{X}\right)^2.$$

Si F no es normal entonces la distribución de R ya no es una t_{n-1}, aunque, bajo condiciones muy generales, $R \xrightarrow{d} \mathcal{N}(0, 1)$.

En el universo bootstrap se reemplaza la distribución poblacional (desconocida) F por una estimación, $\hat{F}$, de la misma. A partir de la aproximación $\hat{F}$ podríamos generar, condicionalmente a la muestra observada, remuestras

$$\mathbf{X}^* = (X_1^*, \dots, X_n^*)$$

con distribución $X_i^* \sim \hat{F}$, que demoninaremos remuestras bootstrap. Por lo que podemos hablar de la distribución en el remuestreo de

$$R^* = R\left(\mathbf{X}^*, \hat{F}\right),$$

denominada *distribución bootstrap*.

Una de las consideraciones más importantes al diseñar un buen método de remuestreo bootstrap es **imitar por completo el procedimiento de muestreo en la población original** (incluyendo el estadístico y las características de su distribución muestral).

Como ya se comentó anteriormente, en el bootstrap uniforme se emplea como aproximación la distribución empírica (ver Sección A.1.2):

$$F_n(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}\{X_i \leq x\}.$$

Es decir, $\hat{F} = F_n$, y por tanto $R^* = R(\mathbf{X}^*, F_n)$. En raras ocasiones (e.g. Sección 1.3 de Cao y Fernández-Casal, 2020) es posible calcular exactamente la distribución bootstrap de R^*. Normalmente se aproxima esa distribución mediante Monte Carlo, generando una gran cantidad, B, de réplicas de R^*. En el caso del bootstrap uniforme, el algoritmo es:

1. Para cada $i = 1, \dots, n$ generar X_i^* a partir de F_n, es decir $P^*\left(X_i^* = X_j\right) = \frac{1}{n}$, $j = 1, \dots, n$
2. Obtener $\mathbf{X}^* = (X_1^*, \dots, X_n^*)$
3. Calcular $R^* = R(\mathbf{X}^*, F_n)$
4. Repetir B veces los pasos 1-3 para obtener las réplicas bootstrap $R^{*(1)}, \dots, R^{*(B)}$
5. Utilizar esas réplicas bootstrap para aproximar la distribución en el muestreo de R.

Para la elección del número de réplicas Monte Carlo B se aplicarían las mismas recomendaciones de la Sección 3.2 para el caso general de una aproximación por simulación.

Como ya se mostró anteriormente, el paso 1 se puede llevar a cabo simulando una distribución uniforme discreta mediante el método de la transformación cuantil (Sección 5.5):

1. Para cada $i = 1, \dots, n$ arrojar $U_i \sim \mathcal{U}(0,1)$ y hacer $X_i^* = X_{\lfloor nU_i \rfloor + 1}$

Aunque en `R` es recomendable[2] emplear la función `sample` para generar muestras aleatorias con reemplazamiento del conjunto de datos original:

```
muestra_boot <- sample(muestra, replace = TRUE)
```

Ejemplo 8.2 (Inferencia sobre la media con varianza desconocida)

Como ejemplo consideramos la muestra de tiempos de vida de microorganismos `lifetimes` del paquete `simres` [Figura 8.6]:

```
library(simres)
muestra <- lifetimes
summary(muestra)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##  0.1430  0.2650  0.6110  0.8053  1.1200  2.0800
```

```
hist(muestra)
rug(muestra)
```

Supongamos que queremos obtener una estimación por intervalo de confianza de su vida media a partir de los 15 valores observados mediante bootstrap uniforme considerando el estadístico

$$R = R(\mathbf{X}, F) = \sqrt{n}\frac{\bar{X} - \mu}{S_{n-1}},$$

(en la Sección Sección **??** se tratará con más detalle la construcción de intervalos de confianza).

[2] De esta forma se evitan posibles problemas numéricos al emplear el método de la transformación cuantil cuando n es extremadamente grande (e.g. https://stat.ethz.ch/pipermail/r-devel/2018-September/076817.html).

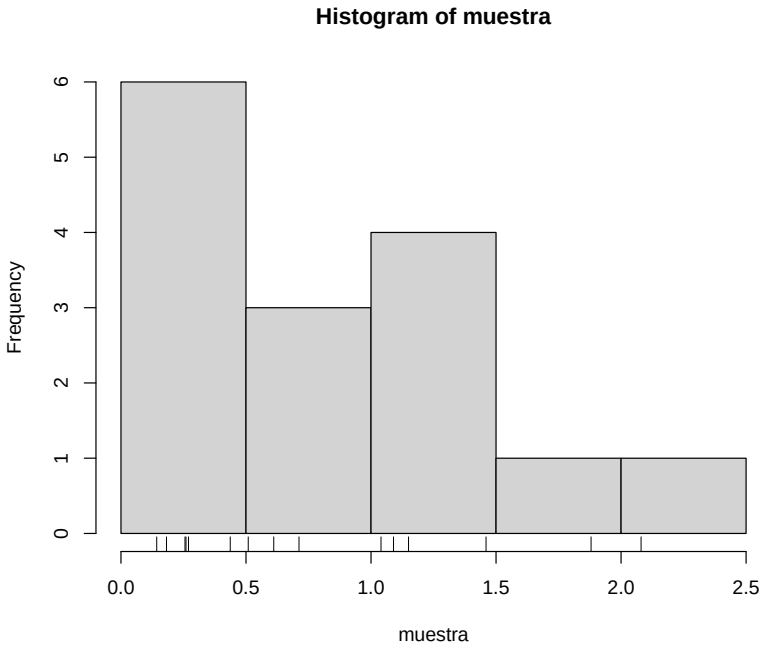


Figura 8.6: Distribución del tiempo de vida de una muestra de microorganismos.

En el bootstrap uniforme se emplea $\hat{F} = F_n$, con lo cual el análogo bootstrap del estadístico R será

$$R^* = R\left(\mathbf{X}^*, F_n\right) = \sqrt{n}\frac{\bar{X}^* - \bar{X}}{S^*_{n-1}},$$

siendo

$$\bar{X}^* = \frac{1}{n}\sum_{i=1}^{n} X_i^*,$$

$$S^{*2}_{n-1} = \frac{1}{n-1}\sum_{i=1}^{n}\left(X_i^* - \bar{X}^*\right)^2.$$

El algoritmo bootstrap (aproximado por Monte Carlo) procedería así:

1. Para cada $i = 1, \dots, n$ arrojar $U_i \sim \mathcal{U}(0,1)$ y hacer $X_i^* = X_{\lfloor nU_i \rfloor + 1}$
2. Obtener $\bar{X}^*$ y S^{*2}_{n-1}
3. Calcular $R^* = \sqrt{n}\frac{\bar{X}^* - \bar{X}}{S^*_{n-1}}$
4. Repetir B veces los pasos 1-3 para obtener las réplicas bootstrap $R^{*(1)}, \dots, R^{*(B)}$
5. Aproximar la distribución en el muestreo de R mediante la distribución empírica de $R^{*(1)}, \dots, R^{*(B)}$

Por ejemplo, podríamos emplear el siguiente código:

```
n <- length(muestra)
alfa <- 0.05
# Estimaciones muestrales
x_barra <- mean(muestra)
cuasi_dt <- sd(muestra)
# Remuestreo
set.seed(1)
```

```
B <- 1000
estadistico_boot <- numeric(B)
for (k in 1:B) {
  remuestra <- sample(muestra, n, replace = TRUE)
  x_barra_boot <- mean(remuestra)
  cuasi_dt_boot <- sd(remuestra)
  estadistico_boot[k] <- sqrt(n) * (x_barra_boot - x_barra)/cuasi_dt_boot
}
```

Las características de interés de la distribución en el muestreo de R se aproximan por las correspondientes de la distribución bootstrap de R^*. En este caso nos interesa aproximar los puntos críticos $x_{\alpha/2}$ y $x_{1-\alpha/2}$, tales que:

$$P\left(x_{\alpha/2} < R < x_{1-\alpha/2}\right) = 1 - \alpha.$$

Para lo que podemos emplear los cuantiles muestrales[3]:

```
pto_crit <- quantile(estadistico_boot, c(alfa/2, 1 - alfa/2))
pto_crit
```

```
##      2.5%     97.5%
## -3.002197  1.877302
```

A partir de los cuales obtenemos la correspondiente estimación por IC boostrap (ver Sección **??**):

$$\hat{IC}_{1-\alpha}^{boot}(\mu) = \left(\overline{X} - x_{1-\alpha/2}\frac{S_{n-1}}{\sqrt{n}},\ \overline{X} - x_{\alpha/2}\frac{S_{n-1}}{\sqrt{n}}\right).$$

```
ic_inf_boot <- x_barra - pto_crit[2] * cuasi_dt/sqrt(n)
ic_sup_boot <- x_barra - pto_crit[1] * cuasi_dt/sqrt(n)
IC_boot <- c(ic_inf_boot, ic_sup_boot)
names(IC_boot) <- paste0(100*c(alfa/2, 1-alfa/2), "%")
IC_boot
```

```
##      2.5%     97.5%
## 0.5030131 1.2888063
```

Este procedimiento para la construcción de intervalos de confianza se denomina *método percentil-t* y se tratará en la Sección **??**.

Como ejemplo adicional podemos comparar la aproximación de la distribución bootstrap del estadístico con la aproximación t_{n-1} basada en normalidad.

```
hist(estadistico_boot, freq = FALSE, ylim = c(0, 0.4))
abline(v = pto_crit, lty = 2)
curve(dt(x, n - 1), add = TRUE, col = "blue")
pto_crit_t <- qt(1 - alfa/2, n - 1)
abline(v = c(-pto_crit_t, pto_crit_t), col = "blue", lty = 2)
```

En este caso la distribución bootstrap del estadístico es más asimétrica, por lo que el intervalo de confianza no está centrado en la media, al contrario que el obtenido con la aproximación tradicional. Por ejemplo, podemos obtener la estimación basada en normalidad mediante la función `t.test()`:

```
t.test(muestra)$conf.int
```

```
## [1] 0.4599374 1.1507292
```

[3]Se podrían considerar distintos estimadores del cuantil x_α (ver p.e. la ayuda de la función `quantile()`). Si empleamos directamente la distribución empírica, el cuantil se correspondería con la observación ordenada en la posición $B\alpha$ (se suele hacer una interpolación lineal si este valor no es entero), lo que equivale a emplear la función `quantile()` de `R` con el parámetro `type = 1`. Esta función considera por defecto la posición $1 + (B-1)\alpha$ (`type = 7`). En el libro de Davison y Hinkley (1997), y en el paquete `boot`, se emplea $(B+1)\alpha$ (equivalente a `type = 6`; lo que justifica que consideren habitualmente 99, 199 ó 999 réplicas bootstrap).

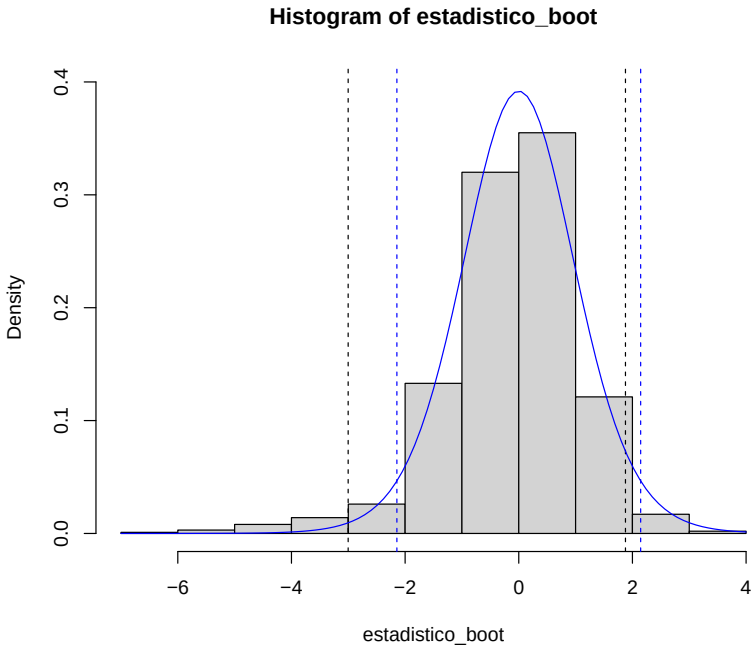


Figura 8.7: Aproximación de la distribución de la media muestral studentizada mediante bootstrap uniforme.

```
## attr(,"conf.level")
## [1] 0.95
```

En el caso multidimensional, cuando trabajamos con un conjunto de datos con múltiples variables, podríamos emplear un procedimiento análogo, a partir de remuestras del vector de índices. Por ejemplo:

```
data(iris)
str(iris)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 4.9 4.7 4.6 5 5.4 4.6 5 4.4 4.9 ...
##  $ Sepal.Width : num  3.5 3 3.2 3.1 3.6 3.9 3.4 3.4 2.9 3.1 ...
##  $ Petal.Length: num  1.4 1.4 1.3 1.5 1.4 1.7 1.4 1.5 1.4 1.5 ...
##  $ Petal.Width : num  0.2 0.2 0.2 0.2 0.2 0.4 0.3 0.2 0.2 0.1 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 1 1 1 1 1 1 1 1 1 ...
```

```
n <- nrow(iris)
# i_boot <- floor(n*runif(n)) + 1
# i_boot <- sample.int(n, replace = TRUE)
i_boot <- sample(n, replace = TRUE)
data_boot <- iris[i_boot, ]
str(data_boot)
```

```
## 'data.frame':    150 obs. of  5 variables:
##  $ Sepal.Length: num  5.1 5.6 6.2 4.8 5.5 6.2 5.5 5.6 5 6.5 ...
##  $ Sepal.Width : num  3.8 2.5 2.9 3.1 2.3 2.9 2.6 2.8 3.6 3 ...
##  $ Petal.Length: num  1.9 3.9 4.3 1.6 4 4.3 4.4 4.9 1.4 5.8 ...
##  $ Petal.Width : num  0.4 1.1 1.3 0.2 1.3 1.3 1.2 2 0.2 2.2 ...
##  $ Species     : Factor w/ 3 levels "setosa","versicolor",..: 1 2 2 1 2 2 2 3 1 3 ...
```

Esta forma de proceder es la que emplea por defecto el paquete `boot` que describiremos más adelante

(Sección 8.3.1).

Ejercicio 8.1 (Bootstrap uniforme multidimensional)
Considerando el conjunto de datos `Prestige` del paquete `carData`, supongamos que queremos realizar inferencias sobre el coeficiente de correlación entre `prestige` (puntuación de ocupaciones obtenidas a partir de una encuesta) e `income` (media de ingresos en la ocupación). Para ello podemos considerar el coeficiente de correlación lineal de Pearson:

$$\rho = \frac{Cov\,(X,Y)}{\sigma\,(X)\,\sigma\,(Y)}$$

Su estimador es el coeficiente de correlación muestral:

$$r = \frac{\sum_{i=1}^{n}(x_i-\overline{x})(y_i-\overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i-\overline{y})^2}},$$

que podemos calcular en `R` empleando la función `cor()`:

```
data(Prestige, package = "carData")
# with(Prestige, cor(income, prestige))
cor(Prestige$income, Prestige$prestige)
```

```
## [1] 0.7149057
```

Para realizar inferencias sobre el coeficiente de correlación, como aproximación más simple, se puede considerar que la distribución muestral de r es aproximadamente normal de media ρ y varianza

$$Var(r) \approx \frac{1-\rho^2}{n-2}.$$

Aproximar mediante bootstrap uniforme (multididimensional) la distribución del estadístico $R = r-\rho$, empleando $B = 1000$ réplicas, y compararla con la aproximación normal, considerando

$$\widehat{Var}(r) = \frac{1-r^2}{n-2}.$$

8.3 Herramientas disponibles en R sobre bootstrap

En `R` hay una gran cantidad de paquetes que implementan métodos bootstrap. Por ejemplo, al ejecutar el comando `??bootstrap` (o `help.search('bootstrap')`) se mostrarán las funciones de los paquetes instalados que incluyen este término en su documentación (se puede realizar la búsqueda en todos los paquetes disponibles de `R` a través de https://www.rdocumentation.org).

De entre todos estas herramientas destacan dos librerías como las más empleadas:

- `bootstrap`: contiene las rutinas (bootstrap, cross-validation, jackknife) y los datos del libro "An Introduction to the Bootstrap" de B. Efron y R. Tibshirani, 1993, Chapman and Hall. La librería fue desarrollada originalmente en `S` por Rob Tibshirani y exportada a `R` por Friedrich Leisch. Es útil para desarrollar los ejemplos que se citan en ese libro.
- `boot`: incluye las funciones y conjuntos de datos utilizados en el libro "Bootstrap Methods and Their Applications" de A. C. Davison y D. V. Hinkley, 1997, Cambridge University Press. Esta librería fue desarrollada originalmente en `S` por Angelo J. Canty y posteriormente exportada a `R` (ver Canty, 2002). Este paquete es mucho más completo que el paquete `bootstrap`, forma parte de la distribución estándar de `R` y es el que emplearemos como referencia en este libro (ver Sección 8.3.1).

Por otra parte existen numerosas rutinas (scripts) realizadas en `R` por diversos autores, que están disponibles en Internet (por ejemplo, puede ser interesante realizar una búsqueda en https://rseek.org).

El bootstrap uniforme se puede implementar fácilmente. Por ejemplo, una rutina general para el caso univariante sería la siguiente:

```
#' @param x vector que contiene la muestra.
#' @param B número de réplicas bootstrap.
#' @param statistic función que calcula el estadístico.
boot.strap0 <- function(x, B=1000, statistic=mean){
  ndat <- length(x)
  x.boot <- sample(x, ndat*B, replace=TRUE)
  x.boot <- matrix(x.boot, ncol=B, nrow=ndat)
  stat.boot <- apply(x.boot, 2, statistic)
}
```

Podríamos aplicar esta función a la muestra de tiempos de vida de microorganismos con el siguiente código:

```
fstatistic0 <- function(x){
  mean(x)
}

B <- 1000
set.seed(1)
stat.dat <- fstatistic0(muestra)
stat.boot <- boot.strap0(muestra, B, fstatistic0)

res.boot <- c(stat.dat, mean(stat.boot)-stat.dat, sd(stat.boot))
names(res.boot) <- c("Estadístico", "Sesgo", "Error Std.")
res.boot
```

```
## Estadístico       Sesgo  Error Std.
## 0.805333333 0.003173267 0.154099013
```

La función `boot.strap0()` anterior no es adecuada para el caso multivariante (por ejemplo cuando estamos interesados en regresión). Como se mostró en la Sección 8.1 sería preferible emplear remuestras del vector de índices. Por ejemplo:

```
#' @param datos vector, matriz o data.frame que contiene los datos.
#' @param B número de réplicas bootstrap.
#' @param statistic función con al menos dos parámetros,
#' los datos y el vector de índices de remuestreo,
#' y que devuelve el vector de estadísticos.
#' @param ... parámetros adicionales de la función statistic.
boot.strap <- function(datos, B=1000, statistic, ...) {
  ndat <- NROW(datos)
  i.boot <- sample(ndat, ndat*B, replace=TRUE)
  i.boot <- matrix(i.boot, ncol=B, nrow=ndat)
  stat.boot <- drop(apply(i.boot, 2, function(i) statistic(datos, i, ...)))
}
```

El paquete `boot`, descrito a continuación, emplea una implementación similar.

8.3.1 El paquete `boot`

La función principal de este paquete es la función `boot()` que implementa distintos métodos de remuestreo para datos i.i.d.. En su forma más simple permite realizar bootstrap uniforme (que en la práctica también se denomina habitualmente *bootstrap noparamétrico*):

```
boot(data, statistic, R)
```

donde `data` es un vector, matriz o `data.frame` que contiene los datos, `R` es el número de réplicas bootstrap, y `statistic` es una función con al menos dos parámetros (con las opciones por defecto), los datos y el vector de índices de remuestreo, y que devuelve el vector de estadísticos.

Por ejemplo, para hacer inferencia sobre la mediana del tiempo de vida de microorganismos, podríamos emplear el siguiente código:

```
library(boot)
muestra <- lifetimes

statistic <- function(data, i){
  # remuestra <- data[i]; mean(remuestra)
  mean(data[i])
}

set.seed(1)
res.boot <- boot(muestra, statistic, R = 1000)
```

El resultado que devuelve esta función es un objeto de clase `boot`, una lista con los siguientes componentes:

```
names(res.boot)
```

```
##  [1] "t0"        "t"         "R"         "data"      "seed"      "statistic"
##  [7] "sim"       "call"      "stype"     "strata"    "weights"
```

Además de los parámetros de entrada (incluyendo los valores por defecto), contiene tres componentes adicionales:

- `t0`: el valor observado del estadístico (su evaluación en los datos originales).
- `t`: la matriz de réplicas bootstrap del estadístico (cada fila se corresponde con una remuestra).
- `seed`: el valor inicial de la semilla (`.Random.seed`) empleada para la generación de las réplicas.

Este tipo de objetos dispone de dos métodos principales: el método `print()` que muestra un resumen de los resultados (incluyendo aproximaciones bootstrap del sesgo y del error estándar de los estadísticos; ver Sección **??**):

```
res.boot
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = muestra, statistic = statistic, R = 1000)
##
##
## Bootstrap Statistics :
##      original      bias    std. error
## t1* 0.8053333 0.003173267   0.1583306
```

y el método `plot()` que genera gráficas básicas de diagnosis de los resultados (correspondientes al estadístico determinado por el parámetro `index`, por defecto `= 1`): [Figura 8.8]

```
plot(res.boot)
```

Es recomendable examinar la distribución bootstrap del estimador (o estadístico) para detectar posibles problemas. Como en este caso puede ocurrir que el estadístico bootstrap tome pocos valores distintos, lo que indicaría que el número de réplicas bootstrap es insuficiente o que hay algún problema con método de remuestreo empleado. Se darán más detalles sobre los posibles problemas del bootstrap uniforme en la Sección 9.1.

Ejercicio 8.2

Repetir el ejemplo anterior considerando simultáneamente la media truncada al 10% y la mediana como estimadores de la posición central de los datos. Estudiar si hay algún problema con su distribu-

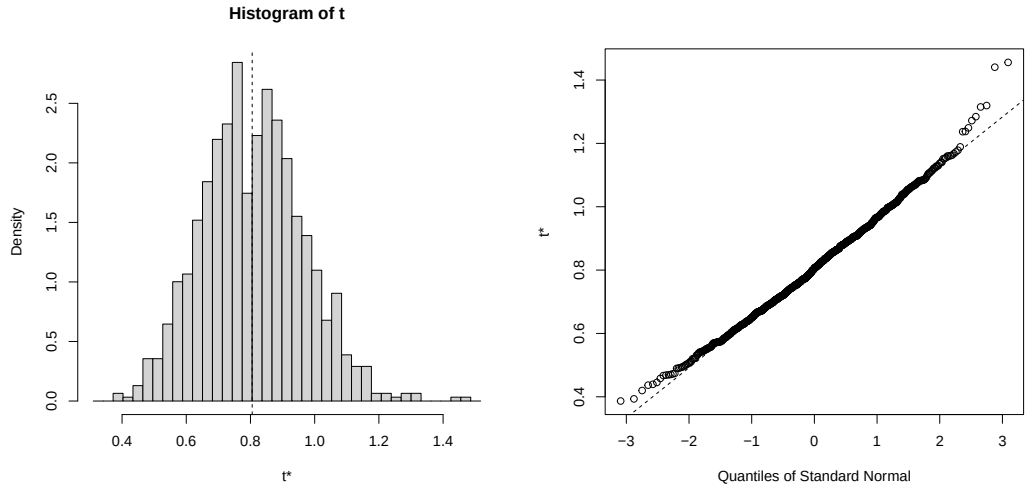


Figura 8.8: Gráficos de diagnóstico de los resultados bootstrap para la media muestral de los tiempos de vida de microorganismos.

ción bootstrap (NOTA: al igual que en el caso anterior, las distribuciones objetivo serían continuas, asumiendo que la distribución del tiempo de vida es continua).

Además de estos métodos, las principales funciones de interés serían:

- `jack.after.boot()`: genera un gráfico para diagnósticar la inluencia de las observaciones individuales en los resultados bootstrap (se representan los cuantiles frente a las diferencias en el estadístico al eliminar una observación; este gráfico también se puede obtener estableciendo `jack = TRUE` en `plot.boot()`).
- `boot.array()`: genera la matriz de índices a partir de la que se obtuvieron las remuestras (permite reconstruir las remuestras bootstrap).
- `boot.ci()`: construye distintos tipos de intervalos de confianza (se tratarán en el Sección 10.2) dependiendo del parámetro `type`:
  - `"norm"`: utiliza la distribución asintótica normal considerando las aproximaciones bootstrap del sesgo y de la varianza.
  - `"basic"`: emplea el estadístico $R = \widehat{\theta} - \theta$ para la construcción del intervalo de confianza.
  - `"stud"`: calcula el intervalo a partir del estadístico studentizado $R = \left(\widehat{\theta} - \theta\right) / \sqrt{\widehat{Var}(\widehat{\theta})}$.
  - `"perc"`: utiliza directamente la distribución bootstrap del estadístico ($R = \widehat{\theta}$).
  - `"bca"`: emplea el método BCa ("bias-corrected and accelerated") propuesto por Efron (1987) (ver Sección 5.3.2 de Davison y Hinkley, 1997).
  - `"all"`: calcula los cinco tipos de intervalos anteriores.

Como ya se comentó, la función `boot()` admite estadísticos multivariantes (haciendo que la función `statistic` devuelva un vector en lugar de un escalar), pero por defecto las funciones anteriores consideran el primer componente como el estadístico principal. Para obtener resultados de otros componentes del vector de estadísticos habrá que establecer el parámetro `index` igual al índice deseado. Además, en algunos casos (por ejemplo para la obtención de intevalos de confianza estudentizados con la función `boot.ci()`) se supone, por defecto, que el segundo componente del vector de estadísticos contiene estimaciones de la varianza del estadístico para cada réplica boostrap.

Ejemplo 8.3 (Inferencia sobre la media con varianza desconocida continuación)

Continuando con el Ejemplo 8.2 de inferencia sobre la media con varianza desconocida. Para obtener la

estimación por intervalo de confianza del tiempo de vida medio de los microorganismos con el paquete `boot`, podríamos emplear el siguiente código:

```
library(boot)
muestra <- lifetimes

statistic <- function(data, i){
  remuestra <- data[i]
  c(mean(remuestra), var(remuestra)/length(remuestra))
}

set.seed(1)
res.boot <- boot(muestra, statistic, R = 1000)
res.boot
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = muestra, statistic = statistic, R = 1000)
##
##
## Bootstrap Statistics :
##      original       bias    std. error
## t1* 0.8053333  0.003173267 0.158330646
## t2* 0.0259338 -0.002155755 0.007594682
```

```
boot.ci(res.boot)
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = res.boot)
##
## Intervals :
## Level      Normal              Basic              Studentized
## 95%   ( 0.4918,  1.1125 )   ( 0.4825,  1.0980 )   ( 0.4715,  1.2320 )
##
## Level     Percentile            BCa
## 95%   ( 0.5127,  1.1282 )   ( 0.5384,  1.1543 )
## Calculations and Intervals on Original Scale
```

El intervalo marcado como `Studentized` se obtuvo empleando el mismo estadístico del Ejemplo 8.2.

8.3.2 Extensiones del bootstrap uniforme con el paquete `boot`

Estableciendo parámetros adicionales de la función `boot` se pueden llevar a cabo modificaciones del bootstrap uniforme (Capítulo 9). Algunos de estos parámetros son los siguientes:

- `strata`: permite realizar remuestreo estratificado estableciendo este parámetro como un vector numérico o factor que defina los grupos.
- `sim = c("ordinary" , "parametric", "balanced", "permutation", "antithetic")`: permite establecer distintos tipos de remuestreo. Por defecto es igual a `"ordinary"` que se corresponde con el bootstrap uniforme, descrito anteriormente. Entre el resto de opciones destacaríamos `sim = "permutation"`, que permite realizar contrastes de permutaciones (remuestreo sin reemplazamiento), y `sim = "parametric"`, que permite realizar bootstrap paramétrico (Sección 9.2).

En este último caso también habrá que establecer los parámetros `ran.gen` y `mle`, y la función `statistics` no empleará el segundo parámetro de índices.

- `ran.gen`: función que genera los datos. El primer argumento será el conjunto de datos original y el segundo un vector de parámetros adicionales (normalmente los valores de los parámetros de la distribución).
- `mle`: parámetros de la distribución (típicamente estimados por máxima verosimilitud) o parámetros adicionales para `ran.gen` ó `statistics`.

Además hay otros parámetros para el procesamiento en paralelo: `parallel = c("no", "multicore", "snow")`, `ncpus`, `cl`. En el Apéndice C se incluye una pequeña introducción al procesamiento en paralelo y se muestran algunos ejemplos sobre el uso de estos parámetros. También se puede consultar la ayuda de la función `boot()` (`?boot`).

El paquete `boot` también incluye otras funciones que implementan métodos boostrap para otros tipos de datos, como la función `tsboot()` para series de tiempo (Sección **??**) o la función `censboot()` para datos censurados (ver p.e. Cao y Fernández-Casal, 2021, Capítulo 8).

Finalmente destacar que hay numerosas extensiones implementadas en otros paquetes utilizando el paquete `boot` (ver *Reverse dependencies* en la web de CRAN). Por ejemplo en la Sección 9.4 se ilustrará el uso de la función `Boot()` del paquete `car` para hacer inferencia sobre modelos de regresión.

8.3.3 Ejemplo: Bootstrap uniforme multidimensional

Como ya se mostró en las Secciones 8.2 y 8.3 podemos implementar el bootstrap uniforme en el caso multidimensional (denominado también *remuestreo de casos* o *bootstrap de las observaciones*) de modo análogo al unidimensional.

Como ejemplo realizamos el Ejercicio 8.1 empleando el paquete `boot` para estudiar la correlación lineal entre `prestige` e `income` del conjunto de datos `Prestige`:

```
library(boot)

statistic <- function(data, i){
  remuestra <- data[i, ]
  cor(remuestra$income, remuestra$prestige)
}

set.seed(1)
B <- 1000
res.boot <- boot(Prestige, statistic, R = B)
res.boot
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Prestige, statistic = statistic, R = B)
##
##
## Bootstrap Statistics :
##      original      bias    std. error
## t1* 0.7149057 0.006306905  0.04406473
```

```
plot(res.boot)
```

En este caso podemos observar que la distribución bootstrap del estimador es asimétrica, por lo que asumir que su distribución es normal podría no ser adecuado (por ejemplo para la construcción de intervalos de confianza, que se tratarán en la Sección 10.2.6).

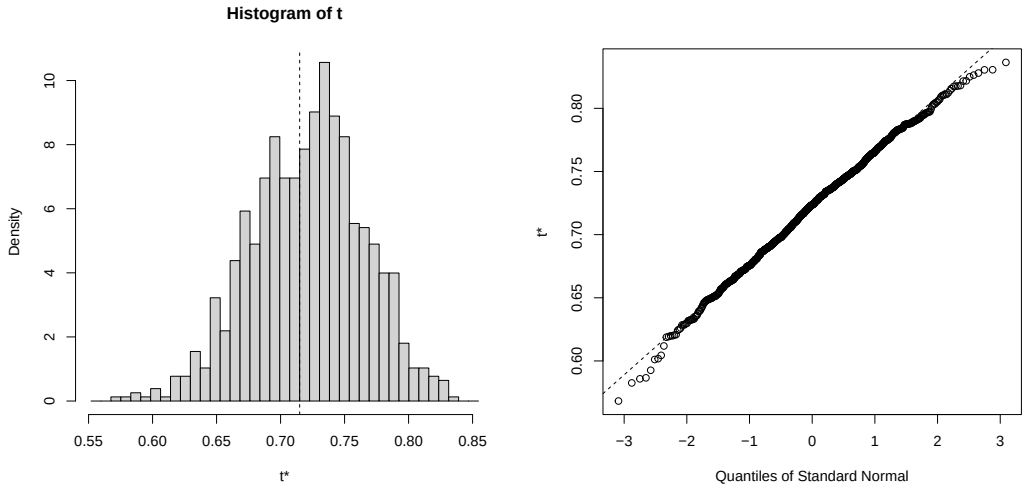


Figura 8.9: Gráficos de diagnóstico para la distribución bootstrap del coeficiente de correlación.

Como comentario final, nótese que en principio el paquete boot está diseñado para obtener réplicas bootstrap de un estimador, por lo que si lo que nos interesa es emplear otro estadístico habría que construirlo a partir de ellas (como hacen otras funciones secundarias como `boot.ci()`). Por ejemplo, si queremos emplear el estadístico $R = \hat{\theta} - \theta$ (bootstrap percentil básico o natural), podemos obtener la correspondiente distribución bootstrap (aproximada por Monte Carlo) con el siguiente código:

```
estadistico_boot <- res.boot$t - res.boot$t0
hist(estadistico_boot)
```

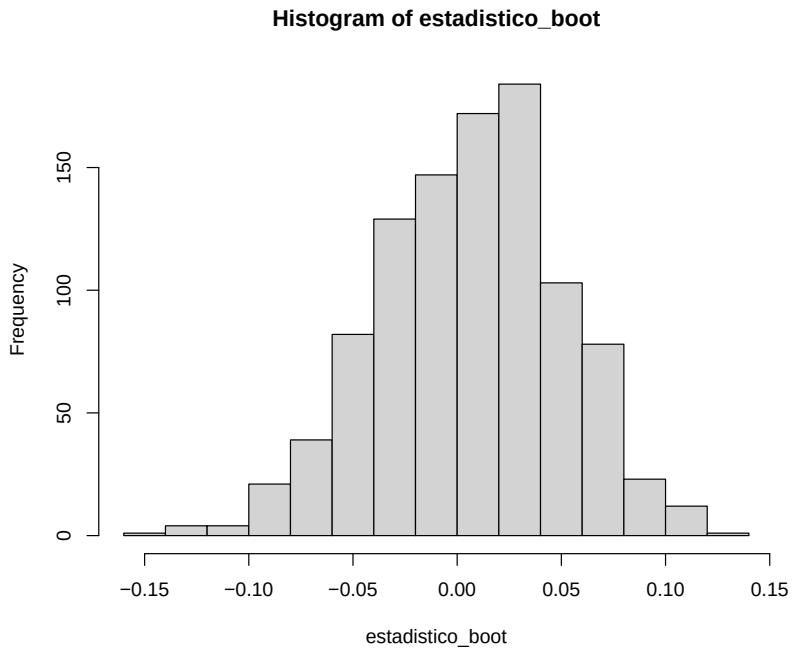


Figura 8.10: Distribución bootstrap del estadístico percentil básico para el coeficiente de correlación.

Podemos emplear la distribución empírica del estadístico bootstrap $R^* = \hat{\theta}^* - \hat{\theta}$ para aproximar la característica de interés de la distribución en el muestreo de $R = \hat{\theta} - \theta$. Por ejemplo, para aproximar

$\psi(u) = P\left(R \leq u\right)$ podríamos emplear la frecuencia relativa:

$$\hat{\psi}_B(u) = \frac{1}{B}\sum_{i=1}^{B} \mathbf{1}\left\{R^{*(i)} \leq u\right\}.$$

```
u <- 0
sum(estadistico_boot <= u)/B
```

```
## [1] 0.427
```

```
# Equivalentemente:
mean(estadistico_boot <= u)
```

```
## [1] 0.427
```

Capítulo 9

Extensiones del bootstrap uniforme

Cuando no se dispone de ninguna información adicional sobre la distribución poblacional es razonable emplear bootstrap uniforme (naïve o no paramétrico), ya que la distribución empírica es el estimador máximo verosímil no paramétrico de la función de distribución poblacional. En general (ver p.e. Cao y Fernández-Casal, 2021, Sección 3.6), el bootstrap uniforme funcionará bien cuando el estadístico sea una función suave de la muestra[1] (suponiendo además que la aproximación Monte Carlo converge, es decir, que las colas de la distribución poblacional no son muy pesadas; ver Sección 3.1). Si el estadístico depende de la muestra de una manera poco suave o inestable, como es el caso de los cuantiles muestrales, el bootstrap uniforme puede no funcionar muy bien (Sección 9.1), y puede ser recomendable emplear otro método de remuestreo.

Además, cuando en el contexto en el que nos encontremos conozcamos alguna propiedad adicional de la distribución poblacional (por ejemplo que es continua o simétrica), debería tenerse en cuenta en en el método de remuestreo, ya que habría que tratar de imitar todas las características de la distribución poblacional. Esto da lugar a modificaciones del bootstrap uniforme, algunas de las cuales se muestran en secciones siguientes.

9.1 Deficiencias del bootstrap uniforme

Como ya se comentó en la introducción, si el estadístico es función de los cuantiles muestrales el bootstrap uniforme puede no funcionar bien (por ejemplo en el caso de la mediana; ver Ejercicio 8.2). Esto es principalmente debido a que el conjunto de posibles valores de las réplicas bootstrap del estadístico puede ser muy pequeño (si el tamaño muestral es pequeño).

Un ejemplo en el que el bootstrap uniforme falla por completo es el caso del máximo de una distribución uniforme. Supongamos que la distribución poblacional, F, es la $\mathcal{U}(0,\theta)$ y que estamos interesados en hacer inferencia sobre del parámetro θ empleando su estimador máximo verosímil $\widehat{\theta} = X_{(n)}$, siendo $\mathbf{X} = (X_1, X_2, \dots, X_n)$ la muestra observada. Para realizar dicha inferencia nos interesaría aproximar la distribución de $R(\mathbf{X}, F) = \widehat{\theta} - \theta$.

La función de distribución en el muestreo, $G(x)$, de $\widehat{\theta}$ puede calcularse de forma sencilla:

$$\begin{aligned} G(x) &= P\left(X_{(n)} \leq x\right) = P\left(X_i \leq x\,; 1 \leq i \leq n\right) \\ &= \prod_{i=1}^{n} P\left(X_i \leq x\right) = F(x)^n = \left(\frac{x}{\theta}\right)^n, \text{ si } x \in [0,\theta] \end{aligned}$$

con lo cual su función de densidad viene dada por

$$g(x) = \frac{n}{\theta}\left(\frac{x}{\theta}\right)^{n-1}, \text{ si } x \in [0,\theta].$$

[1] Por ejemplo si es una función diferenciable de los momentos muestrales.

Lo que confirma que $\widehat{\theta}$ es un estimador sesgado de θ, puesto que se tiene que $\widehat{\theta} \leq \theta$ con probabilidad 1. A partir de esta densidad podemos calcular fácilmente el sesgo del estimador:

$$Sesgo\left(\widehat{\theta}\right) = E\left(\widehat{\theta}\right) - \theta = -\frac{\theta}{n+1}.$$

Si deseamos aproximar mediante bootstrap la distribución en el muestreo de $\widehat{\theta}$ (o la de R) y utilizamos un bootstrap uniforme (naïve), la versión bootstrap del estimador resulta ser $\widehat{\theta}^* = X^*_{(n)}$, siendo $\mathbf{X}^* = (X^*_1, X^*_2, \ldots, X^*_n)$ una remuestra bootstrap obtenida a partir de la distribución empírica F_n. La distribución en el remuestreo de $\widehat{\theta}^*$ resulta un poco más complicada pues es discreta y sólo puede tomar cualquiera de los valores de la muestra.

Suponiendo que no hay empates en las observaciones de la muestra, es fácil darse cuenta de que

$$P^*\left(\widehat{\theta}^* \leq X_{(j)}\right) = P^*\left(X^*_{(n)} \leq X_{(j)}\right) = P^*\left(X^*_i \leq X_{(j)}\,; 1 \leq i \leq n\right) = \left(\frac{j}{n}\right)^n$$

y, por tanto, su masa de probabilidad viene dada por

$$P^*\left(\widehat{\theta}^* = X_{(j)}\right) = \left(\frac{j}{n}\right)^n - \left(\frac{j-1}{n}\right)^n, \; j = 1, \ldots, n.$$

En particular,

$$P^*\left(\widehat{\theta}^* = X_{(n)}\right) = 1 - \left(1 - \frac{1}{n}\right)^n \to 1 - \frac{1}{e} \simeq 0.6321,$$

con lo cual la distribución en remuestreo de $R^* = R\left(\mathbf{X}^*, F_n\right) = \widehat{\theta}^* - X_{(n)}$ tiene un átomo de probabilidad en el valor 0 cuya probabilidad tiende a $1 - \frac{1}{e}$ cuando el tamaño muestral tiende a infinito, es decir

$$\lim_{n\to\infty} P^*\left(R^* = 0\right) = 1 - \frac{1}{e},$$

algo que no ocurre con la distribución en el muestreo de R, que es continua con densidad:

$$g_R\left(x\right) = \frac{n}{\theta}\left(\frac{x+\theta}{\theta}\right)^{n-1}, \text{ si } x \in [-\theta, 0].$$

De esta forma vemos que el bootstrap uniforme (no paramétrico) es inconsistente.

Ejemplo 9.1 (Inferencia sobre el máximo de una distribución uniforme)
El siguiente código implementa el método bootstrap uniforme (también llamado naïve) para aproximar la distribución del estadístico $R = \widehat{\theta} - \theta$, para una muestra de tamaño $n = 50$, proveniente de una población con distribución $\mathcal{U}\left(0, 1\right)$ [Figura 9.1]:

```
theta <- 1
n <- 50
set.seed(1)
muestra <- runif(50) * theta
theta_est <- max(muestra)
# Remuestreo
B <- 2000
maximo <- numeric(B)
estadistico <- numeric(B)
for (k in 1:B) {
    remuestra <- sample(muestra, n, replace = TRUE)
    maximo[k] <- max(remuestra)
    estadistico[k] <- maximo[k] - theta_est
```

```
}
# Distribución estadístico
xlim <- c(-theta/2, 0) # c(-theta, 0)
hist(estadistico, freq = FALSE, main = "", breaks = "FD",
     border = "darkgray", xlim = xlim)
lines(density(estadistico))
rug(estadistico, col = "darkgray")
curve(n/theta * ((x + theta)/theta)^(n - 1), col = "blue", lty = 2, lwd = 2, add = TRUE)
```

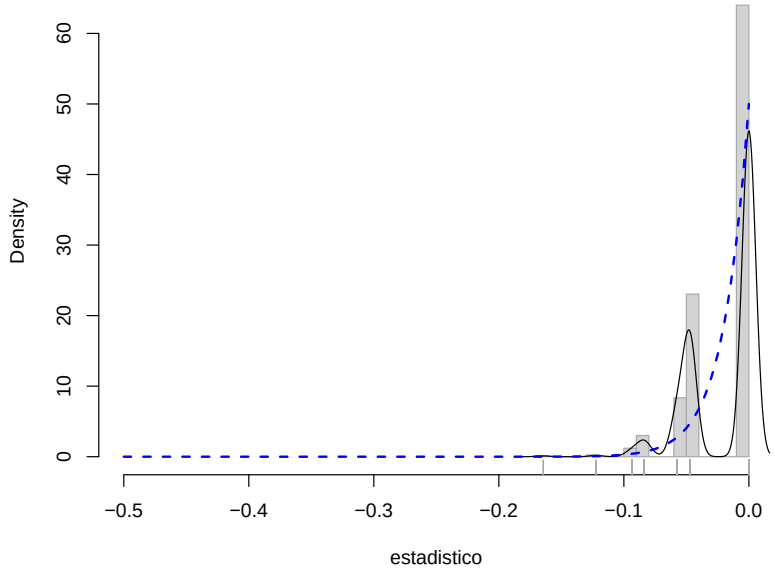


Figura 9.1: Distribución de las réplicas bootstrap (uniforme) del estadístico y distribución poblacional.

Este problema podría evitarse empleando el bootstrap paramétrico descrito a continuación (ver Ejemplo 9.3).

9.2 Bootstrap paramétrico

Supongamos que sabemos que la función de distribución poblacional pertenece a cierta familia paramétrica. Es decir $F = F_\theta$ para algún vector d-dimensional $\theta \in \Theta$. En ese caso parece lógico estimar θ a partir de la muestra (denotemos $\widehat{\theta}$ un estimador de θ, por ejemplo el de máxima verosimilitud) y obtener remuestras de $F_{\widehat{\theta}}$ no de F_n. Entonces, el bootstrap uniforme se modifica de la siguiente forma, dando lugar al llamado bootstrap paramétrico:

1. Dada la muestra $\mathbf{X} = (X_1, \dots, X_n)$, calcular $\widehat{\theta}$
2. Para cada $i = 1, \dots, n$ generar X_i^* a partir de $F_{\widehat{\theta}}$
3. Obtener $\mathbf{X}^* = (X_1^*, \dots, X_n^*)$
4. Calcular $R^* = R\left(\mathbf{X}^*, F_{\widehat{\theta}}\right)$
5. Repetir B veces los pasos 2-4 para obtener las réplicas bootstrap $R^{*(1)}$, ..., $R^{*(B)}$
6. Utilizar esas réplicas bootstrap para aproximar la distribución en el muestreo de R

En el paso 2, debemos poder simular valores de la distribución $F_{\widehat{\theta}}$. Como ya hemos visto en la Sección 1.3, en R (y en la mayoría de lenguajes de programación y software estadístico) se dispone de rutinas

que permiten simular la mayoría de las distribuciones paramétricas habituales. En otros casos habrá que emplear métodos generales, preferiblemente el método de inversión descrito en el Capítulo 4.

Una de las principales aplicaciones del bootstrap paramétrico es el contraste de hipótesis que se tratará en la Sección 10.3.1.

Ejemplo 9.2 (Inferencia sobre la media, continuación)

Continuando con el Ejemplo 8.2 del tiempo de vida de microorganismos, podríamos pensar en emplear bootstrap paramétrico para calcular un intervalo de confianza para la media poblacional.

La valided de los resultados dependerá en gran medida de que el modelo paramétrico sea adecuado para describir la variabilidad de los datos. En este caso parece razonable asumir una distribución exponencial (no lo es que el modelo admita tiempos de vida negativos, como ocurriría al asumir normalidad) [Figura 9.2]:

```
muestra <- simres::lifetimes
# Distribución bootstrap uniforme
# plot(ecdf(muestra), xlim = c(-.5, 3), ylab = "F(x)")
curve(ecdf(muestra)(x), xlim = c(-.5, 3), ylab = "F(x)", type = "s")
# Distribución bootstrap paramétrico normal
curve(pnorm(x, mean(muestra), sd(muestra)), lty = 2, add = TRUE)
# Distribución bootstrap paramétrico exponencial
curve(pexp(x, 1/mean(muestra)), lty = 3, add = TRUE)
legend("bottomright", legend = c("Empírica", "Aprox. normal", "Aprox. exponencial"), lty = 1:3)
```

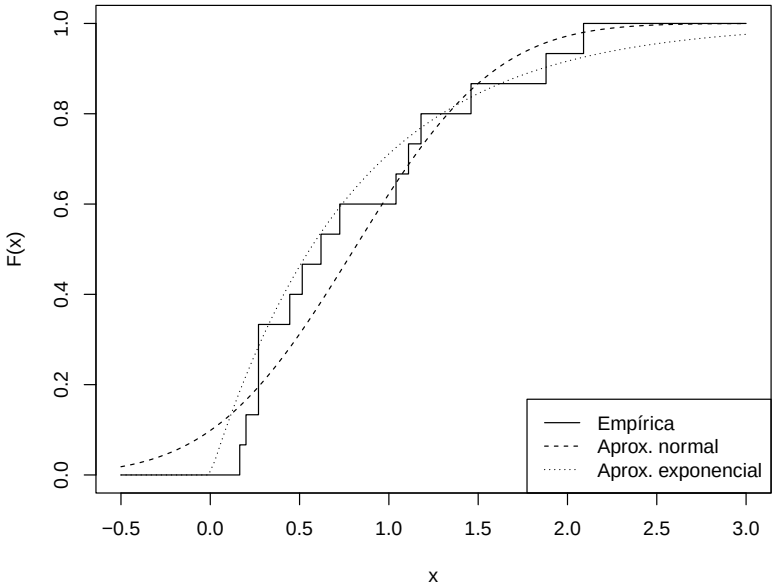


Figura 9.2: Distribución empírica de la muestra de tiempos de vida de microorganismos y aproximaciones paramétricas.

Podemos modificar fácilmente el código mostrado en el Ejemplo 8.2, de forma que se emplee bootstrap paramétrico (exponencial):

```
n <- length(muestra)
alfa <- 0.05
# Estimaciones muestrales
x_barra <- mean(muestra)
```

```
cuasi_dt <- sd(muestra)
# Remuestreo
set.seed(1)
B <- 1000
estadistico_boot <- numeric(B)
for (k in 1:B) {
  # remuestra <- sample(muestra, n, replace = TRUE)
  remuestra <- rexp(n, 1/x_barra)
  x_barra_boot <- mean(remuestra)
  cuasi_dt_boot <- sd(remuestra)
  estadistico_boot[k] <- sqrt(n) * (x_barra_boot - x_barra)/cuasi_dt_boot
}
# Aproximación Monte Carlo de los ptos críticos:
pto_crit <- quantile(estadistico_boot, c(alfa/2, 1 - alfa/2))
# Construcción del IC
ic_inf_boot <- x_barra - pto_crit[2] * cuasi_dt/sqrt(n)
ic_sup_boot <- x_barra - pto_crit[1] * cuasi_dt/sqrt(n)
IC_boot <- c(ic_inf_boot, ic_sup_boot)
names(IC_boot) <- paste0(100*c(alfa/2, 1-alfa/2), "%")
IC_boot
```

```
##      2.5%     97.5%
## 0.5319787 1.3961382
```

Para emplear el paquete `boot`, como se comentó en la Sección 8.3.1, habría que establecer en la llamada a la función `boot()` los argumentos: `sim = "parametric"`, `mle` igual a los parámetros necesarios para la simulación y `ran.gen = function(data, mle)`, una función de los datos originales y de los parámetros que devuelve los datos generados. En este caso además, la función `statistic` no necesita el vector de índices como segundo parámetro. Por ejemplo, para calcular el intervalo de confianza para la media del tiempo de vida de los microorganismos, podríamos utilizar el siguiente código:

```
library(boot)
ran.gen.exp <- function(data, mle) {
    # Función para generar muestras aleatorias exponenciales
    # mle contendrá la media de los datos originales
    out <- rexp(length(data), 1/mle)
    out
}

statistic <- function(data){
    c(mean(data), var(data)/length(data))
}

set.seed(1)
res.boot <- boot(muestra, statistic, R = B, sim = "parametric",
                 ran.gen = ran.gen.exp, mle = mean(muestra))

boot.ci(res.boot, type = "stud")
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = res.boot, type = "stud")
##
## Intervals :
## Level     Studentized
```

```
## 95%   ( 0.5308,  1.4061 )
## Calculations and Intervals on Original Scale
```

Ejemplo 9.3 (Inferencia sobre el máximo de una distribución uniforme, continuación)
En el Ejemplo 9.1 la distribución poblacional era uniforme. Si se dispusiese de esa información, lo natural sería utilizar un bootstrap paramétrico, consistente en obtener las remuestras bootstrap a partir de una distribución uniforme con parámetro estimado:

$$\mathbf{X}^* = (X_1^*, X_2^*, \dots, X_n^*), \text{ con } X_i^* \sim \mathcal{U}\left(0, \hat{\theta}\right).$$

En estas circunstancias es muy sencillo obtener la distribución en el remuestreo de $\hat{\theta}^*$, ya que su deducción es totalmente paralela a la de la distribución en el muestreo de $\hat{\theta}$. Así, la función de densidad de $\hat{\theta}^*$ es

$$\hat{g}(x) = \frac{n}{\hat{\theta}}\left(\frac{x}{\hat{\theta}}\right)^{n-1}, \text{ si } x \in \left[0, \hat{\theta}\right].$$

Con lo cual, al utilizar un bootstrap paramétrico, la distribución en el remuestreo de $R^* = R(\mathbf{X}^*, F_{\hat{\theta}}) = \hat{\theta}^* - \hat{\theta}$ imita a la distribución en muestreo de $R = R(\mathbf{X}, F) = \hat{\theta} - \theta$.

Para implementarlo en la práctica podríamos emplear un código muy similar al del ejemplo original [Figura 9.3]:

```r
# Remuestreo
B <- 2000
maximo <- numeric(B)
estadistico <- numeric(B)
for (k in 1:B) {
    remuestra <- runif(n) * theta_est
    maximo[k] <- max(remuestra)
    estadistico[k] <- maximo[k] - theta_est
}
# Distribución estadístico
xlim <- c(-theta/2, 0) # c(-theta, 0)
hist(estadistico, freq = FALSE, main = "", breaks = "FD",
     border = "darkgray", xlim = xlim)
lines(density(estadistico))
rug(estadistico, col = "darkgray")
curve(n/theta * ((x + theta)/theta)^(n - 1), col = "blue", lty = 2, lwd = 2, add = TRUE)
```

9.3 Bootstrap suavizado

Cuando la distribución poblacional, F, es continua es lógico incorporar dicha información al bootstrap. Eso significa que la función de distribución tiene una función de densidad asociada, relacionadas mediante la expresión: $f(x) = F'(x)$. Una forma de remuestrear de un universo bootstrap continuo es emplear un estimador de la función de densidad, como por ejemplo el estimador no paramétrico tipo núcleo, y remuestrear de él.

Si $(X_1, X_2, \dots, X_n)$ es una muestra aleatoria simple (m.a.s.), de una población con función de distribución F, absolutamente continua, y función de densidad f, el estimador tipo núcleo propuesto por Parzen (1962) y Rosenblatt (1956) viene dado por

$$\hat{f}_h(x) = \frac{1}{nh}\sum_{i=1}^{n} K\left(\frac{x - X_i}{h}\right) = \frac{1}{n}\sum_{i=1}^{n} K_h(x - X_i),$$

donde $K_h(u) = \frac{1}{h}K\left(\frac{u}{h}\right)$, K es una función núcleo (normalmente una densidad simétrica en torno al cero) y $h > 0$ es una parámetro de suavizado, llamado ventana, que regula el tamaño del entorno que se usa para llevar a cabo la estimación.

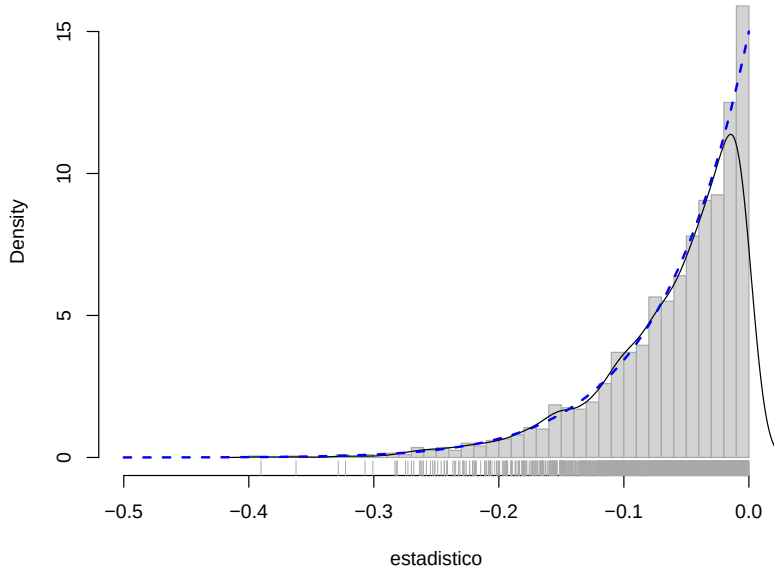


Figura 9.3: Distribución bootstrap paramétrica y distribución poblacional.

Es habitual seleccionar una función núcleo K no negativa y con integral uno:

$$K(u) \geq 0, \ \forall u, \ \int_{-\infty}^{\infty} K(u)\, du = 1$$

(i.e. una función de densidad). Además, frecuentemente K es una función simétrica ($K(-u) = K(u)$). Aunque la elección de esta función no tiene gran impacto en las propiedades del estimador (salvo sus condiciones de regularidad: continuidad, diferenciabilidad, etc.) la elección del parámetro de suavizado sí es muy importante para una correcta estimación. En otras palabras, el tamaño del entorno usado para la estimación no paramétrica debe ser adecuado (ni demasiado grande ni demasiado pequeño).

En `R` podemos emplear la función `density()` del paquete base para obtener una estimación tipo núcleo de la densidad. Los principales parámetros (con los valores por defecto) son los siguientes:

```
density(x, bw = "nrd0", adjust = 1, kernel = "gaussian", n = 512, from, to)
```

- `bw`: ventana, puede ser un valor numérico o una cadena de texto que la determine (en ese caso llamará internamente a la función `bw.xxx()` donde `xxx` se corresponde con la cadena de texto). Las opciones son:
  - `"nrd0"`, `"nrd"`: Reglas del pulgar de Silverman (1986, page 48, eqn (3.31)) y Scott (1992), respectivamente. Como es de esperar que la densidad objetivo no sea tan suave como la normal, estos criterios tenderán a seleccionar ventanas que producen un sobresuavizado de las observaciones.
  - `"ucv"`, `"bcv"`: Métodos de validación cruzada insesgada y sesgada, respectivamente.
  - `"sj"`, `"sj-ste"`, `"sj-dpi"`: Métodos de Sheather y Jones (1991), "solve-the-equation" y "direct plug-in", respectivamente.
- `adjust`: parameto para reescalado de la ventana, las estimaciones se calculan con la ventana `adjust*bw`.
- `kernel`: cadena de texto que determina la función núcleo, las opciones son: `"gaussian"`, `"epanechnikov"`, `"rectangular"`, `"triangular"`, `"biweight"`, `"cosine"` y `"optcosine"`.

- `n`, `from`, `to`: permiten establecer la rejilla en la que se obtendrán las estimaciones (si $n > 512$ se emplea `fft()` por lo que se recomienda establecer `n` a un múltiplo de 2; por defecto `from` y `to` se establecen como `cut = 3` veces la ventana desde los extremos de las observaciones).

Como ejemplo consideraremos el conjunto de datos `precip`, que contiene el promedio de precipitación, en pulgadas de lluvia, de 70 ciudades de Estados Unidos.

```
x <- precip
h <- bw.SJ(x)
npden <- density(x, bw = h)
# npden <- density(x, bw = "SJ")
# h <- npden$bw

# plot(npden)
hist(x, freq = FALSE, breaks = "FD", main = "Kernel density estimation",
     xlab = paste("Bandwidth =", formatC(h)), border = "darkgray",
     xlim = c(0, 80), ylim = c(0, 0.04))
lines(npden, lwd = 2)
rug(x, col = "darkgray")
```

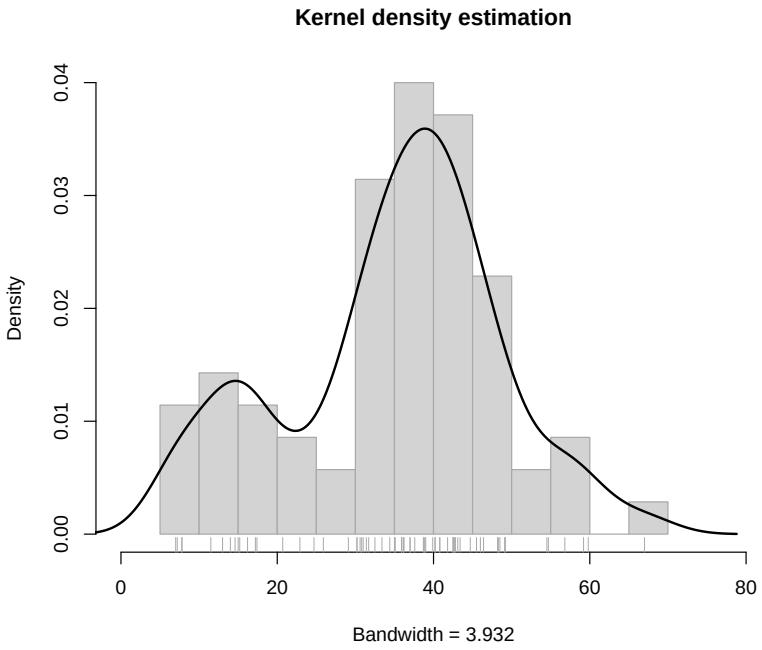


Figura 9.4: Estimación tipo núcleo de la densidad de ‘precip‘.

Alternativamente podríamos emplear implementaciones en otros paquetes de `R`. Uno de los más empleados es `ks` (Duong, 2019), que admite estimación incondicional y condicional multidimensional. También se podrían emplear los paquetes `KernSmooth` (Wand y Ripley, 2019), `sm` (Bowman y Azzalini, 2019), `np` (Tristen y Jeffrey, 2019), `kedd` (Guidoum, 2019), `features` (Duong y Matt, 2019) y `npsp` (Fernández-Casal, 2019), entre otros.

La función de distribución asociada al estimador tipo núcleo de la función de densidad viene dada por

$$\begin{aligned}\hat{F}_h(x) &= \int_{-\infty}^{x} \hat{f}_h(y)\,dy = \int_{-\infty}^{x} \frac{1}{n}\sum_{i=1}^{n}\frac{1}{h}K\left(\frac{y-X_i}{h}\right)dy \\ &= \frac{1}{nh}\sum_{i=1}^{n}\int_{-\infty}^{x} K\left(\frac{y-X_i}{h}\right)dy \\ &= \frac{1}{n}\sum_{i=1}^{n}\int_{-\infty}^{\frac{x-X_i}{h}} K(u)\,du = \frac{1}{n}\sum_{i=1}^{n}\mathbb{K}\left(\frac{x-X_i}{h}\right)\end{aligned}$$

donde $\mathbb{K}$ es la función de distribución asociada al núcleo K, es decir

$$\mathbb{K}(t) = \int_{-\infty}^{t} K(u)\,du.$$

Por ejemplo, en el caso de del conjunto de datos de precipitaciones, el siguiente código compara la estimación tipo núcleo de la distribución con la empírica [Figura 9.5]:

```
Fn <- ecdf(precip)
curve(Fn, xlim = c(0, 75), ylab = "F(x)", type = "s")
Fnp <- function(x) sapply(x, function(y) mean(pnorm(y, precip, h)))
curve(Fnp, lty = 2, add = TRUE)
legend("bottomright", legend = c("Empírica", "Tipo núcleo"), lty = 1:2)
```

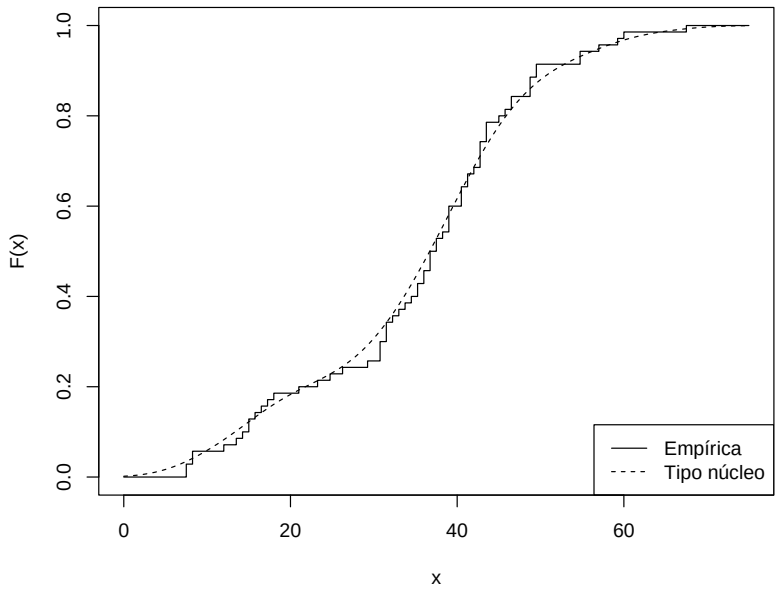


Figura 9.5: Estimación empírica y tipo núcleo de la función de distribución de 'precip'.

Tendiendo en cuenta que el estimador $\hat{f}_h(x)$ es una combinación lineal convexa de funciones de densidad, $K_h(x-X_i)$, con pesos $\frac{1}{n}$, podemos simular valores X^*, procedentes de esta distribución empleando el método de composición descrito en la Sección 4.4. El primer paso consistiría en elegir (aleatoriamente y con equiprobabilidad) un índice $i \in \{1,\ldots,n\}$, y posteriormente simular X^* a partir de la densidad $K_h(\cdot - X_i)$. Este último paso puede realizarse simulando un valor V con densidad K y haciendo $X_i + hV$. Por tanto, podemos pensar que el bootstrap suavizado parte del bootstrap uniforme ($X_i^* = X_{\lfloor nU_i \rfloor+1}$) y añade una perturbación (hV_i), cuya magnitud viene dada por el parámetro de suavizado (h) y cuya forma imita a la de una variable aleatoria (V_i) con densidad K.

Por ejemplo, la función `density()` emplea por defecto un núcleo gaussiano, y como se muestra en la ayuda de esta función, podemos emplear un código como el siguiente para obtener `nsim` simulaciones(ver Figura 9.6):

```
## simulation from a density() fit:
# a kernel density fit is an equally-weighted mixture.
nsim <- 1e6
set.seed(1)
# x_boot <- sample(x, nsim, replace = TRUE)
# x_boot <- x_boot + bandwidth * rnorm(nsim)
x_boot <- rnorm(nsim, sample(x, nsim, replace = TRUE), h)
# Representar
plot(npden, main = "")
lines(density(x_boot), col = "blue", lwd = 2, lty = 2)
```

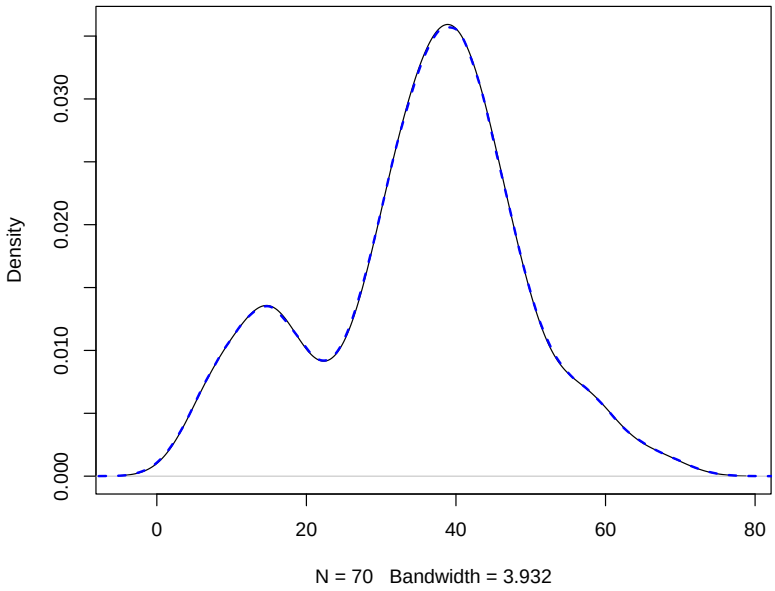


Figura 9.6: Estimaciónes tipo núcleo de las densidades de ‘precip‘ y de una simulación.

En el boostrap suavizado, la distribución de una observación X_i^* de la remuestra bootstrap es continua (puede tomar infinitos valores), mientras que en el bootstrap uniforme es discreta. De esta forma se pueden evitar algunos de los problemas del bootstrap uniforme, como los descritos en la Sección 9.1. Un problema importante es la elección del parámetro de suavizado, h, en este procedimiento de remuestreo. En la práctica es razonable elegir h como un valor bastante pequeño, en relación con la desviación típica de la muestra. Es fácil observar que en el caso extremo $h = 0$ este método de remuestreo se reduce al bootstrap uniforme.

Ejemplo 9.4 (Inferencia sobre la media, continuación)

Continuando con el Ejemplo 8.2 del tiempo de vida de microorganismos, podríamos pensar en emplear bootstrap suavizado para calcular un intervalo de confianza para la media poblacional.

```
muestra <- simres::lifetimes
n <- length(muestra)
alfa <- 0.05
# Estimaciones muestrales
x_barra <- mean(muestra)
```

```
cuasi_dt <- sd(muestra)
# Remuestreo
set.seed(1)
B <- 1000
# h <- 1e-08
# h <- 0.05*cuasi_dt
h <- bw.SJ(muestra)/2
estadistico_boot <- numeric(B)
for (k in 1:B) {
  # remuestra <- sample(muestra, n, replace = TRUE)
  # remuestra <- rexp(n, 1/x_barra)
  remuestra <- rnorm(n, sample(muestra, n, replace = TRUE), h)
  x_barra_boot <- mean(remuestra)
  cuasi_dt_boot <- sd(remuestra)
  estadistico_boot[k] <- sqrt(n) * (x_barra_boot - x_barra)/cuasi_dt_boot
}
# Aproximación Monte Carlo de los ptos críticos:
pto_crit <- quantile(estadistico_boot, c(alfa/2, 1 - alfa/2))
# Construcción del IC
ic_inf_boot <- x_barra - pto_crit[2] * cuasi_dt/sqrt(n)
ic_sup_boot <- x_barra - pto_crit[1] * cuasi_dt/sqrt(n)
IC_boot <- c(ic_inf_boot, ic_sup_boot)
names(IC_boot) <- paste0(100*c(alfa/2, 1-alfa/2), "%")
IC_boot
```

```
##      2.5%     97.5%
## 0.4960975 1.1880279
```

Con el paquete `boot`, la recomendación es implementarlo como un bootstrap paramétrico:

```
library(boot)
ran.gen.smooth <- function(data, mle) {
    # Función para generar muestras aleatorias mediante
    # bootstrap suavizado con función núcleo gaussiana,
    # mle contendrá la ventana.
    n <- length(data)
    h <- mle
    out <- rnorm(n, sample(data, n, replace = TRUE), h)
    out
}

statistic <- function(data){
    c(mean(data), var(data)/length(data))
}

set.seed(1)
res.boot <- boot(muestra, statistic, R = B, sim = "parametric",
                 ran.gen = ran.gen.smooth, mle = h)

boot.ci(res.boot, type = "stud")
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = res.boot, type = "stud")
##
```

```
## Intervals :
## Level    Studentized
## 95%   ( 0.4960,  1.1927 )
## Calculations and Intervals on Original Scale
```

En el Capítulo 6 de Cao y Fernández-Casal (2022) se describen métodos bootstrap diseñados específicamente para hacer inferencia sobre la densidad empleando el estimador no paramétrico tipo núcleo (i.e. empleando bootstrap suavizado), como la construcción de intervalos de confianza o la selección del parámetro de suavizado.

9.4 Bootstrap basado en modelos

En ocasiones nos pueden interesar modelos semiparamétricos, en los que se asume una componente paramétrica pero no se especifica por completo la distribución de los datos. Una de las situaciones más habituales es en regresión, donde se puede considerar un modelo para la tendencia pero sin asumir una forma concreta para la distribución del error.

Nos centraremos en el caso de regresión y consideraremos como base el siguiente modelo general:

$$Y = m(\mathbf{X}) + \varepsilon, \tag{9.1}$$

donde Y es la respuesta, $\mathbf{X} = (X_1, X_2, \dots, X_p)$ es el vector de variables explicativas, $m(\mathbf{x}) = E\left(Y|_{\mathbf{X}=\mathbf{x}}\right)$ es la media condicional, denominada función de regresión (o tendencia), y ε es un error aleatorio de media cero y varianza σ^2, independiente de $\mathbf{X}$ (errores homocedásticos independientes).

Supondremos que el objetivo es, a partir de una muestra:

$$\left\{\left(X_{1_i}, \dots, X_{p_i}, Y_i\right) : i = 1, \dots, n\right\},$$

realizar inferencias sobre la distribución condicional $Y|_{\mathbf{X}=\mathbf{x}}$.

El modelo (9.1) se corresponde con el denominado *diseño aleatorio*, mas general. Alternativamente se podría asumir que los valores de las variables explicativas no son aleatorios (por ejemplo han sido fijados por el experimentador), hablaríamos entonces de *diseño fijo*. Para realizar inferencias sobre modelos de regresión con errores homocedásticos se podrían emplear dos algoritmos bootstrap (e.g. Canty, 2002, y subsecciones siguientes). El primero consistiría en utilizar directamente bootstrap uniforme, remuestreando las observaciones, y sería adecuado para el caso de diseño aleatorio. La otra alternativa, que podría ser más adecuada para el caso de diseño fijo, sería lo que se conoce como *remuestreo residual*, *remuestreo basado en modelos* o *bootstrap semiparamétrico*. En esta aproximación se mantienen fijos los valores de las variables explicativas y se remuestrean los residuos. Una de las aplicaciones del bootstrap semiparamétrico es el contraste de hipótesis en regresión, que se tratará en la Sección 10.3.3.

Se puede generalizar el modelo (9.1) de diversas formas, por ejemplo asumiendo que la distribución del error depende de X únicamente a través de la varianza (error heterocedástico independiente). En este caso se suele reescribir como:

$$Y = m(\mathbf{X}) + \sigma(\mathbf{X})\varepsilon,$$

siendo $\sigma^2(\mathbf{x}) = Var\left(Y|_{\mathbf{X}=\mathbf{x}}\right)$ la varianza condicional y suponiendo adicionalmente que ε tiene varianza uno. Se podría modificar el bootstrap residual para este caso pero habría que modelizar y estimar la varianza condicional. Alternativamente se podría emplear el denominado *Wild Bootstrap* que se describirá en la Sección **??** para el caso de modelos de regresión no paramétricos.

En esta sección nos centraremos en el caso de regresión lineal:

$$m_\beta(\mathbf{x}) = \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \dots + \beta_p X_p,$$

siendo $\beta = \left(\beta_0, \beta_1, \dots, \beta_p\right)^\top$ el vector de parámetros (desconocidos). Su estimador mínimo cuadrático es:

$$\hat{\beta} = \left(X^\top X\right)^{-1} X^\top \mathbf{Y},$$

siendo $\mathbf{Y} = (Y_1, \dots, Y_n)^\top$ el vector de observaciones de la variable Y y X la denominada *matriz del diseño* de las variables regresoras, cuyas filas son los valores observados de las variables explicativas.

En regresión lineal múltiple, bajo las hipótesis estructurales del modelo de normalidad y homocedásticidad, se dispone de resultados teóricos que permiten realizar inferencias sobre características de la distribución condicional. Si alguna de estas hipótesis no es cierta se podrían emplear aproximaciones basadas en resultados asintóticos, pero podrían ser poco adecuadas para tamaños muestrales no muy grandes. Alternativamente se podría emplear bootstrap. Con otros métodos de regresión, como los modelos no paramétricos descritos en el Capítulo **??**, es habitual emplear bootstrap para realizar inferencias sobre la distribución condicional.

En esta sección se empleará el conjunto de datos `Prestige` del paquete `carData`, considerando como variable respuesta `prestige` (puntuación de ocupaciones obtenidas a partir de una encuesta) y como variables explicativas: `income` (media de ingresos en la ocupación) y `education` (media de los años de educación). Para ajustar el correspondiente modelo de regresión lineal podemos emplear el siguiente código:

```
data(Prestige, package = "carData")
# ?Prestige
modelo <- lm(prestige ~ income + education, data = Prestige)
summary(modelo)
```

```
##
## Call:
## lm(formula = prestige ~ income + education, data = Prestige)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -19.4040  -5.3308   0.0154   4.9803  17.6889
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -6.8477787  3.2189771  -2.127   0.0359 *
## income       0.0013612  0.0002242   6.071 2.36e-08 ***
## education    4.1374444  0.3489120  11.858  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.81 on 99 degrees of freedom
## Multiple R-squared:  0.798,  Adjusted R-squared:  0.7939
## F-statistic: 195.6 on 2 and 99 DF,  p-value: < 2.2e-16
```

Como ejemplo, consideraremos que el objetivo es realizar inferencias sobre el coeficiente de determinación ajustado:

```
res <- summary(modelo)
names(res)
```

```
##  [1] "call"          "terms"         "residuals"     "coefficients"
##  [5] "aliased"       "sigma"         "df"            "r.squared"
##  [9] "adj.r.squared" "fstatistic"    "cov.unscaled"
```

```
res$adj.r.squared
```

```
## [1] 0.7939201
```

9.4.1 Remuestreo de las observaciones

Como ya se comentó, en regresión podríamos emplear bootstrap uniforme multidimensional para el caso de diseño aleatorio, aunque hay que tener en cuenta que con este método la distribución en el

remuestreo de $Y^*|_{X^*=X_i}$ es degenerada.

En este caso, podríamos realizar inferencias sobre el coeficiente de determinación ajustado empleando el siguiente código:

```
library(boot)

case.stat <- function(data, i) {
  fit <- lm(prestige ~ income + education, data = data[i, ])
  summary(fit)$adj.r.squared
}

set.seed(1)
boot.case <- boot(Prestige, case.stat, R = 1000)
boot.case
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = Prestige, statistic = case.stat, R = 1000)
##
##
## Bootstrap Statistics :
##      original      bias    std. error
## t1* 0.7939201 0.002495631   0.0315275
```

```
# plot(boot.case)
boot.ci(boot.case, type = c("basic", "perc", "bca"))
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = boot.case, type = c("basic", "perc", "bca"))
##
## Intervals :
## Level      Basic              Percentile            BCa
## 95%   ( 0.7331,  0.8570 )   ( 0.7308,  0.8547 )   ( 0.7203,  0.8497 )
## Calculations and Intervals on Original Scale
```

9.4.2 Bootstrap residual

Como ya se comentó, en el caso de diseño fijo podemos realizar un remuestreo de los residuos:

$$\mathbf{r} = \mathbf{Y} - X\hat{} = \mathbf{Y} - \hat{\mathbf{Y}}$$

obteniéndose las réplicas bootstrap:

$$\mathbf{Y}^* = \hat{\mathbf{Y}} + \mathbf{r}^*.$$

Por ejemplo, adaptando el código en Canty (2002) para este conjunto de datos, podríamos emplear:

```
pres.dat <- Prestige
pres.dat$fit <- fitted(modelo)
pres.dat$res <- residuals(modelo)

mod.stat <- function(data, i) {
    data$prestige <- data$fit + data$res[i]
    fit <- lm(prestige ~ income + education, data = data)
```

```
    summary(fit)$adj.r.squared
}

set.seed(1)
boot.mod <- boot(pres.dat, mod.stat, R = 1000)
boot.mod
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = pres.dat, statistic = mod.stat, R = 1000)
##
##
## Bootstrap Statistics :
##      original      bias    std. error
## t1* 0.7939201 0.004401997  0.02671996
```

```
# plot(boot.mod)
boot.ci(boot.mod, type = c("basic", "perc", "bca"))
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = boot.mod, type = c("basic", "perc", "bca"))
##
## Intervals :
## Level      Basic              Percentile            BCa
## 95%   ( 0.7407,  0.8464 )   ( 0.7415,  0.8471 )   ( 0.7244,  0.8331 )
## Calculations and Intervals on Original Scale
## Some BCa intervals may be unstable
```

Sin embargo, la variabilidad de los residuos no reproduce la de los verdaderos errores, por lo que podría ser preferible (especialmente si el tamaño muestral es pequeño) emplear la modificación descrita en Davison y Hinkley (1997, Alg. 6.3, p. 271). Teniendo en cuenta que:

$$\mathbf{r} = (I - H)\,\mathbf{Y},$$

siendo $H = X\left(X^\top X\right)^{-1} X^\top$ la matriz de proyección. La idea es remuestrear los residuos reescalados (de forma que su varianza sea constante) y centrados $e_i - \bar{e}$, siendo:

$$e_i = \frac{r_i}{\sqrt{1 - h_{ii}}},$$

donde h_{ii} es el valor de influencia o leverage, el elemento i-ésimo de la diagonal de H.

En `R` podríamos obtener estos residuos mediante los comandos[2]:

```
pres.dat$sres <- residuals(modelo)/sqrt(1 - hatvalues(modelo))
pres.dat$sres <- pres.dat$sres - mean(pres.dat$sres)
```

Sin embargo puede ser más cómodo emplear la función `Boot()` del paquete `car` (que internamente llama a la función `boot()`), como se describe en el apéndice "Bootstrapping Regression Models in R" del libro "An R Companion to Applied Regression" de Fox y Weisberg (2018), disponible aquí.

Esta función es de la forma:

[2] Para reescalar los residuos de un modelo `gam` del paquete `mgcv`, como no implementa un método `hatvalues()`, habrá que emplear `influence.gam()` (o directamente `modelo.gam$hat`).

```
Boot(object, f = coef, labels = names(f(object)), R = 999,
     method = c("case", "residual"))
```

donde:

- `object`: es un objeto que contiene el ajuste de un modelo de regresión.
- `f`: es la función de estadísticos (utilizando el ajuste como argumento).
- `method`: especifica el tipo de remuestreo: remuestreo de observaciones (`"case"`) o de residuos (`"residual"`), empleando la modificación descrita anteriormente.

Ejercicio 9.1

Emplear la función `Boot()` del paquete `car` para hacer inferencia sobre el coeficiente de determinación ajustado del modelo de regresión lineal que explica `prestige` a partir de `income` y `education` (obtener una estimación del sesgo y de la predicción, y una estimación por intervalo de confianza de este estadístico).

Capítulo 10

Aplicaciones del remuestreo

Como se ha visto en capítulos anteriores, los métodos bootstrap permiten aproximar la distribución en el muestreo de un estadístico. En particular nos pueden interesar determinadas características de esta distribución, como el sesgo y la precisión de un estimador (Sección 10.1). Otro ejemplo es la construcción de intervalos de confianza, donde la características de interés son los cuantiles del estadístico pivotal empleado (Sección 10.2). También los contrastes de hipótesis, donde interesa aproximar cuantiles de la distribución muestral del estadístico del contraste bajo la hipótesis nula (Sección 10.3)

10.1 Estimación del sesgo y la precisión de un estimador

Como ya se comentó, una de las principales aplicaciones de los métodos bootstrap es la aproximación del sesgo y de la varianza de un estimador. Al igual que en capítulos anteriores, supongamos que $\mathbf{X} = (X_1, \dots, X_n)$ es una m.a.s. de una población con distribución F y que tenemos interés en realizar inferencia sobre un parámetro de la población $\theta = \theta(F)$ empleando un estimador $\hat{\theta} = T(\mathbf{X})$.

Podemos considerar el estadístico

$$R = R(\mathbf{X}, F) = T(\mathbf{X}) - \theta(F) = \hat{\theta} - \theta,$$

para aproximar características de la distribución del estimador, ya que:

$$\begin{aligned} Sesgo\left(\hat{\theta}\right) &= E\left(\hat{\theta} - \theta\right) = E(R), \\ Var\left(\hat{\theta}\right) &= Var\left(\hat{\theta} - \theta\right) = Var(R), \\ MSE\left(\hat{\theta}\right) &= E\left[\left(\hat{\theta} - \theta\right)^2\right] = E(R^2). \end{aligned}$$

Para ello, como se mostró en capítulos anteriores, consideraríamos una aproximación $\hat{F}$ de la distribución de probabilidad (por ejemplo, $\hat{F} = F_n$ si se considera que el bootstrap uniforme es adecuado) y emplearíamos Monte Carlo para aproximar la correspondiente distribución bootstrap:

1. Para cada $i = 1, \dots, n$ generar X_i^* a partir de $\hat{F}$ y obtener $\mathbf{X}^* = (X_1^*, \dots, X_n^*)$.
2. Calcular $R^* = R\left(\mathbf{X}^*, \hat{F}\right) = T(\mathbf{X}^*) - \theta\left(\hat{F}\right) = \hat{\theta}^* - \hat{\theta}$.
3. Repetir B veces los pasos 1-2 para obtener las réplicas bootstrap $R^{*(1)}, \dots, R^{*(B)}$.
4. Utilizar las réplicas bootstrap para aproximar las características de interés de la distribución en el muestreo de R.
   - Estimación bootstrap del sesgo:

$$Sesgo^*\left(\hat{\theta}^*\right) = \bar{R}^* = \frac{1}{B}\sum_{b=1}^{B} R^{*(b)}.$$

- Estimación bootstrap de la varianza:

$$Var^*\left(\hat{\theta}^*\right) = \frac{1}{B}\sum_{b=1}^{B}\left(R^{*(b)} - \bar{R}^*\right)^2.$$

- Estimación bootstrap del error cuadrático medio:

$$MSE^*\left(\hat{\theta}^*\right) = \frac{1}{B}\sum_{b=1}^{B} R^{*(b)2}.$$

Ejemplo 10.1 (media y media truncada)

Como ejemplo consideramos la aproximación del sesgo y de la varianza de la media y la media truncada al 10% como estimadores de la media teórica del tiempo de vida de microorganismos mediante bootstrap uniforme (ver Ejercicio 8.2).

```
library(boot)
muestra <- simres::lifetimes

statistic <- function(data, i){
  remuestra <- data[i]
  c(mean(remuestra), mean(remuestra, trim = 0.1))
}

set.seed(1)
res.boot <- boot(muestra, statistic, R = 1000)
res.boot
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = muestra, statistic = statistic, R = 1000)
##
##
## Bootstrap Statistics :
##      original      bias    std. error
## t1* 0.8053333 0.003173267   0.1583306
## t2* 0.7582308 0.011902692   0.1739443
```

Lamentablemente la función `print.boot()` calcula las aproximaciones bootstrap del sesgo y de la precisión pero no las almacena. En el caso más simple podríamos obtenerlas con el siguiente código:

```
op <- with(res.boot, cbind(
  t0, apply(t, 2, mean, na.rm = TRUE) -  t0,
  apply(t, 2, sd, na.rm = TRUE)
  ))
rownames(op) <- paste0("t", 1:ncol(res.boot$t), "*")
colnames(op) <- c("original", "bias  ", " std. error")
op
```

```
##      original      bias    std. error
## t1* 0.8053333 0.003173267   0.1583306
## t2* 0.7582308 0.011902692   0.1739443
```

Ejercicio 10.1

Como continuación del ejemplo mostrado en la Sección 8.3.3, emplear el paquete `boot` para aproximar el sesgo y la precisión del coeficiente de correlación lineal r entre `prestige` e `income` del conjunto de

datos `Prestige` mediante bootstrap uniforme multidimensional. Comparar los resultados con los obtenidos mediante la aproximación asintótica normal, que como se indicó en el Ejercicio 8.1, consideraría que el sesgo es nulo y como estimación de su varianza:

$$\widehat{Var}(r) = \frac{1-r^2}{n-2}.$$

Como también se comentó en la introducción del Capítulo 8, se pueden emplear otros tipos de remuestreo para la aproximación del sesgo y varianza de un estimador. El más conocido es el *jackknife*, que es uno de los métodos de remuestreo más antiguos (propuesto inicialmente por Quenouille, 1949). De hecho el bootstrap surgió (Efron, 1979) como una alternativa a este tipo de remuestreo. En el jackknife se consideran las n remuestras obtenidas al ir eliminando cada una de las observaciones:

$$\mathbf{X}^* = \mathbf{X}_{(i)} = \left(X_1, \dots, X_{i-1}, X_{i+1}, \dots, X_n\right), \; i = 1, \dots, n.$$

Para cada una de estas remuestras se obtienen las correspondientes réplicas del estadístico

$$\hat{\theta}_{(i)} = T\left(\mathbf{X}_{(i)}\right), \; i = 1, \dots, n.$$

A partir de las cuales se aproxima el sesgo y la varianza, considerando un factor de elevación $n-1$ para que sean insesgadas:

$$Sesgo^*_{jackk}\left(\hat{\theta}^*\right) = (n-1)\left(\overline{\theta_{(\cdot)}} - \hat{\theta}\right) = \frac{n-1}{n}\sum_{i=1}^{n}\left(\hat{\theta}_{(i)} - \hat{\theta}\right),$$

$$Var^*_{jackk}\left(\hat{\theta}^*\right) = \frac{n-1}{n}\sum_{i=1}^{n}\left[\hat{\theta}_{(i)} - \overline{\theta_{(\cdot)}}\right]^2,$$

donde $\overline{\theta_{(\cdot)}} = \frac{1}{n}\sum_{j=1}^{n}\hat{\theta}_{(j)}$. Para más detalles, ver Sección 2.2 de Cao y Fernández-Casal (2021).

10.2 Intervalos de confianza bootstrap

En esta sección consideraremos el problema de construcción, mediante bootstrap, de un intervalo de confianza bilateral, con nivel de confianza $1-\alpha$, para un parámetro θ de la distribución F. Una vez elegido el método bootstrap adecuado, teniendo en cuenta la información disponible en el contexto del que se trate, otro aspecto importante es el método para la construcción del intervalo de confianza bootstrap de forma que la probabilidad de cobertura sea lo más parecida posible al nivel nominal $1-\alpha$.

Las diferencias entre los distintos métodos dependen del estadístico R empleado y de las suposiciones sobre su distribución. Como se comentó en la Sección 8.3.1, la función `boot.ci()` del paquete `boot` permite construir distintos tipos de intervalos de confianza dependiendo del parámetro `type`. En el Ejemplo 8.3 se ilustra la obtención de estimaciones por intervalo de confianza para la media empleando los distintos métodos bajo bootstrap uniforme (en el Capítulo **??** se incluyen ejemplos adicionales empleando bootstrap paramétrico y suavizado).

En esta sección se describirán brevemente los distintos métodos implementados en la función `boot.ci()`. Para un tratamiento más detallado, incluyendo los órdenes de los errores de cobertura, ver por ejemplo el Capítulo 4 de Cao y Fernández-Casal (2021) o el Capítulo 5 de Davison y Hinkley (1997).

10.2.1 Aproximación normal

Este método emplea las aproximaciones bootstrap del sesgo $Sesgo^*\left(\hat{\theta}^*\right)$ y de la varianza $Var^*\left(\hat{\theta}^*\right)$, y asume que la distribución del correspondiente estadístico studentizado es una normal estándar

$$\frac{\hat{\theta} - Sesgo^*\left(\hat{\theta}^*\right) - \theta}{\sqrt{Var^*\left(\hat{\theta}^*\right)}} \underset{aprox}{\sim} \mathcal{N}(0,1).$$

De esta forma se obtiene la estimación por intervalo de confianza:

$$\hat{I}_{norm} = \left(\hat{\theta} - Sesgo^*\left(\hat{\theta}^*\right) - z_{1-\alpha/2}\sqrt{Var^*\left(\hat{\theta}^*\right)}, \hat{\theta} - Sesgo^*\left(\hat{\theta}^*\right) + z_{1-\alpha/2}\sqrt{Var^*\left(\hat{\theta}^*\right)}\right).$$

Podemos obtener este intervalo de confianza estableciendo `type = "norm"` (o `type = "all"`) en la llamada a la función `boot.ci()` (ver Ejemplo 8.3).

10.2.2 Método percentil directo

Este método se basa en la construcción del intervalo de confianza, mediante bootstrap, empleando como estadístico el estimador

$$R = \hat{\theta}.$$

Una vez elegido el método de remuestreo, empleando un estimador, $\hat{F}$, de la distribución poblacional, F, la distribución en el muestreo de $R = \hat{\theta}$ se aproxima directamente mediante la distribución bootstrap de $R^* = \hat{\theta}^*$. A partir de las réplicas bootstrap del estimador aproximamos los cuantiles $x_{\alpha/2}$ y $x_{1-\alpha/2}$ (denotando por x_β el valor verificando $P^*\left(R^* \leq x_\beta\right) = \beta$), de forma que

$$\begin{aligned} 1-\alpha = 1 - \frac{\alpha}{2} - \frac{\alpha}{2} &= P^*\left(\hat{\theta}^* < x_{1-\alpha/2}\right) - P^*\left(\hat{\theta}^* \leq x_{\alpha/2}\right) \\ &= P^*\left(x_{\alpha/2} < \hat{\theta}^* < x_{1-\alpha/2}\right), \end{aligned}$$

y asumimos que esto aproxima lo que ocurre con la distribución poblacional

$$P\left(x_{\alpha/2} < \hat{\theta} < x_{1-\alpha/2}\right) \approx 1-\alpha.$$

De donde se obtiene el intervalo de confianza bootstrap calculado por el método percentil directo

$$\hat{I}_{perc} = \left(x_{\alpha/2}, x_{1-\alpha/2}\right).$$

Una ventaja de los intervalos construidos con este método es que son invariantes frente a transformaciones del estimador (en el caso de que fuese más adecuado trabajar en otra escala, no sería necesario conocer la transformación). Sin embargo, como se comentó en la Sección 8.1, la precisión puede verse seriamente afectada en el caso de estimadores sesgados.

Podemos obtener este intervalo de confianza estableciendo `type = "perc"` (o `type = "all"`) en la llamada a la función `boot.ci()` (ver Ejemplo 8.3).

10.2.3 Método percentil básico

En este método se emplea como estadístico el estimador centrado (no estandarizado)

$$R = \hat{\theta} - \theta.$$

De forma análoga, la distribución en el muestreo de R se aproxima mediante la distribución bootstrap de

$$R^* = \hat{\theta}^* - \theta\left(\hat{F}\right) = \hat{\theta}^* - \hat{\theta}.$$

A partir de las réplicas bootstrap del estadístico se aproximan los cuantiles $x_{\alpha/2}$ y $x_{1-\alpha/2}$ tales que

$$1-\alpha = P^*\left(x_{\alpha/2} < R^* < x_{1-\alpha/2}\right),$$

tomándolo como aproximación de lo que ocurre con la distribución poblacional

$$\begin{aligned} 1-\alpha &\approx P\left(x_{\alpha/2} < R < x_{1-\alpha/2}\right) \\ &= P\left(x_{\alpha/2} < \hat{\theta} - \theta < x_{1-\alpha/2}\right) \\ &= P\left(\hat{\theta} - x_{1-\alpha/2} < \theta < \hat{\theta} - x_{\alpha/2}\right). \end{aligned}$$

De donde se obtiene el intervalo de confianza bootstrap calculado por el método percentil básico

$$\hat{I}_{basic} = \left(\hat{\theta} - x_{1-\alpha/2}, \hat{\theta} - x_{\alpha/2}\right).$$

Podemos obtener este intervalo de confianza estableciendo `type = "basic"` (o `type = "all"`) en la llamada a la función `boot.ci()` (ver Ejemplo 8.3).

10.2.4 Método percentil-t

Este método bootstrap, construye un intervalo de confianza bootstrap a partir del estadístico studentizado:

$$R = \frac{\hat{\theta} - \theta}{\sqrt{\widehat{Var}(\hat{\theta})}}.$$

Procediendo de modo análogo a los casos anteriores, se aproximan los cuantiles $x_{\alpha/2}$ y $x_{1-\alpha/2}$ tales que

$$1 - \alpha = P^* \left(x_{\alpha/2} < R^* < x_{1-\alpha/2} \right),$$

a partir de los cuales se obtiene el intervalo de confianza bootstrap calculado por el método percentil-t (o percentil studentizado)

$$\hat{I}_{stud} = \left(\hat{\theta} - x_{1-\alpha/2}\sqrt{\widehat{Var}(\hat{\theta})}, \hat{\theta} - x_{\alpha/2}\sqrt{\widehat{Var}(\hat{\theta})} \right).$$

Si uno de los componentes del vector de estadísticos (por defecto el segundo) es una estimación de la varianza del estimador (por defecto el primer componente), podemos obtener este intervalo de confianza estableciendo `type = "stud"` (o `type = "all"`) en la llamada a la función `boot.ci()` (ver Ejemplo 8.3). En caso de que el primer y segundo componente del vector de estadísticos no sean el estimador y su varianza estimada, respectivamente, habrá que emplear el argumento `index`, que debe ser un vector de longitud 2 con las posiciones correspondientes.

Hay una variante de este método, denominada percentil-t simetrizado, en la que se asume que la distribución del estadístico es simétrica (aunque no está implementado en `boot.ci()`). Asumiendo que esta suposición es correcta, podemos calcular los cuantiles de forma más eficiente (ya que dispondríamos del doble de información sobre las colas de la distribución). En lugar de tomar cuantiles que dejen colas iguales ($\frac{\alpha}{2}$) a la izquierda y a la derecha, respectivamente, se considera el valor $x_{1-\alpha}$ que verifica $P^*\left(|R^*| \leq x_{1-\alpha}\right) = 1 - \alpha$. Así se obtiene el intervalo de confianza bootstrap

$$\hat{I}_{simstud} = \left(\hat{\theta} - x_{1-\alpha}\sqrt{\widehat{Var}(\hat{\theta})}, \hat{\theta} + x_{1-\alpha}\sqrt{\widehat{Var}(\hat{\theta})} \right).$$

Ejemplo 10.2 (IC bootstrap para la media mediante el método percentil-*t* simetrizado)

Continuando con el Ejemplo 8.2 de inferencia sobre la media con varianza desconocida, podríamos obtener una estimación por intervalo de confianza del tiempo de vida medio de los microorganismos empleando el método bootstrap percentil-t simetrizado con el siguiente código:

```
muestra <- simres::lifetimes
n <- length(muestra)
alfa <- 0.05
x_barra <- mean(muestra)
cuasi_dt <- sd(muestra)

# Remuestreo
set.seed(1)
B <- 1000
estadistico_boot <- numeric(B)
for (k in 1:B) {
  remuestra <- sample(muestra, n, replace = TRUE)
  x_barra_boot <- mean(remuestra)
  cuasi_dt_boot <- sd(remuestra)
  estadistico_boot[k] <- sqrt(n) * abs(x_barra_boot - x_barra)/cuasi_dt_boot
}

# Aproximación bootstrap del pto crítico
pto_crit <- quantile(estadistico_boot, 1 - alfa)
```

```
# Construcción del IC
ic_inf_boot <- x_barra - pto_crit * cuasi_dt/sqrt(n)
ic_sup_boot <- x_barra + pto_crit * cuasi_dt/sqrt(n)
IC_boot <- c(ic_inf_boot, ic_sup_boot)
names(IC_boot) <- paste0(100*c(alfa/2, 1-alfa/2), "%")
IC_boot
```

```
##      2.5%     97.5%
## 0.4334742 1.1771924
```

10.2.5 Método BCa

El método BCa (bias-corrected and accelerated) propuesto por Efron (1987) considera una transformación de forma que la distribución se aproxime a la normalidad, construye el intervalo en esa escala asumiendo normalidad y transforma el resultado a la escala original empleando la distribución bootstrap. El intervalo obtenido es de la forma:

$$\hat{I}_{perc} = \left(x_{\alpha/2}, x_{1-\alpha/2}\right),$$

donde

$$x_u = \hat{G}^{-1}\left(\Phi\left(z + \frac{z + z_u}{1 - a(z + z_u)}\right)\right),$$

siendo $\hat{G}$ la distribución empírica de $\hat{\theta}^*$, $\Phi(z)$ la función de distribución de la normal estándar, $z_u = \Phi^{-1}(u)$ el correspondiente cuantil y:

- $z = \Phi^{-1}(\hat{G}(\hat{\theta}))$ un factor de corrección de sesgo.
- a la denominada constante aceleradora (o corrección de asimetría), que suele ser aproximada mediante jackknife.

Podemos obtener este intervalo de confianza estableciendo `type = "bca"` (o `type = "all"`) en la llamada a la función `boot.ci()` (ver Ejemplo 8.3). Para más detalles ver Sección 5.3.2 de Davison y Hinkley (1997).

Ejercicio 10.2

Como continuación del ejemplo mostrado en la Sección 8.3.3, y de los Ejercicios 8.1 y 10.1, emplear el paquete `boot` para obtener estimaciones por intervalo de confianza del coeficiente de correlación lineal r entre `prestige` e `income` del conjunto de datos `Prestige` (mediante bootstrap uniforme multidimensional). En el caso del método percentil-t, como se indicó en el Ejercicio 8.1, considerar el estimador de la varianza:

$$\widehat{Var}(r) = \frac{1 - r^2}{n - 2}.$$

Comparar los resultados con la aproximación paramétrica implementada en la función `cor.test` y descrita en la siguiente sección.

10.2.6 Ejemplo: IC bootstrap para el coeficiente de correlación

Supongamos de nuevo que queremos estudiar la correlación entre dos variables X e Y a partir del coeficiente de correlación lineal de Pearson:

$$\rho = \frac{Cov\,(X, Y)}{\sigma\,(X)\,\sigma\,(Y)},$$

empleando como estimador el coeficiente de correlación muestral:

$$r = \frac{\sum_{i=1}^{n}(x_i - \overline{x})(y_i - \overline{y})}{\sqrt{\sum_{i=1}^{n}(x_i - \overline{x})^2}\sqrt{\sum_{i=1}^{n}(y_i - \overline{y})^2}}.$$

Para realizar inferencias sobre el coeficiente de correlación, como aproximación más simple, se puede considerar que la distribución muestral de r es aproximadamente normal (ver Ejercicio 8.1) y emplear el estadístico:

$$\frac{r-\rho}{\sqrt{\frac{1-r^2}{n-2}}} \underset{aprox}{\sim} t_{n-2} \tag{10.1}$$

Pero esta aproximación solo sería válida en el caso de muestras grandes (o si la distribución bivariante de (X, Y) es aproximadamente normal) cuando la correlación entre las variables es débil o moderada. En caso contrario la distribución muestral de r puede ser muy asimétrica y los resultados obtenidos con el estadístico anterior no ser muy adecuados (esto concuerda con lo observado en la Sección 8.3.3, al emplear bootstrap uniforme multidimensional para hacer inferencia sobre $R = r - \rho$). Para evitar este problema se suelen obtener intervalos de confianza para ρ empleando la transformación Z de Fisher (1915):

$$Z = \frac{1}{2}\ln\left(\frac{1+r}{1-r}\right) = \operatorname{arctanh}(r),$$

que es una transformación (aprox.) normalizadora y estabilizadora de la varianza. Suponiendo que (X, Y) es normal bivariante y que hay independencia entre las observaciones:

$$Z \sim \mathcal{N}\left(\frac{1}{2}\ln\left(\frac{1+\rho}{1-\rho}\right), \frac{1}{n-3}\right).$$

El intervalo de confianza asintótico se obtiene empleando la aproximación normal tradicional en la escala Z y aplicando posteriormente la transformación inversa:

$$r = \frac{\exp(2Z)-1}{\exp(2Z)+1} = \tanh(Z).$$

Esta aproximación está implementada en la función `cor.test()` del paquete base `stat` de R[1], además de que también realiza el contraste $H_0 : \rho = 0$ empleando el estadístico (10.1).

Continuando con el ejemplo de la Sección 8.3.3 (y de los Ejercicios 8.1, 10.1 y 10.2), para obtener un intervalo de confianza para el coeficiente de correlación lineal entre las variables `income` y `prestige` del conjunto de datos `Prestige`, podríamos emplear el siguiente código:

```
data(Prestige, package="carData")
# with(Prestige, cor.test(income, prestige))
cor.test(Prestige$income, Prestige$prestige)
```

```
##
##  Pearson's product-moment correlation
##
## data:  Prestige$income and Prestige$prestige
## t = 10.224, df = 100, p-value < 2.2e-16
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
##  0.6044711 0.7983807
## sample estimates:
##       cor
## 0.7149057
```

También es de esperar que mejore la precisión de los intervalos de confianza bootstrap si se emplea una transformación que estabilice la varianza del estimador, especialmente en el caso del método basado en la aproximación normal y del bootstrap percentil básico. La función `boot.ci()` del paquete `boot` permite obtener intervalos de confianza calculados en una escala transformada del estadístico, mediante los parámetros:

[1] Se puede obtener el código tecleando en la consola `stats:::cor.test.default`.

- `h`: función vectorial que define la transformación. Los intervalos se calculan en la escala de $h(t)$ y se aplica la función inversa (si se especifica) para transformarlos a la escala original.
- `hinv`: (opcional) función inversa de la transformación (si no se especifica solo se calculan los intervalos en la escala transformada).
- `hdot`: (opcional en el método percentil o básico) función derivada de la transformación (empleada por algunos métodos para aproximar la varianza en la escala transformada mediante el método delta).

Por ejemplo, para considerar la transformación Z de Fisher en este caso, se podría emplear el siguiente código:

```
library(boot)

statistic <- function(data, i){
  remuestra <- data[i, ]
  cor(remuestra$income, remuestra$prestige)
}

set.seed(1)
res.boot <- boot(Prestige, statistic, R = 1000)

h <- function(t) atanh(t)
hdot <- function(t) 1/(1 - t^2)
hinv <- function(t) tanh(t)

# boot.ci(res.boot, type = "norm", h = h)
boot.ci(res.boot, type = "norm", h = h, hdot = hdot, hinv = hinv)
```

```
## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = res.boot, type = "norm", h = h, hdot = hdot,
##     hinv = hinv)
##
## Intervals :
## Level      Normal
## 95%   ( 0.6016,  0.7858 )
## Calculations on Transformed Scale;  Intervals on Original Scale
```

Esto sería en principio preferible a trabajar en la escala original, ya que la distribución bootstrap en la escala transformada se aproximaría más a la normalidad:

```
ht <- h(res.boot$t)
hist(ht, freq = FALSE, breaks = "FD", main = "")
curve(dnorm(x, mean=mean(ht), sd=sd(ht)), lty = 2, add = TRUE)
```

10.3 Contrastes de hipótesis bootstrap

El objetivo de los contrastes de hipótesis es, a partir de la información que proporciona una muestra, decidir (tratando de controlar el riesgo de equivocarse al no disponer de toda la información) entre dos hipótesis sobre alguna característica de interés de la población: hipótesis nula (H_0) e hipótesis alternativa (H_1).

Entre los distintos tipos de contrastes de hipótesis (e.g. paramétricos, no paramétricos, ...), nos centraremos principalmente en los contrastes de bondad de ajuste. En este caso interesará distinguir principalmente entre hipótesis nulas simples (especifican un único modelo) y compuestas (especifican

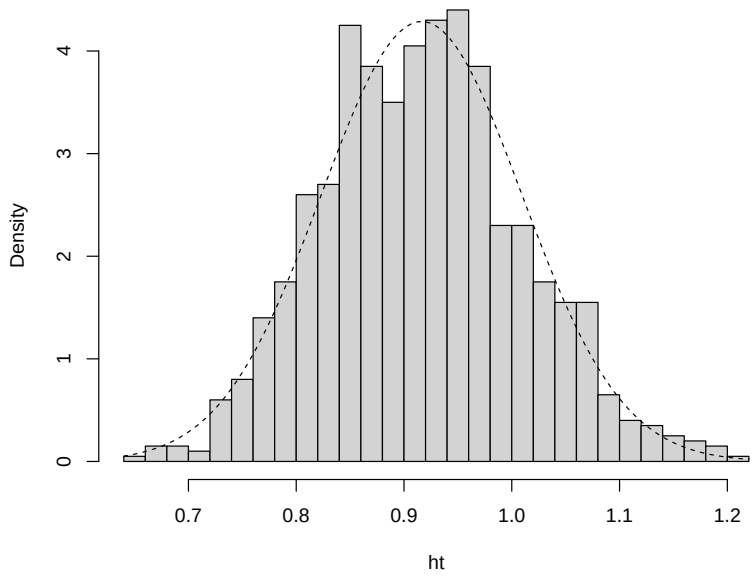


Figura 10.1: Distribución bootstrap en la escala transformada.

un conjunto/familia de modelos).

Para realizar el contraste se emplea un estadístico $D(X_1, \dots, X_n; H_0)$, que mide la discrepancia entre la muestra observada y la hipótesis nula, con distribución conocida (o que se puede aproximar) bajo H_0. Por ejemplo, en el caso de una hipótesis nula paramétrica es habitual emplear un estadístico studentizado de la forma:

$$D(X_1, \dots, X_n; H_0) = \frac{\hat{\theta} - \theta_0}{\hat{\sigma}_{\hat{\theta}}}$$

(o algún tipo de razón de verosimilitudes).

La regla de decisión depende de la hipótesis alternativa y del riesgo asumible al rechazar H_0 siendo cierta:

$$P(\text{rechazar } H_0 \mid H_0 \text{ cierta}) = \alpha,$$

denominado nivel de significación. Se determina una región de rechazo (RR) a partir de los valores que tiende a tomar el estadístico cuando H_1 es cierta, de forma que[2]:

$$P(D \in RR \mid H_0 \text{ cierta}) = \alpha.$$

Se rechaza la hipótesis nula cuando el valor observado del estadístico $\hat{d} = D(x_1, \dots, x_n; H_0)$ pertenece a la región de rechazo.

Para medir el nivel de evidencia en contra de H_0 se emplea el p-valor del contraste (también denominado valor crítico o tamaño del contraste), el menor valor del nivel de significación para el que se rechaza H_0 (que se puede interpretar también como la probabilidad de obtener una discrepancia mayor o igual que $\hat{d}$ cuando H_0 es cierta).

El cálculo del p-valor dependerá por tanto de la hipótesis altervativa. Por ejemplo, si el estadístico del contraste tiende a tomar valores grandes cuando H_0 es falsa (contraste unilateral derecho):

$$p = P\left(D \geq \hat{d} \middle| H_0\right).$$

[2]Aunque cuando la hipótesis nula es compuesta: $P(D \in RR \mid H_0 \text{ cierta}) \leq \alpha$.

En otros casos (contrastes bilaterales) hay evidencias en contra de H_0 si el estadístico toma valores significativamente grandes o pequeños. En estos casos la distribución del estadístico del contraste bajo H_0 suele ser simétrica en torno al cero, por lo que:

$$p = 2P\left(D \geq |\hat{d}| \middle| H_0\right).$$

Pero si esta distribución es asimétrica:

$$p = 2\min\left\{P\left(D \leq \hat{d} \middle| H_0\right), P\left(D \geq \hat{d} \middle| H_0\right)\right\}.$$

La regla de decisión a partir del p-valor es siempre la misma. Rechazamos H_0, al nivel de significación α, si $p \leq \alpha$, en cuyo caso se dice que el contraste es estadísticamente significativo (rechazamos H_0 con mayor seguridad cuanto más pequeño es el p-valor). Por tanto, la correspondiente variable aleatoria $\mathcal{P}$ debería verificar:

$$P\left(\mathcal{P} \leq \alpha | H_0\right) = \alpha.$$

Es decir, la distribución del p-valor bajo H_0 debería ser $\mathcal{U}(0,1)$ (si la distribución del estadístico del constrate es continua).

En los métodos tradicionales de contrastes de hipótesis se conoce o se puede aproximar la distribución del estadístico del contraste bajo H_0. Muchas de estas aproximaciones están basadas en resultados asintóticos y pueden no ser adecuadas para tamaños muestrales pequeños. En ese caso, o si no se dispone de estas herramientas, se puede recurrir a métodos de remuestreo para aproximar el p-valor. Uno de los procedimientos más antiguos es el denominado *contraste de permutaciones* (Fisher, 1935; Pitman, 1937; Welch, 1937). Aunque el bootstrap paramétrico y el semiparamétrico son los procedimientos de remuestreo más empleados para aproximar la distribución del estadístico de contraste bajo la hipótesis nula.

La idea es obtener remuestras de una aproximación de la distribución del estadístico bajo H_0. En el bootstrap paramétrico y semiparamétrico se estima la distribución de los datos bajo la hipótesis nula, $\hat{F}_0$, y se obtienen réplicas del estadístico a partir de remuestras de esta distribución (no sería adecuado emplear directamente la distribución empírica). En el caso de los contrastes de permutaciones las remuestras se obtienen directamente de los datos, remuestreando sin reemplazamiento los valores de la respuesta (y manteniendo fijas las covariables).

Finalmente, se emplean las réplicas bootstrap del estadístico $d_1^*, \ldots, d_B^*$ para aproximar el p-valor. Por ejemplo, en el caso de un contraste unilateral en el que el estadístico del contraste tiende a tomar valores grandes si la hipótesis nula es falsa, se podría emplear como aproximación:

$$p_{boot} = \frac{1}{B}\#\left\{d_i^* \geq \hat{d}\right\}.$$

Mientras que en el caso bilateral, asumiendo que la distribución del estadístico no es necesariamente simétrica, habría que emplear:

$$p_{boot} = \frac{2}{B}\min\left(\#\left\{d_i^* \leq \hat{d}\right\}, \#\left\{d_i^* \geq \hat{d}\right\}\right).$$

10.3.1 Contrastes bootstrap paramétricos

En los casos en los que la hipótesis nula especifica por completo la distribución (hipótesis nula simple) o solo desconocemos los valores de algunos parámetros (hipótesis nula paramétrica compuesta) podemos emplear bootstrap paramétrico para obtener las remuestras bootstrap de los datos (realmente en el primer caso se trataría de simulaciones Monte Carlo). Siempre hay que tener en cuenta que las réplicas bootstrap del estadístico se deberían obtener empleando el mismo procedimiento utilizado en la muestra (p.e. reestimando los parámetros si es el caso).

Ejemplo 10.3 (Contraste de Kolmogorov-Smirnov)

Se trata de un contraste de bondad de ajuste (similar a la prueba de Cramer-von Mises o a la de Anderson-Darling, implementadas en el paquete `goftest` de R, que son en principio mejores). A

partir de $X_1, \dots, X_n$ m.a.s. de X con función de distribución F, se pretende contrastar:

$$\begin{cases} H_0 : F = F_0 \\ H_1 : F \neq F_0 \end{cases}$$

siendo F_0 una función de distribución continua. El estadístico empleado para ello compara la función de distribución bajo H_0 (F_0) con la empírica (F_n):

$$\begin{aligned} D_n &= \sup_x |F_n(x) - F_0(x)| \\ &= \max_{1 \leq i \leq n} \left\{ |F_n(X_{(i)}) - F_0(X_{(i)})|, |F_n(X_{(i-1)}) - F_0(X_{(i)})| \right\} \\ &= \max_{1 \leq i \leq n} \left\{ \frac{i}{n} - F_0(X_{(i)}),\ F_0(X_{(i)}) - \frac{i-1}{n} \right\} \\ &= \max_{1 \leq i \leq n} \left\{ D^+_{n,i},\ D^-_{n,i} \right\}, \end{aligned}$$

y su distribución bajo H_0 no depende F_0 (es de distribución libre), si H_0 es simple y F_0 es continua. Esta distribución está tabulada (para tamaños muestrales grandes se utiliza la aproximación asintótica) y se rechaza H_0 si el valor observado d del estadístico es significativamente grande:

$$p = P(D_n \geq d) \leq \alpha.$$

Este método está implementado en la función `ks.test()` del paquete base de R:

```
ks.test(x, y, ...)
```

donde `x` es un vector que contiene los datos, `y` es una función de distribución (o una cadena de texto que la especifica; también puede ser otro vector de datos para el contraste de dos muestras) y `...` representa los parámetros de la distribución.

Si H_0 es compuesta, el procedimiento habitual es estimar los parámetros desconocidos por máxima verosimilitud y emplear $\hat{F}_0$ en lugar de F_0. Sin embargo, al proceder de esta forma es de esperar que $\hat{F}_0$ se aproxime más que F_0 a la distribución empírica, por lo que los cuantiles de la distribución de D_n pueden ser demasiado conservativos (los p-valores tenderán a ser mayores de lo que deberían) y se tenderá a aceptar la hipótesis nula.

Para evitar este problema, en el caso de contrastar normalidad se desarrolló el test de Lilliefors, implementado en la función `lillie.test()` del paquete `nortest` (también hay versiones en este paquete para los métodos de Cramer-von Mises y Anderson-Darling). Como ejemplo analizaremos el comportamiento de ambos métodos para contrastar normalidad considerando 1000 pruebas con muestras de tamaño 30 de una $\mathcal{N}(0,1)$ (estudiaremos el *tamaño de los contrastes*).

```r
# Valores iniciales
library(nortest)
set.seed(1)
nx <- 30
mx <- 0
sx <- 1
nsim <- 1000
# Realizar contrastes
pvalor.ks <- numeric(nsim)
pvalor.lil <- numeric(nsim)
for(isim in 1:nsim) {
  rx <- rnorm(nx, mx, sx)
  pvalor.ks[isim] <- ks.test(rx, "pnorm", mean(rx), sd(rx))$p.value
  pvalor.lil[isim] <- lillie.test(rx)$p.value
}
```

Bajo la hipótesis nula el p-valor debería de seguir una distribución uniforme, por lo que podríamos generar el correspondiente histograma para estudiar el tamaño del contraste. Alternativamente podríamos representar su función de distribución empírica, que se correspondería con la proporción de rechazos para los distintos niveles de significación.

```
old.par <- par(mfrow=c(2, 2))
# Test de KS
  # Histograma
hist(pvalor.ks, freq=FALSE, main = "")
abline(h=1, lty=2)   # curve(dunif(x,0,1), add=TRUE)
  # Distribución empírica
curve(ecdf(pvalor.ks)(x), type = "s", lwd = 2, main = '',
      ylab = 'Proporción de rechazos', xlab = 'Nivel de significación')
abline(a=0, b=1, lty=2)   # curve(punif(x, 0, 1), add = TRUE)
# Test de Lilliefors
  # Histograma
hist(pvalor.lil, freq=FALSE, main = "")
abline(h=1, lty=2)   # curve(dunif(x,0,1), add=TRUE)
  # Distribución empírica
curve(ecdf(pvalor.lil)(x), type = "s", lwd = 2, main = '',
      ylab = 'Proporción de rechazos',  xlab = 'Nivel de significación')
abline(a=0, b=1, lty=2)   # curve(punif(x, 0, 1), add = TRUE)
```

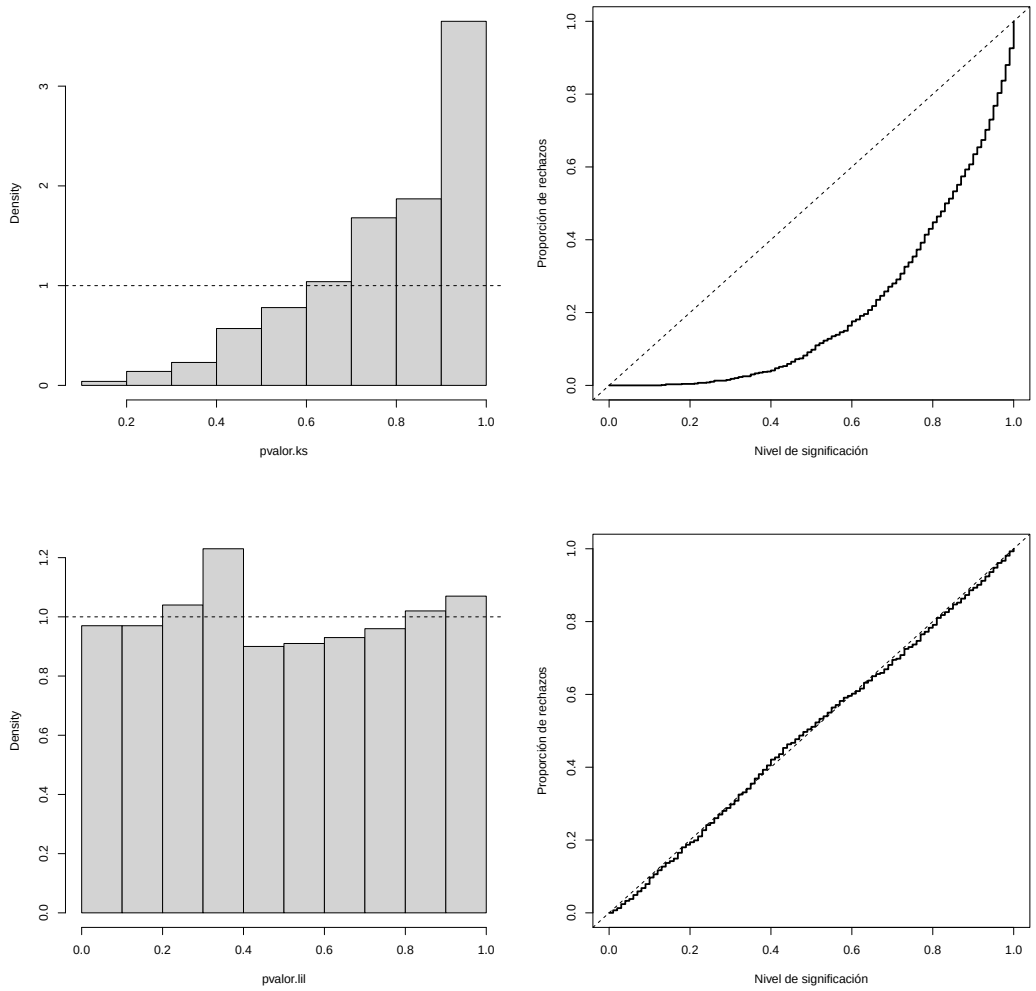


Figura 10.2: Distribución del p-valor (izquierda) y tamaño (proporción de rechazos bajo la hipótesis nula; derecha), aproximados por Monte Carlo, para el contraste de Kolmogorov-Smirnov (arriba) y el de Lilliefors (abajo).

```
par(old.par)
```

En el caso del contraste de Kolmogorov-Smirnov (KS) se observa que el p-valor tiende a tomar valores grandes y por tanto se rechaza la hipótesis nula muchas menos veces de las que se debería.

En el caso de otras distribuciones se puede emplear bootstrap paramétrico para aproximar la distribución del estadístico del contraste. Es importante recordar que el bootstrap debería imitar el procedimiento empleado sobre la muestra, por lo que en este caso también habría que estimar los parámetros en cada remuestra (en caso contrario aproximaríamos la distribución de D_n).

Por ejemplo, la siguiente función implementaría el contraste KS de bondad de ajuste de una variable exponencial aproximando el p-valor mediante bootstrap paramétrico:

```
ks.exp.boot <- function(x, nboot = 10^3) {
  DNAME <- deparse(substitute(x))
  METHOD <- "Kolmogorov-Smirnov Test of pexp by bootstrap"
  n <- length(x)
  RATE <- 1/mean(x)
  ks.exp.stat <- function(x, rate = 1/mean(x)) { # se estima el parámetro
    DMinus <- pexp(sort(x), rate=rate) - (0:(n - 1))/n
    DPlus <- 1/n - DMinus
    Dn = max(c(DMinus, DPlus))
  }
  STATISTIC <- ks.exp.stat(x, rate = RATE)
  names(STATISTIC) <- "Dn"
  # PVAL <- 0
  # for(i in 1:nboot) {
  #   rx <- rexp(n, rate = RATE)
  #   if (STATISTIC <= ks.exp.stat(rx)) PVAL <- PVAL + 1
  # }
  # PVAL <- PVAL/nboot
  # PVAL <- (PVAL + 1)/(nboot + 1) # Alternativa para aproximar el p-valor
  rx <- matrix(rexp(n*nboot, rate = RATE), ncol=n)
  PVAL <- mean(STATISTIC <= apply(rx, 1, ks.exp.stat))
  return(structure(list(statistic = STATISTIC, alternative = "two.sided",
                   p.value = PVAL, method = METHOD, data.name = DNAME),
                   class = "htest"))
}
```

Como ejemplo estudiaremos el caso de contrastar una distribución exponencial considerando 500 pruebas con muestras de tamaño 30 de una $Exp(1)$ y 200 réplicas bootstrap (para disminuir el tiempo de computación).

```
# Valores iniciales
set.seed(1)
nx <- 30
ratex <- 1
nsim <- 500
# Realizar contrastes
pvalor.ks <- numeric(nsim)
pvalor.ks.boot <- numeric(nsim)
for(isim in 1:nsim) {
  rx <- rexp(nx, ratex)
  pvalor.ks[isim] <- ks.test(rx, "pexp", 1/mean(rx))$p.value
  pvalor.ks.boot[isim] <- ks.exp.boot(rx, nboot = 200)$p.value
}
# Generar gráficos
old.par <- par(mfrow=c(2, 2))
# Test de KS
```

```
  # Histograma
hist(pvalor.ks, freq=FALSE, main = "")
abline(h=1, lty=2)   # curve(dunif(x,0,1), add=TRUE)
  # Distribución empírica
curve(ecdf(pvalor.ks)(x), type = "s", lwd = 2, main = '',
      ylab = 'Proporción de rechazos', xlab = 'Nivel de significación')
abline(a=0, b=1, lty=2)   # curve(punif(x, 0, 1), add = TRUE)
# Contraste bootstrap paramétrico
  # Histograma
hist(pvalor.ks.boot, freq=FALSE, main = "")
abline(h=1, lty=2)   # curve(dunif(x,0,1), add=TRUE)
  # Distribución empírica
curve(ecdf(pvalor.ks.boot)(x), type = "s", lwd = 2, main = '',
      ylab = 'Proporción de rechazos',  xlab = 'Nivel de significación')
abline(a=0, b=1, lty=2)   # curve(punif(x, 0, 1), add = TRUE)
```

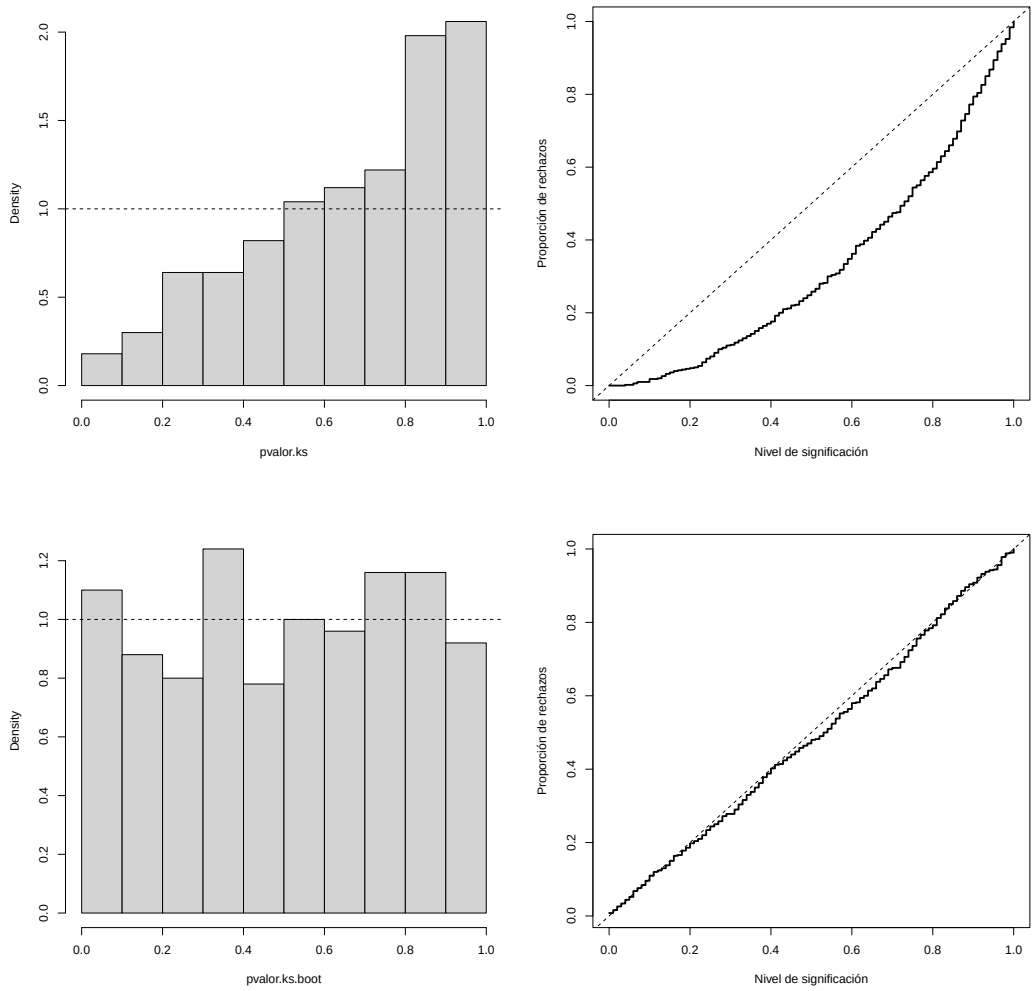


Figura 10.3: Distribución del p-valor (izquierda) y tamaño (proporción de rechazos bajo la hipótesis nula; derecha), aproximados por Monte Carlo, para el contraste de Kolmogorov-Smirnov (arriba) y el correspondiente contraste boostrap paramétrico (abajo).

```
par(old.par)
```

El estadístico de Kolmogorov-Smirnov `Dn = max(c(DMinus, DPlus))` tiene ventajas desde el punto de vista teórico, pero puede no ser muy potente para detectar diferencias entre la distribución bajo la hipótesis nula y la distribución de los datos. La ventaja de la aproximación por simulación es que no estamos atados a resultados teóricos y podemos emplear el estadístico que se considere oportuno (la principal desventaja es el tiempo de computación). Por ejemplo, podríamos pensar en utilizar como estadístico la suma de los errores en valor absoluto del correspondiente gráfico PP, y solo habría que cambiar el estadístico `Dn` en la función `ks.exp.sim` por `Dn = sum(abs( (1:n - 0.5)/n - pexp(sort(x), rate=rate) ))`.

10.3.2 Contrastes de permutaciones

Supongamos que a partir de una muestra $\{(\mathbf{X}_i, Y_i) : i = 1, \dots, n\}$ estamos interesados en contrastar la hipótesis nula de independencia entre $\mathbf{X}$ e Y:

$$H_0 : F_{Y|\mathbf{X}} = F_Y$$

o equivalentemente que $\mathbf{X}$ no influye en la distribución de Y.

En este caso los valores de la respuesta serían intercambiables bajo la hipótesis nula, por lo que podríamos obtener las remuestras manteniendo fijos los valores[3] $\mathbf{X}_i$ y permutando los Y_i. Es decir:

1. Generar Y_i^*, con $i = 1, \dots, n$, mediante muestreo sin reemplazamiento de $\{Y_i : i = 1, \dots, n\}$.
2. Considerar la remuestra bootstrap $\{(\mathbf{X}_i, Y_i^*) : i = 1, \dots, n\}$.

Se pueden realizar contrastes de este tipo con el paquete `boot` estableciendo el parámetro `sim = "permutation"` al llamar a la función `boot()` (el argumento `i` de la función `statistic` contendrá permutaciones del vector de índices). Puede ser también de interés el paquete `coin`, que implementa muchos contrastes de este tipo.

Ejemplo 10.4 (Inferencia sobre el coeficiente de correlación lineal)
Continuando con el ejemplo de las secciones 8.3.3 y 10.2.6 (y de los Ejercicios 8.1, 10.1 y 10.2), estamos interesados en hacer inferencia sobre el coeficiente de correlación lineal ρ empleando el coeficiente de correlación muestral r como estimador. En este caso sin embargo, consideraremos como ejemplo el conjunto de datos `dogs` del paquete `boot`, que contiene observaciones sobre el consumo de oxígeno cardíaco (`mvo`) y la presión ventricular izquierda (`lvp`) de 7 perros domésticos.

```
library(boot)
data('dogs', package = "boot")
# plot(dogs)
cor(dogs$mvo, dogs$lvp)
```

```
## [1] 0.8536946
```

```
# with(dogs, cor(mvo, lvp))
```

Como ya se comentó, para realizar inferencias sobre ρ podemos emplear la función `cor.test()`:

```
cor.test(dogs$mvo, dogs$lvp)
```

```
##
##  Pearson's product-moment correlation
##
## data:  dogs$mvo and dogs$lvp
## t = 3.6655, df = 5, p-value = 0.01451
## alternative hypothesis: true correlation is not equal to 0
## 95 percent confidence interval:
```

[3] Nótese que no se hace ninguna suposición sobre el tipo de covariables, podrían ser categóricas, numéricas o una combinación de ambas.

```
##  0.2818014 0.9780088
## sample estimates:
##       cor
## 0.8536946
```

```
# with(dogs, cor.test(mvo, lvp))
```

Esta función realiza el contraste $H_0 : \rho = 0$ empleando el estadístico:

$$\frac{r\sqrt{n-2}}{\sqrt{1-r^2}} \underset{aprox}{\sim} t_{n-2},$$

bajo la hipótesis nula de que la verdadera correlación es cero. Alternativamente se pueden realizar contrastes unilaterales estableciendo el parámetro `alternative` igual a `"less"` o `"greater"`. Por ejemplo, para contrastar $H_0 : \rho \leq 0$ podríamos emplear:

```
cor.test(dogs$mvo, dogs$lvp, alternative = "greater")
```

```
##
##  Pearson's product-moment correlation
##
## data:  dogs$mvo and dogs$lvp
## t = 3.6655, df = 5, p-value = 0.007255
## alternative hypothesis: true correlation is greater than 0
## 95 percent confidence interval:
##  0.4195889 1.0000000
## sample estimates:
##       cor
## 0.8536946
```

Para realizar el contraste con la función `boot` podríamos emplear el siguiente código:

```
library(boot)

statistic <- function(data, i) cor(data$mvo, data$lvp[i])

set.seed(1)
res.boot <- boot(dogs, statistic, R = 1000, sim = "permutation")
# res.boot
```

Posteriormente emplearíamos las réplicas (almacenadas en `res.boot$t`) y el valor observado del estadístico del contraste (almacenado en `res.boot$t0`) para aproximar el p-valor:

```
hist(res.boot$t, freq = FALSE, main ="")
abline(v = res.boot$t0, lty = 2)
```

Por ejemplo, para el contraste unilateral $H_0 : \rho \leq 0$ (`alternative = "greater"`), obtendríamos:

```
pval.greater <- mean(res.boot$t >= res.boot$t0)
pval.greater
```

```
## [1] 0.009
```

Mientras que para realizar el contraste bilateral $H_0 : \rho = 0$ (`alternative = "two.sided"`), sin asumir que la distribución del estadístico de contraste es simétrica:

```
pval.less <- mean(res.boot$t <= res.boot$t0)
pval <- 2*min(pval.less, pval.greater)
pval
```

```
## [1] 0.018
```

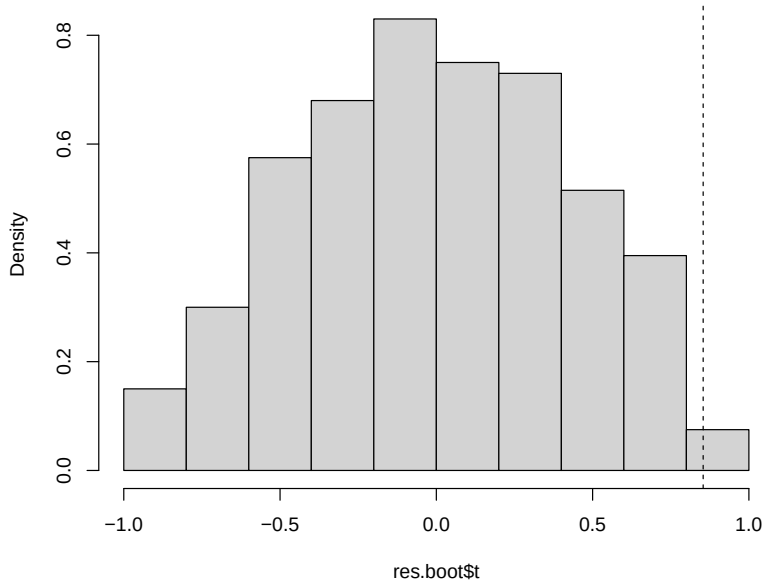


Figura 10.4: Distribución del estadístico del contraste bajo la hipótesis nula aproximada mediante permutación de las observaciones.

10.3.3 Contrastes bootstrap semiparamétricos

Este tipo de aproximación se emplearía en el caso de que la hipótesis nula (o la alternativa) especifique un modelo semiparamétrico, con una componente paramétrica y otra no paramétrica. Típicamente se incluye el error en la componente no paramétrica, y podríamos emplear el bootstrap residual (también denominado semiparamétrico o basado en modelos) descrito en la Sección 9.4.2.

En esta sección nos centraremos en inferencia sobre modelos de regresión lineales (aunque el procedimiento sería análogo en el caso de modelos más generales), empleando como ejemplo el conjunto de datos `Prestige` del paquete `carData`, considerando como variable respuesta `prestige` (puntuación de ocupaciones obtenidas a partir de una encuesta) y como variables explicativas: `income` (media de ingresos en la ocupación) y `education` (media de los años de educación).

```
data(Prestige, package = "carData")
# ?Prestige
```

En la mayoría de los casos nos interesa contrastar un **modelo reducido** frente a un **modelo completo** (que generaliza el modelo reducido). Por ejemplo, en el caso de modelos lineales (estimados por mínimos cuadrados) se dispone del test F para realizar los contrastes de este tipo, que emplea el estadístico:

$$F = \frac{n-q}{q-q_0}\frac{RSS_0 - RSS}{RSS},$$

siendo n el número de observaciones, RSS y q la suma de cuadrados residual y el número de parámetros distintos del modelo completo, y RSS_0 y q_0 los correspondientes al modelo reducido. Este estadístico sigue una distribución $\mathcal{F}_{q-q_0,n-q}$ bajo H_0 y las hipótesis habituales del modelo lineal (ε_i i.i.d. $\mathcal{N}(0,\sigma^2)$).

El contraste de regresión sería un caso particular. Por ejemplo, para contrastar si `income` y `education` influyen linealmente en `prestige` podemos emplear el siguiente código:

```
modelo <- lm(prestige ~ income + education, data = Prestige)
summary(modelo)
```

```
##
```

```
## Call:
## lm(formula = prestige ~ income + education, data = Prestige)
##
## Residuals:
##      Min       1Q   Median       3Q      Max
## -19.4040  -5.3308   0.0154   4.9803  17.6889
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -6.8477787  3.2189771  -2.127   0.0359 *
## income       0.0013612  0.0002242   6.071 2.36e-08 ***
## education    4.1374444  0.3489120  11.858  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.81 on 99 degrees of freedom
## Multiple R-squared:  0.798,  Adjusted R-squared:  0.7939
## F-statistic: 195.6 on 2 and 99 DF,  p-value: < 2.2e-16
```

También podemos obtener el valor observado del estadístico F a partir de los resultados del método `summary.lm()`:

```
res <- summary(modelo)
# names(res)
stat <- res$fstatistic[1]
df <- res$fstatistic[2]
dfr <- res$fstatistic[3]
res$fstatistic
```

```
##    value    numdf    dendf
## 195.5505   2.0000  99.0000
```

o haciendo los cálculos a mano:

```
n <- nrow(Prestige)
q <- 3
q0 <- 1
rss0 <- with(Prestige, sum((prestige - mean(prestige))^2))
rss <- sum(residuals(modelo)^2)
inc.mse <- (rss0 - rss)/(q - q0)  # Incremento en varibilidad explicada
msr <-  rss/(n - q)               # Variabilidad residual
inc.mse/msr
```

```
## [1] 195.5505
```

Desde el punto de vista de comparación de modelos, el modelo reducido bajo la hipótesis nula es:

```
modelo0 <- lm(prestige ~ 1, data = Prestige)
```

y podemos realizar el contraste mediante la función `anova()`

```
anova(modelo0, modelo)
```

```
## Analysis of Variance Table
##
## Model 1: prestige ~ 1
## Model 2: prestige ~ income + education
##   Res.Df     RSS Df Sum of Sq      F    Pr(>F)
## 1    101 29895.4
## 2     99  6038.9  2     23857 195.55 < 2.2e-16 ***
## ---
```

```
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Para aproximar la distribución de este estadístico bajo H_0 podríamos adaptar el bootstrap semiparamétrico[4] descrito en la Sección 9.4.2:

```
library(boot)

pres.dat <- Prestige
# pres.dat$fit0 <- mean(Prestige$prestige)
# pres.dat$fit0 <- predict(modelo0)
pres.dat$res0 <- with(Prestige, prestige - mean(prestige))
# pres.dat$res0 <- residuals(modelo0)

mod.stat <- function(data, i) {
    data$prestige <- mean(data$prestige) + data$res0[i]
    fit <- lm(prestige ~ income + education, data = data)
    summary(fit)$fstatistic[1]
}

set.seed(1)
boot.mod <- boot(pres.dat, mod.stat, R = 1000)
boot.mod
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = pres.dat, statistic = mod.stat, R = 1000)
##
##
## Bootstrap Statistics :
##     original    bias    std. error
## t1* 195.5505 -194.4866    1.096335
```

```
hist(boot.mod$t, freq = FALSE, breaks = "FD", main = "")
curve(pf(x, df, dfr, lower.tail = FALSE), lty = 2, add = TRUE)
```

```
# pval <- mean(boot.mod$t >= boot.mod$t0)
pval <- mean(boot.mod$t >= stat)
pval
```

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

Procediendo de esta forma sin embargo estaríamos sobreestimando la variabilidad del error cuando la hipótesis nula es falsa (la variabilidad no explicada por la tendencia es asumida por el error), lo que disminuirá la potencia del contraste. Para mejorar la potencia, siguiendo la idea propuesta por González-Manteiga y Cao (1993), se pueden remuestrear los residuos del modelo completo. De esta forma reproduciríamos la variabilidad del error de forma consistente tanto bajo la hipótesis alternativa como bajo la nula.

```
old.par <- par(mfrow=c(1,2))
# Variabilidad residual con el modelo reducido
hist(residuals(modelo0), xlim = c(-50, 50), main = "")
# Variabilidad residual con el modelo completo
hist(residuals(modelo), xlim = c(-50, 50), main = "")
```

```
par(old.par)
```

[4] En este caso también podríamos emplear un contraste de permutaciones.

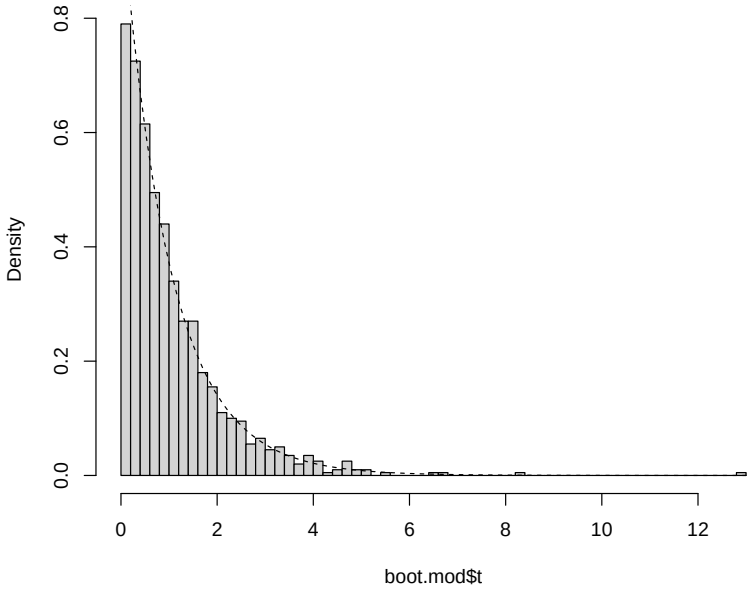


Figura 10.5: Distribución del estadístico del contraste (bajo la hipótesis nula) aproximada mediante bootstrap semiparamétrico.

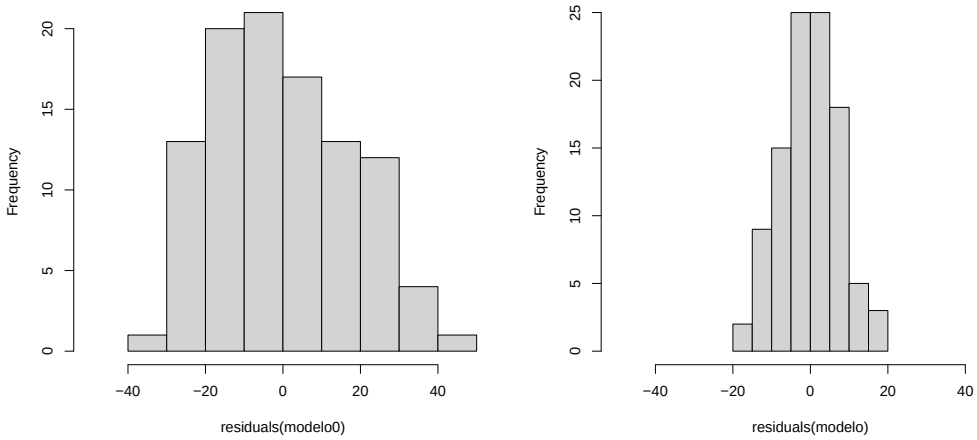


Figura 10.6: Variabilidad residual con el modelo reducido (izquierda) y con el modelo completo (derecha).

Adicionalmente, como se mostró en la Sección 9.4.2, se puede emplear la modificación propuesta en Davison y Hinkley (1997, Alg. 6.3, p. 271) y remuestrear los residuos reescalados y centrados.

```
pres.dat <- Prestige
# pres.dat$fit0 <- mean(Prestige$prestige)
# pres.dat$fit0 <- predict(modelo0)
# pres.dat$res <- residuals(modelo)
pres.dat$sres <- residuals(modelo)/sqrt(1 - hatvalues(modelo))
pres.dat$sres <- pres.dat$sres - mean(pres.dat$sres)
```

```
mod.stat <- function(data, i) {
    # data$prestige <- mean(data$prestige) + data$res[i]
    data$prestige <- mean(data$prestige) + data$sres[i]
    fit <- lm(prestige ~ income + education, data = data)
    summary(fit)$fstatistic[1]
}

set.seed(1)
boot.mod <- boot(pres.dat, mod.stat, R = 1000)
boot.mod
```

```
##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = pres.dat, statistic = mod.stat, R = 1000)
##
##
## Bootstrap Statistics :
##       original   bias    std. error
## t1* 0.01164396 1.029746    1.029715
```

En la aproximación del p-valor hay que tener en cuenta que al modificar los residuos `boot.mod$t0` no va a coincidir con el valor observado del estadístico, almacenado en `stat` (por tanto habría que ignorar `original` y `bias` en `Bootstrap Statistics`; la función `Boot()` del paquete `car` corrige este problema).

```
hist(boot.mod$t, freq = FALSE, breaks = "FD", main = "")
curve(pf(x, df, dfr, lower.tail = FALSE), lty = 2, add = TRUE)
```

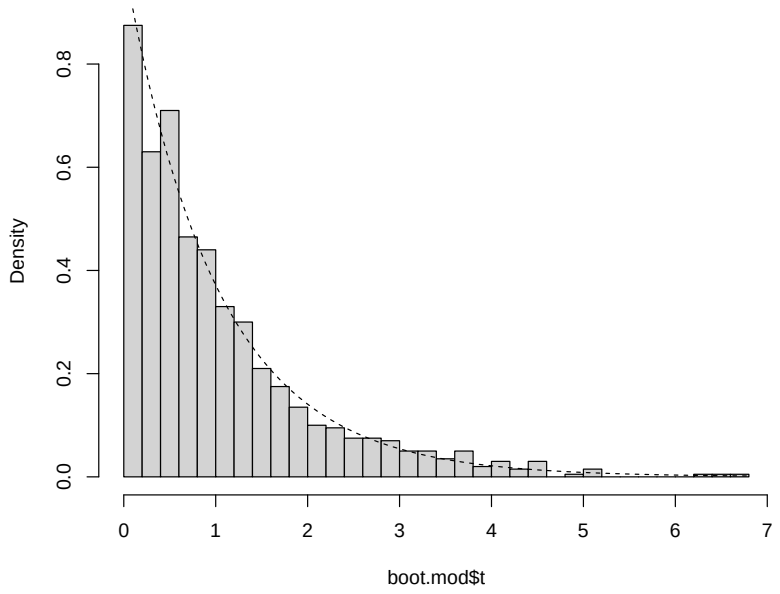


Figura 10.7: Distribución del estadístico del contraste (bajo la hipótesis nula) aproximada mediante bootstrap semiparamétrico.

```
pval <- mean(boot.mod$t >= stat)
pval
```

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

En el caso de modelos no lineales (o otros tipos de modelos lineales) puede ser complicado aproximar los grados de libertad para el cálculo del estadístico F, pero si empleamos bootstrap, vamos a obtener los mismos resultados considerando como estadístico:

$$\tilde{F} = \frac{RSS_0 - RSS}{RSS},$$

que se puede interpretar también como una medida del incremento en la variabilidad residual al considerar el modelo reducido (ya que únicamente difieren en una constante). En este caso también se suelen emplear los residuos sin reescalar, ya que también puede ser difícil encontrar la transformación adecuada.

Ejercicio 10.3

Al estudiar el efecto de las variables explicativas en el modelo anterior, podríamos pensar que no es adecuado asumir un efecto lineal de alguna de las variables explicativas. Por ejemplo, si generamos los gráficos parciales de residuos obtendríamos:

```
car::crPlots(modelo)
```

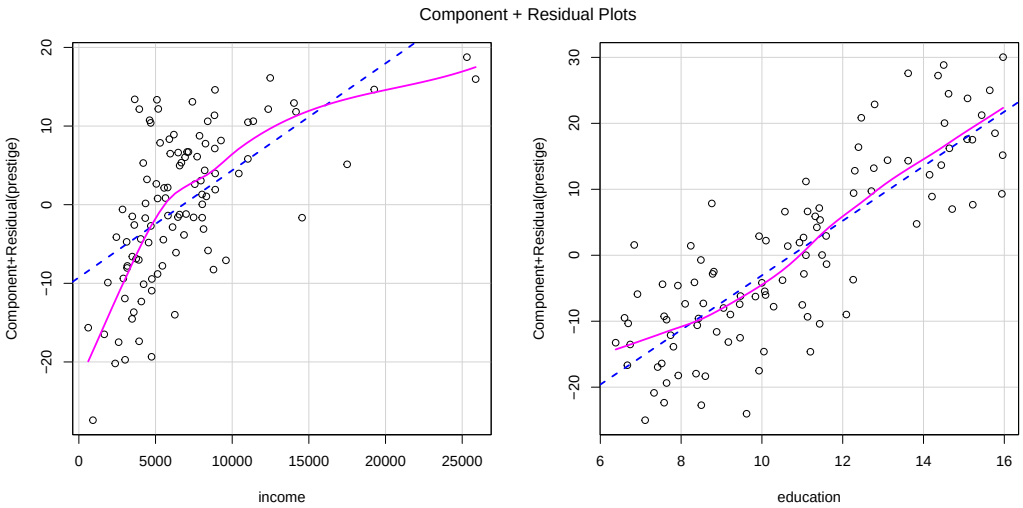


Figura 10.8: Efecto de las variables explicativas en el modelo (gráficos de residuos).

En este caso podría ser razonable considerar un efecto cuadrático de la variable `income`[5]

```
modelo <- lm(prestige ~ income + I(income^2) + education, data = Prestige)
summary(modelo)
```

```
##
## Call:
## lm(formula = prestige ~ income + I(income^2) + education, data = Prestige)
##
## Residuals:
##     Min      1Q  Median      3Q     Max
```

[5]Para ajustar un modelo polinómico puede ser recomendable, especialmente si el grado del polinomio es alto, emplear la función `poly()` ya que utiliza polinomios ortogonales. En el caso cuadrático, al emplear `y ~ x + I(x^2)` estaremos considerando $1, x, x^2$, mientras que `y ~ poly(x, 2)` considerará polinomios de Legendre de la forma $1, x, \frac{1}{2}(3x^2-1)$. En este caso concreto, obtendríamos una parametrización equivalente empleando `modelo <- lm(prestige ~ poly(income, 2) + education, data = Prestige)`.

```
## -15.732  -4.900  -0.057   4.598  18.459
##
## Coefficients:
##               Estimate Std. Error t value Pr(>|t|)
## (Intercept) -1.135e+01  3.272e+00  -3.470 0.000775 ***
## income       3.294e-03  5.669e-04   5.810 7.79e-08 ***
## I(income^2) -7.967e-08  2.169e-08  -3.673 0.000390 ***
## education    3.809e+00  3.407e-01  11.179  < 2e-16 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
##
## Residual standard error: 7.36 on 98 degrees of freedom
## Multiple R-squared:  0.8224, Adjusted R-squared:  0.817
## F-statistic: 151.3 on 3 and 98 DF,  p-value: < 2.2e-16
```

Para comparar el ajuste de este modelo respecto al del anterior, podemos realizar un contraste empleando la función `anova()`:

```r
modelo0 <- lm(prestige ~ income + education, data = Prestige)
anova(modelo0, modelo)
```

```
## Analysis of Variance Table
##
## Model 1: prestige ~ income + education
## Model 2: prestige ~ income + I(income^2) + education
##   Res.Df    RSS Df Sum of Sq      F    Pr(>F)
## 1     99 6038.9
## 2     98 5308.0  1     730.8 13.492 0.0003904 ***
## ---
## Signif. codes:  0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
```

Contrastar si el efecto de `income` es lineal mediante bootstrap residual, empleando como estadístico el incremento en la variabilidad residual con el modelo reducido y remuestreando los residuos del modelo completo (sin reescalar). Aproximar el nivel crítico del contraste y el valor que tendría que superar el estadístico para rechazar H_0 con un nivel de significación $\alpha = 0.05$.

Apéndice A

Bondad de Ajuste y Aleatoriedad

En los métodos clásicos de inferencia estadística es habitual asumir que los valores observados $X_1, \dots, X_n$ (o los errores de un modelo) constituyen una muestra aleatoria simple de una variable aleatoria X. Se están asumiendo por tanto dos hipótesis estructurales: la independencia (aleatoriedad) y la homogeneidad (misma distribución) de las observaciones (o de los errores). Adicionalmente, en inferencia paramétrica se supone que la distribución se ajusta a un modelo paramétrico específico $F_\theta(x)$, siendo θ un parámetro que normalmente es desconocido.

Uno de los objetivos de la inferencia no paramétrica es desarrollar herramientas que permitan verificar el grado de cumplimiento de las hipótesis anteriores[1]. Los procedimientos habituales incluyen métodos descriptivos (principalmente gráficos), contrastes de bondad de ajuste (también de homogeneidad o de datos atípicos) y contrastes de aleatoriedad.

En este apéndice se describen brevemente algunos de los métodos clásicos, principalmente con la idea de que pueden ser de utilidad para evaluar resultados de simulación y para la construcción de modelos del sistema real (e.g. para modelar variables que se tratarán como entradas del modelo general). Se empleará principalmente el enfoque de la estadística no paramétrica, aunque también se mostrarán algunas pequeñas diferencias entre su uso en inferencia y en simulación.

Los métodos genéricos no son muy adecuados para evaluar generadores aleatorios (e.g. L'Ecuyer y Simard, 2007). La recomendación sería emplear baterías de contrastes recientes, como las descritas en la Sección 2.3.2. No obstante, en la última sección se describirán, únicamente con fines ilustrativos, algunos de los primeros métodos diseñados específicamente para generadores aleatorios.

A.1 Métodos de bondad de ajuste

A partir de $X_1, \dots, X_n$ m.a.s. de X con función de distribución F, interesa realizar un contraste de la forma:

$$\begin{cases} H_0 : F = F_0 \\ H_1 : F \neq F_0 \end{cases}$$

En este caso interesará distinguir principalmente entre hipótesis nulas simples (especifican un único modelo) y compuestas (especifican un conjunto o familia de modelos). Por ejemplo:

H_0 simple	H_0 compuesta
$\begin{cases} H_0 : F = \mathcal{N}(0,1) \\ H_1 : F \neq \mathcal{N}(0,1) \end{cases}$	$\begin{cases} H_0 : F = \mathcal{N}(\mu,\sigma^2) \\ H_1 : F \neq \mathcal{N}(\mu,\sigma^2) \end{cases}$

[1] El otro objetivo de la inferencia estadística no paramétrica es desarrollar procedimientos alternativos (métodos de distribución libre) que sean válidos cuando no se verifica alguna de las hipótesis estructurales.

Entre los métodos gráficos habituales estarían: histograma, gráfico de la densidad suavizada, gráfico de tallo y hojas, gráfico de la distribución empírica (o versión suavizada) y gráficos P-P o Q-Q.

Entre los métodos de contrastes de hipótesis generales ($H_0 : F = F_0$) destacarían las pruebas: Chi-cuadrado de Pearson, Kolmogorov-Smirnov, Cramer-von Mises o Anderson-Darling. Además de los específicos de normalidad ($H_0 : F = \mathcal{N}(\mu, \sigma^2)$): Kolmogorov-Smirnov-Lilliefors, Shapiro-Wilks y los de asimetría y apuntamiento.

A.1.1 Histograma

Se agrupan los datos en intervalos $I_k = [L_{k-1}, L_k)$ con $k = 1, \dots, K$ y a cada intervalo se le asocia un valor (altura de la barra) igual a la frecuencia absoluta de ese intervalo $n_k = \sum_{i=1}^{n} \mathbf{1}\left(X_i \in [L_{k-1}, L_k)\right)$, si la longitud de los intervalos es constante, o proporcional a dicha frecuencia (de forma que el área coincida con la frecuencia relativa y pueda ser comparado con una función de densidad):

$$\hat{f}_n(x) = \frac{n_i}{n\left(L_k - L_{k-1}\right)}$$

Como ya se ha visto anteriormente, en R podemos generar este gráfico con la función `hist()` del paquete base. Algunos de los principales parámetros (con los valores por defecto) son los siguientes:

```
hist(x, breaks = "Sturges", freq = NULL, plot = TRUE, ...)
```

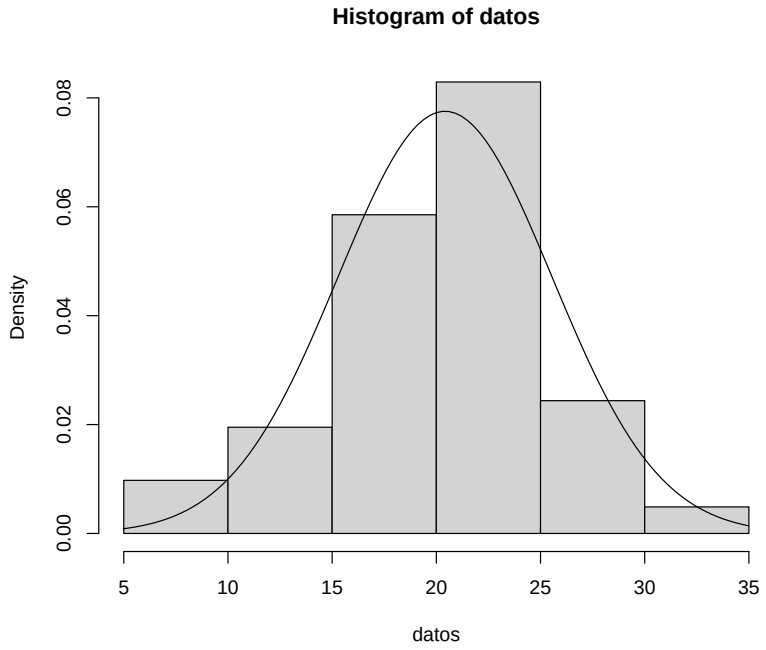
- `breaks`: puede ser un valor numérico con el número de puntos de discretización, un vector con los puntos de discretización, una cadena de texto que los determine (otras opciones son `"Scott"` y `"FD"`; en este caso llamará internamente a la función `nclass.xxx()` donde `xxx` se corresponde con la cadena de texto), o incluso una función personalizada que devuelva el número o el vector de puntos de discretización.

- `freq`: lógico (`TRUE` por defecto si los puntos de discretización son equidistantes), determina si en el gráfico se representan frecuencias o "densidades".

- `plot`: lógico, se puede establecer a `FALSE` si no queremos generar el gráfico y solo nos interesan el objeto con los resultados (que devuelve de forma "invisible", por ejemplo para discretizar los valores en intervalos).

Ejemplo:

```
datos <- c(22.56,22.33,24.58,23.14,19.03,26.76,18.33,23.10,
  21.53,9.06,16.75,23.29,22.14,16.28,18.89,27.48,10.44,
  26.86,27.27,18.74,19.88,15.76,30.77,21.16,24.26,22.90,
  27.14,18.02,21.53,24.99,19.81,11.88,24.01,22.11,21.91,
  14.35,11.14,9.93,20.22,17.73,19.05)
hist(datos, freq = FALSE)
curve(dnorm(x, mean(datos), sd(datos)), add = TRUE)
```



Si el número de valores es muy grande (por ejemplo en el caso de secuencias aleatorias), nos puede interesar establecer la opción `breaks = "FD"` para aumentar el número de intervalos de discretización. En cualquier caso, como se muestra en la Figura A.1, la convergencia del histograma a la densidad teórica se podría considerar bastante lenta. Alternativamente se podría considerar una estimación suave de la densidad, por ejemplo empleando la estimación tipo núcleo implementada en la función `density()`.

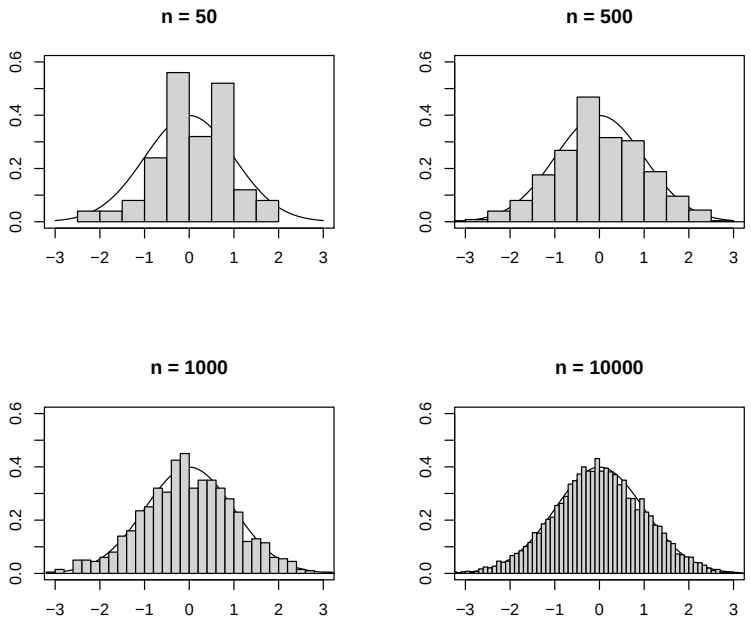


Figura A.1: Convergencia del histograma a la densidad teórica.

A.1.2 Función de distribución empírica

La función de distribución empírica $F_n(x) = \frac{1}{n}\sum_{i=1}^{n} \mathbf{1}(X_i \leq x)$ asigna a cada número real x la frecuencia relativa de observaciones menores o iguales que x. Para obtener las frecuencias relativas acumuladas, se ordena la muestra $X_{(1)} \leq X_{(2)} \leq \cdots \leq X_{(n)}$ y:

$$F_n(x) = \begin{cases} 0 & \text{si} \quad x < X_{(1)} \\ \frac{i}{n} & \text{si} \quad X_{(i)} \leq x < X_{(i+1)} \\ 1 & \text{si} \quad X_{(n)} \leq x \end{cases}$$

Ejemplo:

```
fn <- ecdf(datos)
curve(ecdf(datos)(x), xlim = extendrange(datos), type = 's',
      ylab = 'distribution function', lwd = 2)
curve(pnorm(x, mean(datos), sd(datos)), add = TRUE)
```

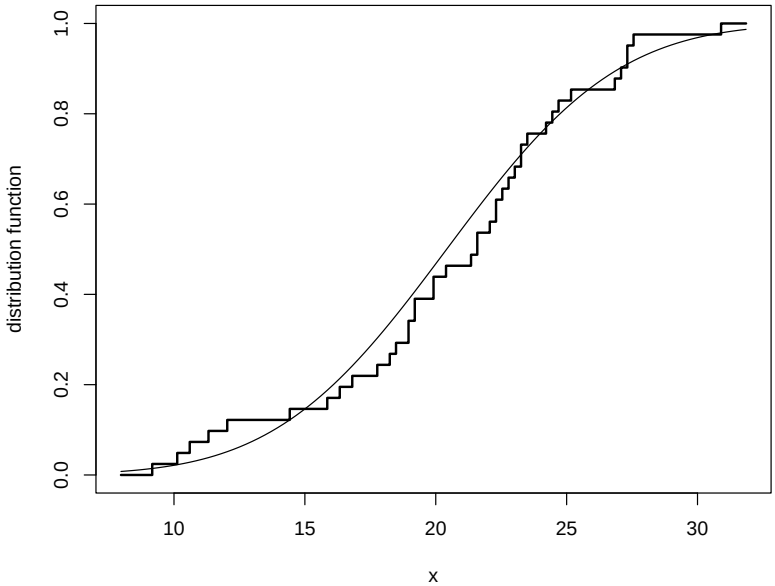


Figura A.2: Comparación de la distribución empírica de los datos de ejemplo con la función de distribución de la aproximación normal.

La función de distribución empírica se corresponde con una variable aleatoria discreta que toma los valores $X_1, \ldots, X_n$ todos ellos con probabilidad $\frac{1}{n}$. Suponiendo que $\mathbf{X} = (X_1, \ldots, X_n)$ es una muestra aleatoria de una población con distribución F:

$$\begin{aligned} nF_n(x) &= \sum_{i=1}^{n} \mathbf{1}\{X_i \leq x\} \sim \mathcal{B}(n, F(x)), \\ E(nF_n(x)) &= nF(x) \implies E(F_n(x)) = F(x), \\ Var(nF_n(x)) &= nF(x)(1 - F(x)) \\ &\implies Var(F_n(x)) = \frac{F(x)(1 - F(x))}{n} \end{aligned}$$

A.1.3 Gráficos P-P y Q-Q

El gráfico de probabilidad (o de probabilidad-probabilidad) es el gráfico de dispersión de:

$$\{(F_0(x_i), F_n(x_i)) : i = 1, \cdots, n\}$$

siendo F_n la función de distribución empírica y F_0 la función de distribución bajo H_0 (con la que desea comparar, si la hipótesis nula es simple) o una estimación bajo H_0 (si la hipótesis nula es compuesta; e.g. si $H_0 : F = \mathcal{N}(\mu, \sigma^2)$, $\hat{F}_0$ función de distribución de $\mathcal{N}(\hat{\mu}, \hat{\sigma}^2)$). Si H_0 es cierta, la nube de puntos estará en torno a la recta $y = x$ (probabilidades observadas próximas a las esperadas bajo H_0).

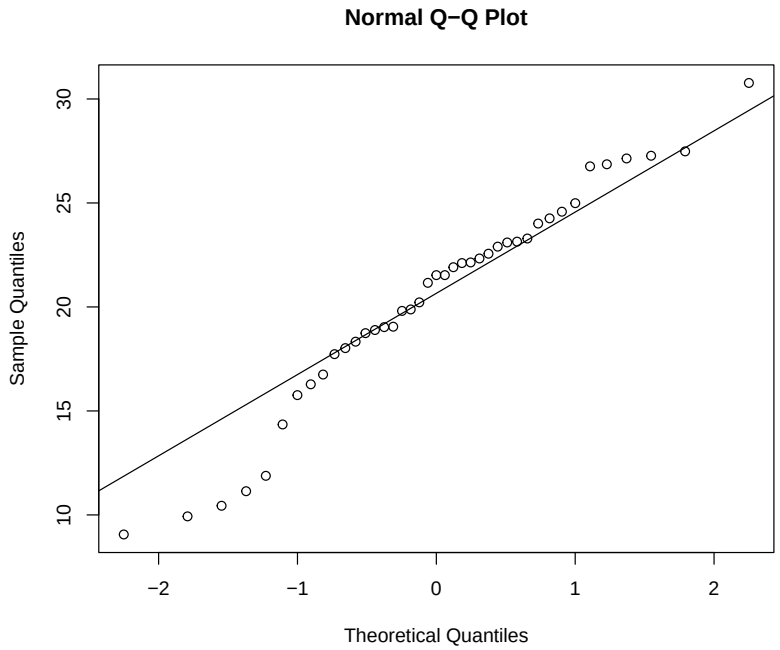
El gráfico Q-Q (cuantil-cuantil) es equivalente al anterior pero en la escala de la variable:

$$\left\{\left(q_i, x_{(i)}\right) : i = 1, \cdots, n\right\}$$

siendo $x_{(i)}$ los cuantiles observados y $q_i = F_0^{-1}(p_i)$ los esperados[2] bajo H_0.

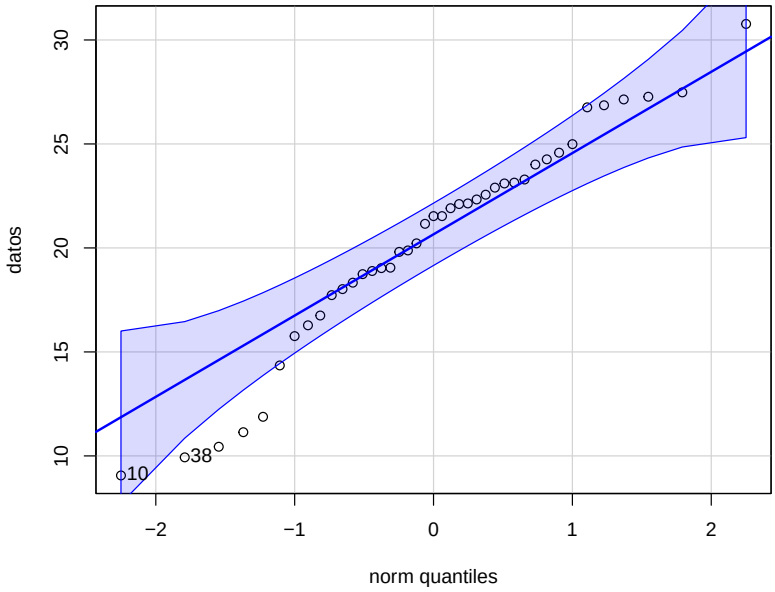
Ejemplo:

```
qqnorm(datos)
qqline(datos)
```




```
require(car)
qqPlot(datos, "norm")
```

[2]Típicamente $\left\{p_i = \frac{(i-0.5)}{n} : i = 1, \cdots, n\right\}$.




```
## [1] 10 38
```

A.1.4 Contraste chi-cuadrado de Pearson

Se trata de un contraste de bondad de ajuste:

$$\begin{cases} H_0 : F = F_0 \\ H_1 : F \neq F_0 \end{cases}$$

desarrollado inicialmente para variables categóricas. En el caso general, podemos pensar que los datos están agrupados en k clases: $C_1, \cdots, C_k$. Por ejemplo, si la variable es categórica o discreta, cada clase se puede corresponder con una modalidad. Si la variable es continua habrá que categorizarla en intervalos.

Si la hipótesis nula es simple, cada clase tendrá asociada una probabilidad $p_i = P(X \in C_i)$ bajo H_0 . Si por el contrario es compuesta, se trabajará con una estimación de dicha probabilidad (y habrá que correguir la distribución aproximada del estadístico del contraste).

Clases	Discreta	Continua	H_0 simple	H_0 compuesta
C_1	x_1	$[L_0, L_1)$	p_1	$\hat{p}_1$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_k	x_k	$[L_{k-1}, L_k)$	p_k	$\hat{p}_k$
			$\sum_i p_i = 1$	$\sum_i \hat{p}_i = 1$

Se realizará un contraste equivalente:

$$\begin{cases} H_0 : \text{Las probabilidades son correctas} \\ H_1 : \text{Las probabilidades no son correctas} \end{cases}$$

Si H_0 es cierta, la frecuencia relativa f_i de la clase C_i es una aproximación de la probabilidad teórica, $f_i \approx p_i$. Equivalentemente, las frecuencias observadas $n_i = n \cdot f_i$ deberían ser próximas a las esperadas $e_i = n \cdot p_i$ bajo H_0, sugiriendo el estadístico del contraste (Pearson, 1900):

$$\chi^2 = \sum_{i=1}^{k} \frac{(n_i - e_i)^2}{e_i} \underset{aprox.}{\sim} \chi^2_{k-r-1}, \text{ si } H_0 \text{ cierta}$$

siendo k el número de clases y r el número de parámetros estimados (para aproximar las probabilidades bajo H_0).

Clases	n_i observadas	p_i bajo H_0	e_i bajo H_0	$\frac{(n_i-e_i)^2}{e_i}$
C_1	n_1	p_1	e_1	$\frac{(n_1-e_1)^2}{e_1}$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
C_k	n_k	p_k	e_k	$\frac{(n_k-e_k)^2}{e_k}$
Total	$\sum_i n_i = n$	$\sum_i p_i = 1$	$\sum_i e_i = n$	$\chi^2 = \sum_{i=1}^k \frac{(n_i-e_i)^2}{e_i}$

Cuando H_0 es cierta el estadístico tiende a tomar valores pequeños y grandes cuando es falsa. Por tanto se rechaza H_0, para un nivel de significación α, si:

$$\sum_{i=1}^{k} \frac{(n_i - e_i)^2}{e_i} \geq \chi^2_{k-r-1,1-\alpha}$$

Si realizamos el contraste a partir del p-valor o nivel crítico:

$$p = P\left(\chi^2_{k-r-1} \geq \sum_{i=1}^{k} \frac{(n_i - e_i)^2}{e_i}\right)$$

rechazaremos H_0 si $p \leq \alpha$ (y cuanto menor sea se rechazará con mayor seguridad) y aceptaremos H_0 si $p > \alpha$ (con mayor seguridad cuanto mayor sea).

Este método está implementado en la función `chisq.test()` para el caso discreto (no corrige los grados de libertad). Ejemplo:

```
x <- trunc(5 * runif(100))
chisq.test(table(x))            # NOT 'chisq.test(x)'!
```

```
##
##  Chi-squared test for given probabilities
##
## data:  table(x)
## X-squared = 9.2, df = 4, p-value = 0.05629
```

La distribución exacta del estadístico del contraste es discreta (se podría aproximar por simulación, por ejemplo empleando los parámetros `simulate.p.value = TRUE` y `B = 2000` de la función `chisq.test()`; ver también el Ejemplo 6.10 de la Sección 6.7.2 para el caso del contraste chi-cuadrado de independencia). Para que la aproximación continua χ^2 sea válida:

- El tamaño muestral debe ser suficientemente grande (p.e. $n > 30$).
- La muestra debe ser una muestra aleatoria simple.
- Los parámetros deben estimarse (si es necesario) por máxima verosimilitud.
- Las frecuencias esperadas $e_i = n \cdot p_i$ deberían ser todas ≥ 5 (realmente esta es una restricción conservadora, la aproximación puede ser adecuada si no hay frecuencias esperadas inferiores a 1 y menos de un 20% inferiores a 5).

Si la frecuencia esperada de alguna clase es < 5, se suele agrupar con otra clase (o con varias si no fuese suficiente con una) para obtener una frecuencia esperada ≥ 5:

- Cuando la variable es nominal (no hay una ordenación lógica) se suele agrupar con la(s) que tiene(n) menor valor de e_i.
- Si la variable es ordinal (o numérica) debe juntarse la que causó el problema con una de las adyacentes.

Si la variable de interés es continua, una forma de garantizar que $e_i \geq 5$ consiste en tomar un número de intervalos $k \leq \lfloor n/5 \rfloor$ y de forma que sean equiprobables $p_i = 1/k$, considerando los puntos críticos $x_{i/k}$ de la distribución bajo H_0.

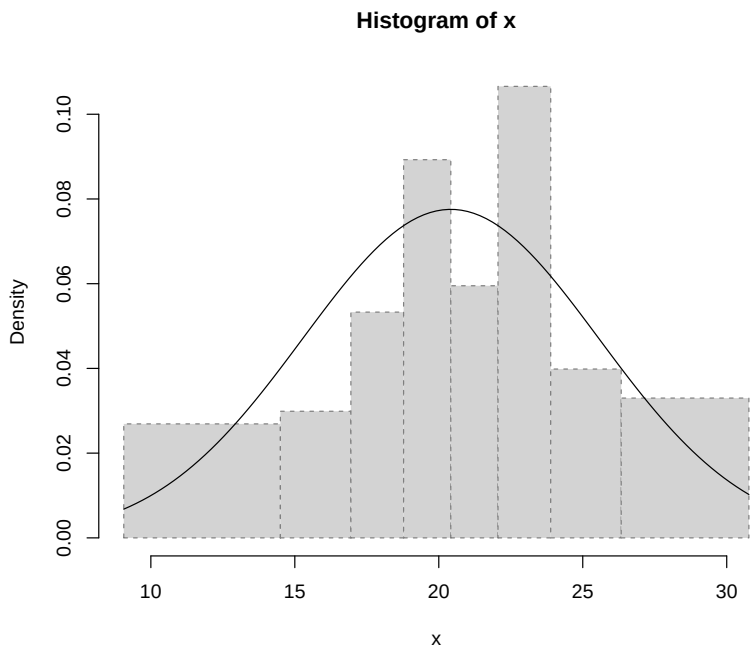
Por ejemplo, se podría emplear la función `simres::chisq.cont.test()` (fichero *test.R*), que imita a las incluidas en R:

```
simres::chisq.cont.test
```

```
## function(x, distribution = "norm", nclass = floor(length(x)/5),
##                              output = TRUE, nestpar = 0, ...) {
##   # Función distribución
##   q.distrib <- eval(parse(text = paste("q", distribution, sep = "")))
##   # Puntos de corte
##   q <- q.distrib((1:(nclass - 1))/nclass, ...)
##   tol <- sqrt(.Machine$double.eps)
##   xbreaks <- c(min(x) - tol, q, max(x) + tol)
##   # Gráficos y frecuencias
##   if (output) {
##     xhist <- hist(x, breaks = xbreaks, freq = FALSE,
##                   lty = 2, border = "grey50")
##     # Función densidad
##     d.distrib <- eval(parse(text = paste("d", distribution, sep = "")))
##     curve(d.distrib(x, ...), add = TRUE)
##   } else {
##     xhist <- hist(x, breaks = xbreaks, plot = FALSE)
##   }
##   # Cálculo estadístico y p-valor
##   O <- xhist$counts  # Equivalente a table(cut(x, xbreaks)) pero más eficiente
##   E <- length(x)/nclass
##   DNAME <- deparse(substitute(x))
##   METHOD <- "Pearson's Chi-squared test"
##   STATISTIC <- sum((O - E)^2/E)
##   names(STATISTIC) <- "X-squared"
##   PARAMETER <- nclass - nestpar - 1
##   names(PARAMETER) <- "df"
##   PVAL <- pchisq(STATISTIC, PARAMETER, lower.tail = FALSE)
##   # Preparar resultados
##   classes <- format(xbreaks)
##   classes <- paste("(", classes[-(nclass + 1)], ",", classes[-1], "]",
##                    sep = "")
##   RESULTS <- list(classes = classes, observed = O, expected = E,
##                   residuals = (O - E)/sqrt(E))
##   if (output) {
##     cat("\nPearson's Chi-squared test table\n")
##     print(as.data.frame(RESULTS))
##   }
##   if (any(E < 5))
##     warning("Chi-squared approximation may be incorrect")
##   structure(c(list(statistic = STATISTIC, parameter = PARAMETER, p.value = PVAL,
##                    method = METHOD, data.name = DNAME), RESULTS), class = "htest")
## }
## <bytecode: 0x00000000376d7688>
## <environment: namespace:simres>
```

Continuando con el ejemplo anterior, podríamos contrastar normalidad mediante:

```
library(simres)
chisq.cont.test(datos, distribution = "norm", nestpar = 2, mean=mean(datos), sd=sd(datos))
```




```
##
## Pearson's Chi-squared test table
##                classes observed expected  residuals
## 1 ( 9.06000,14.49908]        6    5.125  0.3865103
## 2 (14.49908,16.94725]        3    5.125 -0.9386680
## 3 (16.94725,18.77800]        4    5.125 -0.4969419
## 4 (18.77800,20.41732]        6    5.125  0.3865103
## 5 (20.41732,22.05663]        4    5.125 -0.4969419
## 6 (22.05663,23.88739]        8    5.125  1.2699625
## 7 (23.88739,26.33556]        4    5.125 -0.4969419
## 8 (26.33556,30.77000]        6    5.125  0.3865103
```

```
##
##  Pearson's Chi-squared test
##
## data:  datos
## X-squared = 3.6829, df = 5, p-value = 0.5959
```

A.1.5 Contraste de Kolmogorov-Smirnov

Se trata de un contraste de bondad de ajuste diseñado para distribuciones continuas (similar a la prueba de Cramer-von Mises o a la de Anderson-Darling, implementadas en el paquete `goftest` de R, que son en principio mejores). Se basa en comparar la función de distribución F_0 bajo H_0 con la función de distribución empírica F_n:

$$\begin{aligned} D_n &= \sup_x |F_n(x) - F_0(x)|, \\ &= \max_{1\leq i\leq n} \left\{ |F_n(X_{(i)}) - F_0(X_{(i)})|, |F_n(X_{(i-1)}) - F_0(X_{(i)})| \right\} \end{aligned}$$

Teniendo en cuenta que $F_n\left(X_{(i)}\right) = \frac{i}{n}$:

$$\begin{aligned} D_n &= \max_{1\leq i\leq n}\left\{\frac{i}{n} - F_0(X_{(i)}), F_0(X_{(i)}) - \frac{i-1}{n}\right\} \\ &= \max_{1\leq i\leq n}\left\{D^+_{n,i}, D^-_{n,i}\right\} \end{aligned}$$

Si H_0 es simple y F_0 es continua, la distribución del estadístico D_n bajo H_0 no depende F_0 (es de distribución libre). Esta distribución está tabulada (para tamaños muestrales grandes se utiliza la aproximación asintótica). Se rechaza H_0 si el valor observado d del estadístico es significativamente grande:

$$p = P\left(D_n \geq d\right) \leq \alpha.$$

Este método está implementado en la función `ks.test()` del paquete base de R:

```
ks.test(x, y, ...)
```

donde `x` es un vector que contiene los datos, `y` es una función de distribución (o una cadena de texto que la especifica; también puede ser otro vector de datos para el contraste de dos muestras) y `...` representa los parámetros de la distribución.

Continuando con el ejemplo anterior, para contrastar $H_0 : F = \mathcal{N}(20, 5^2)$ podríamos emplear:

```
ks.test(datos, pnorm, mean = 20, sd = 5) # One-sample
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  datos
## D = 0.13239, p-value = 0.4688
## alternative hypothesis: two-sided
```

Si H_0 es compuesta, el procedimiento habitual es estimar los parámetros desconocidos por máxima verosimilitud y emplear $\hat{F}_0$ en lugar de F_0. Sin embargo, al proceder de esta forma es de esperar que $\hat{F}_0$ se aproxime más que F_0 a la distribución empírica, por lo que los cuantiles de la distribución de D_n pueden ser demasiado conservativos (los p-valores tenderán a ser mayores de lo que deberían) y se tenderá a aceptar la hipótesis nula (puede ser preferible aproximar el p-valor mediante simulación; como se muestra en el Ejercicio 7.6 de la Sección 7.4.3).

Para evitar este problema, en el caso de contrastar normalidad se desarrolló el test de Lilliefors, implementado en la función `lillie.test()` del paquete `nortest` (también hay versiones en este paquete para los métodos de Cramer-von Mises y Anderson-Darling).

Por ejemplo:

```
ks.test(datos, pnorm, mean(datos), sd(datos)) # One-sample Kolmogorov-Smirnov test
```

```
##
##  One-sample Kolmogorov-Smirnov test
##
## data:  datos
## D = 0.097809, p-value = 0.8277
## alternative hypothesis: two-sided
```

```
library(nortest)
lillie.test(datos)
```

```
##
##  Lilliefors (Kolmogorov-Smirnov) normality test
##
## data:  datos
## D = 0.097809, p-value = 0.4162
```

A.2 Diagnosis de la independencia

Los métodos "clásicos" de inferencia estadística se basan en suponer que las observaciones $X_1, \ldots, X_n$ son una muestra aleatoria simple (m.a.s.) de X. Por tanto suponen que las observaciones son independientes (o los errores, en el caso de un modelo de regresión).

- La ausencia de aleatoriedad es difícil de corregir y puede influir notablemente en el análisis estadístico.
- Si existe dependencia entre las observaciones muestrales (e.g. el conocimiento de X_i proporciona información sobre los valores de X_{i+1}, X_{i+2}, ...), los métodos "clásicos" no serán en principio adecuados (pueden conducir a conclusiones erróneas).
  - Esto es debido principalmente a que introduce un sesgo en los estimadores de las varianzas (diseñados asumiendo independencia).
  - Los correspondientes intervalos de confianza y contrastes de hipótesis tendrán una confianza o una potencia distinta de la que deberían (aunque las estimaciones de los parámetros pueden no verse muy afectadas).

Si X_1 e X_2 son independientes ($Cov(X_1, X_2) = 0$):

$$Var(X_1 + X_2) = Var(X_1) + Var(X_2)$$

En el caso general (dependencia):

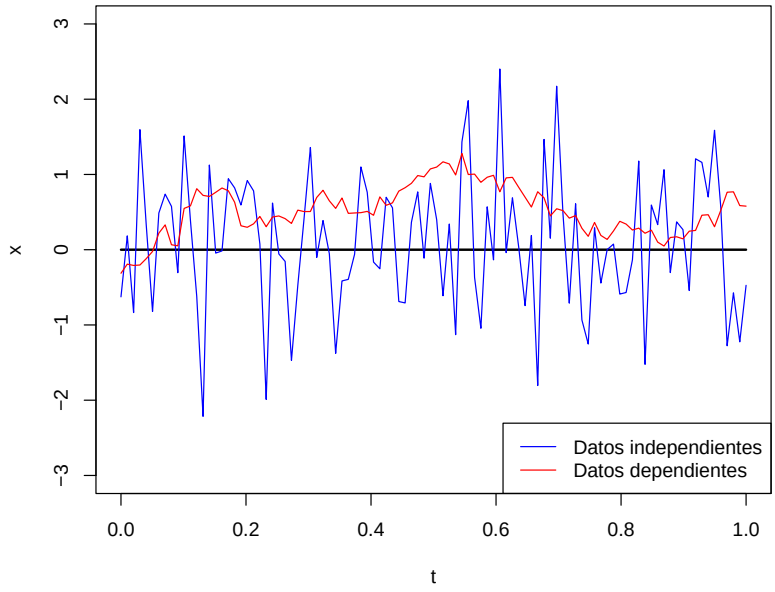
$$Var(X_1 + X_2) = Var(X_1) + Var(X_2) + 2Cov(X_1, X_2)$$

Típicamente $Cov(X_1, X_2) > 0$ por lo que con los métodos "clásicos" (basados en independencia) se suelen producir subestimaciones de las varianzas (IC más estrechos y tendencia a rechazar H_0 en contrastes).

Ejemplo A.1 (Datos simulados dependientes)

Consideramos un proceso temporal estacionario con dependencia exponencial (la dependencia entre las observaciones depende del "salto" entre ellas; ver Ejemplo 6.4 en la Sección 6.3).

```
n <- 100          # Nº de observaciones
t <- seq(0, 1, length = n)
mu <- rep(0, n)   # Media
# mu <- 0.25 + 0.5*t
# mu <- sin(2*pi*t)

# Matriz de covarianzas
t.dist <- as.matrix(dist(t))
t.cov <- exp(-t.dist)
# str(t.cov)
# num [1:100, 1:100] 1 0.99 0.98 0.97 0.96 ...

# Simulación de las observaciones
set.seed(1)
library(MASS)

z <- rnorm(n)
x1 <- mu + z # Datos independientes
x2 <- mvrnorm(1, mu, t.cov) # Datos dependientes

plot(t, mu, type="l", lwd = 2, ylim = c(-3,3), ylab = 'x')
lines(t, x1, col = 'blue')
lines(t, x2, col = 'red')
legend("bottomright", legend = c("Datos independientes", "Datos dependientes"), col = c('blue', 'r
```



En el caso anterior la varianza es uno con ambos procesos. Las estimaciones suponiendo independencia serían:

```
var(x1)
```

```
## [1] 0.8067621
```

```
var(x2)
```

```
## [1] 0.1108155
```

En el caso de datos dependientes se produce una clara subestimación de la varianza.

A.2.1 Métodos para detectar dependencia

Es de esperar que datos cercanos en el tiempo (o en el espacio) sean más parecidos (dependientes) que datos más alejados, hablaríamos entonces de dependencia temporal (espacial o espacio-temporal).

En esta sección nos centraremos en el caso de dependencia temporal (unidimensional). Entre los métodos para detectar este tipo de dependencia destacaríamos:

- Gráficos:
  - Secuencial / Dispersión frente al tiempo
  - Dispersión retardado
  - Correlograma
- Contrastes:
  - Tests basados en rachas
  - Test de Ljung-Box

A.2.2 Gráfico secuencial

El gráfico de dispersión $\{(i, X_i) : i = 1, \dots, n\}$ permite detectar la presencia de un efecto temporal (en la tendencia o en la variabilidad).

- Es importante mantener/guardar el orden de recogida de los datos.

- Si existe una tendencia los datos no son homogéneos (debería tenerse en cuenta la variable índice, o tiempo, como variable explicativa). Podría indicar la presencia de un "efecto aprendizaje".

- Comandos R: `plot(as.ts(x))`

Ejemplo:

```
old.par <- par(mfrow = c(1, 2))
plot(datos, type = 'l')
plot(as.ts(datos))
```

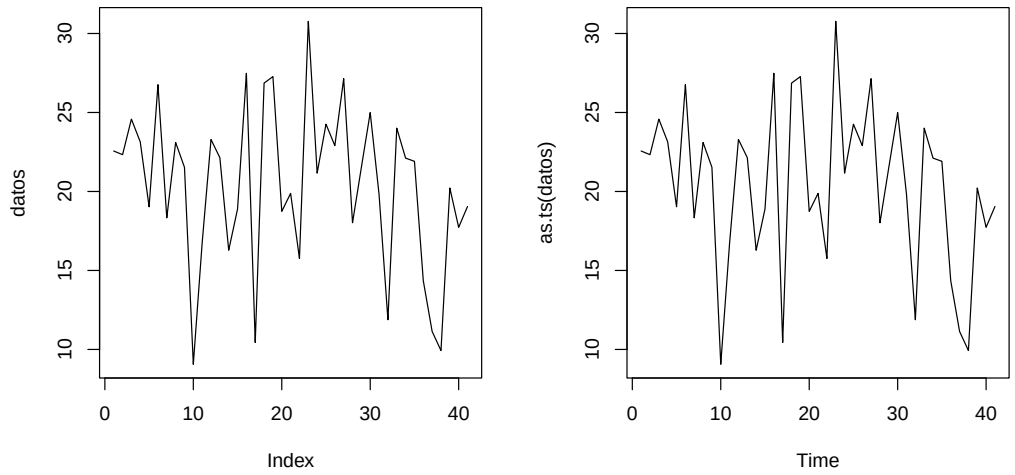


Figura A.3: Ejemplos de gráficos secuenciales.

```
par(old.par)
```

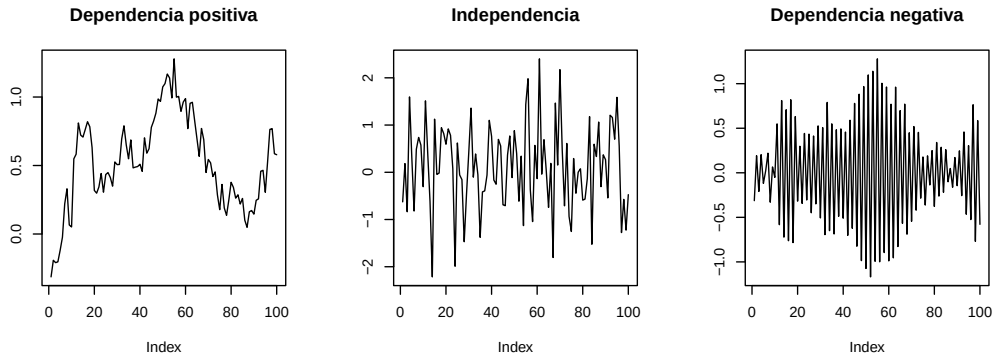
Es habitual que este tipo de análisis se realice sobre los residuos de un modelo de regresión (e.g. `datos <- residuals(modelo)`)

Este gráfico también podría servir para detectar dependencia temporal:

- Valores próximos muy parecidos (valores grandes seguidos de grandes y viceversa) indicarían una posible dependencia positiva.

- Valores próximos dispares (valores grandes seguidos de pequeños y viceversa) indicarían una posible dependencia negativa.

```
old.par <- par(mfrow = c(1, 3))
plot(x2, type = 'l', ylab = '', main = 'Dependencia positiva')
plot(x1, type = 'l', ylab = '', main = 'Independencia')
x3 <- x2 * c(1, -1)
plot(x3, type = 'l', ylab = '', main = 'Dependencia negativa')
```




```
par(old.par)
```

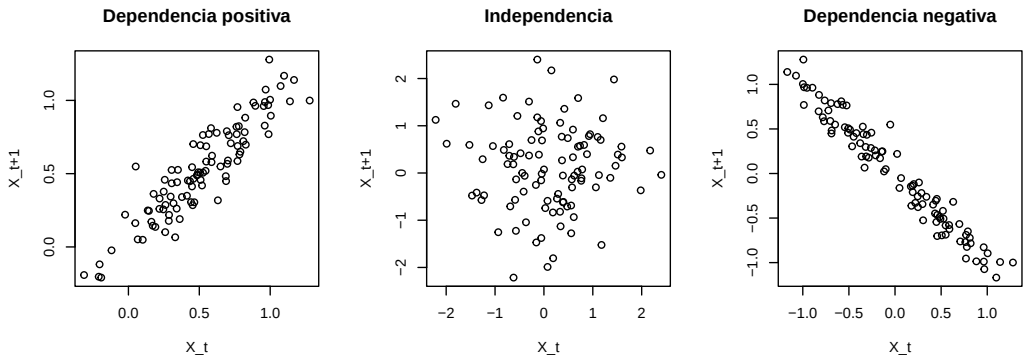
pero suele ser preferible emplear un gráfico de dispersión retardado.

A.2.3 Gráfico de dispersion retardado

El gráfico de dispersión $\{(X_i, X_{i+1}) : i = 1, \dots, n-1\}$ permite detectar dependencias a un retardo (relaciones entre valores separados por un instante)

- Comando R:`plot(x[-length(x)], x[-1], xlab = "X_t", ylab = "X_t+1")`

```
old.par <- par(mfrow = c(1, 3))
plot(x2[-length(x2)], x2[-1], xlab = "X_t", ylab = "X_t+1", main = 'Dependencia positiva')
plot(x1[-length(x1)], x1[-1], xlab = "X_t", ylab = "X_t+1", main = 'Independencia')
plot(x3[-length(x3)], x3[-1], xlab = "X_t", ylab = "X_t+1", main = 'Dependencia negativa')
```

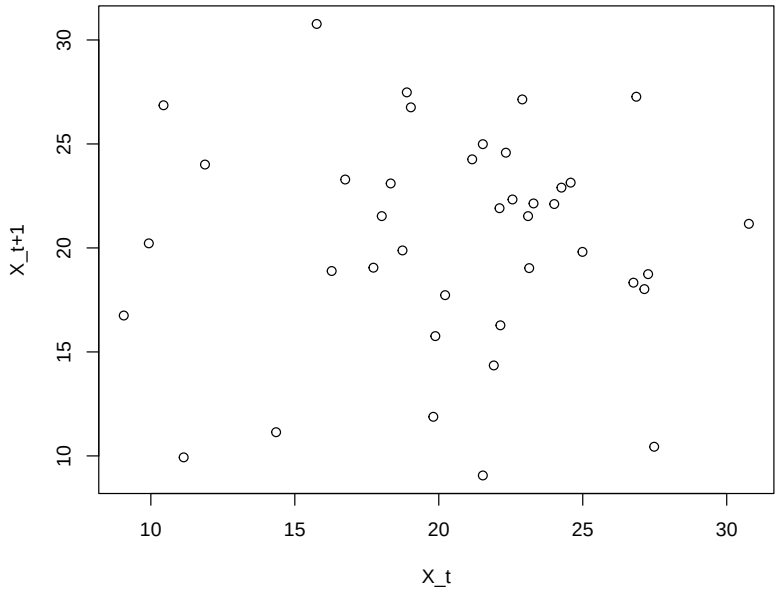



```
par(old.par)
```

Se puede generalizar al gráfico $\{(X_i, X_{i+k}) : i = 1, \dots, n-k\}$ que permite detectar dependencias a k retardos (separadas k instantes).

Ejemplo:

```
# Gráfico de dispersion retardado
plot(datos[-length(datos)], datos[-1], xlab = "X_t", ylab = "X_t+1")
```



El correspondiente coeficiente de correlación es una medida numérica del grado de relación lineal (denominado autocorrelación de orden 1).

```
cor(datos[-length(datos)], datos[-1])
```

```
## [1] 0.01344127
```

Ejemplo A.2 (Calidad de un generador aleatorio)

En el caso de una secuencia muy grande de número pseudoaleatorios (supuestamente independientes), sería muy dificil distinguir un patrón a partir del gráfico anterior. La recomendación en R sería utilizar puntos con color de relleno:

```
plot(u[-length(u)], u[-1], xlab="U_t", ylab="U_t+1", pch=21, bg="white")
```

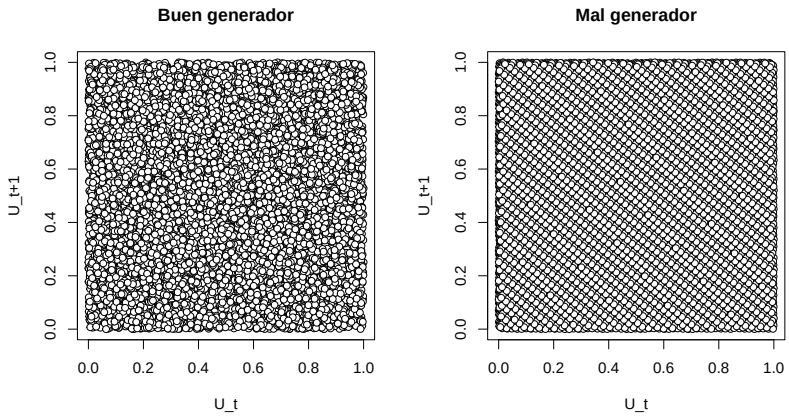


Figura A.4: Ejemplos de gráficos de dispensión retardados de dos secuencias de longitud 10000.

Si se observa algún tipo de patrón indicaría dependencia (se podría considerar como una versión descriptiva del denominado "Parking lot test"). Se puede generalizar también a d-uplas $(X_{t+1}, X_{t+2}, \dots, X_{t+d})$ (ver ejemplo del generador RANDU en Figura 2.1 de la Sección 2.1).

A.2.4 El correlograma

Para estudiar si el grado de relación (lineal) entre X_i e X_{i+k} podemos utilizar el coeficiente de correlación:

$$\rho\left(X_i, X_{i+k}\right) = \frac{Cov\left(X_i, X_{i+k}\right)}{\sigma\left(X_i\right)\sigma\left(X_{i+k}\right)}$$

- En el caso de datos homogéneos (estacionarios) la correlación sería función únicamente del salto:

$$\rho\left(X_i, X_{i+k}\right) \equiv \rho\left(k\right)$$

  denominada función de autocorrelación simple (fas) o correlograma.

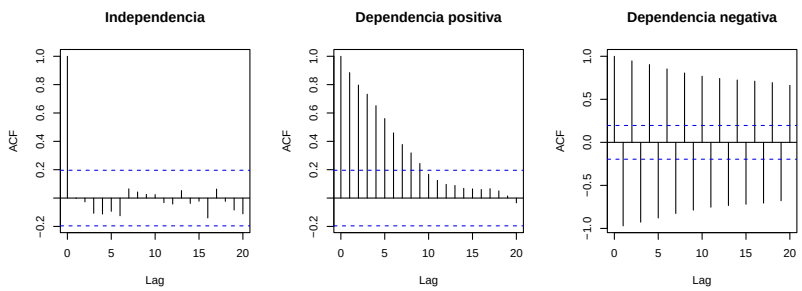
- Su estimador es el correlograma muestral:

$$r(k) = \frac{\sum_{i=1}^{n-k}(X_i - \overline{X})(X_{i+k} - \overline{X})}{\sum_{i=1}^{n}(X_i - \overline{X})^2}$$

- Comando R:`acf(x)`

En caso de independencia es de esperar que las autocorrelaciones muestrales sean próximas a cero (valores "grandes" indicarían dependencia positiva o negativa según el signo).

```
old.par <- par(mfrow = c(1, 3))
acf(x1, main = 'Independencia')
acf(x2, main = 'Dependencia positiva')
acf(x3, main = 'Dependencia negativa')
```




```
par(old.par)
```

Suponiendo normalidad e independencia, asintóticamente:

$$r(k) \underset{aprox.}{\sim} N\left(\rho(k), \frac{1}{n}\right)$$

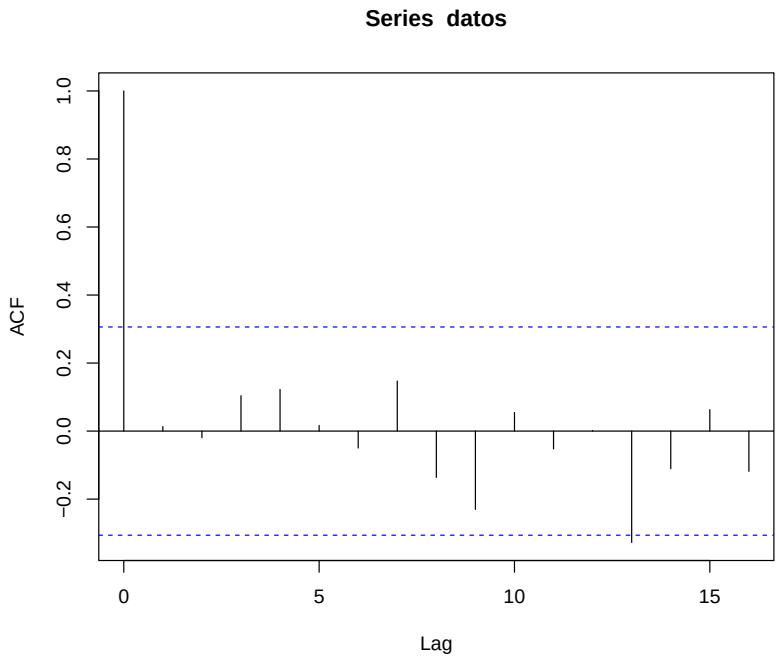
- Si el tamaño muestral es grande, podríamos aceptar $H_0 : \rho\left(k\right) = 0$ si:

$$|r(k)| < \frac{2}{\sqrt{n}}$$

- En el *gráfico de autocorrelaciones muestrales* (también denominado correlograma) se representan las estimaciones $r(k)$ de las autocorrelaciones correspondientes a los primeros retardos (típicamente $k < n/4$) y las correspondientes bandas de confianza (para detectar dependencias significativas).

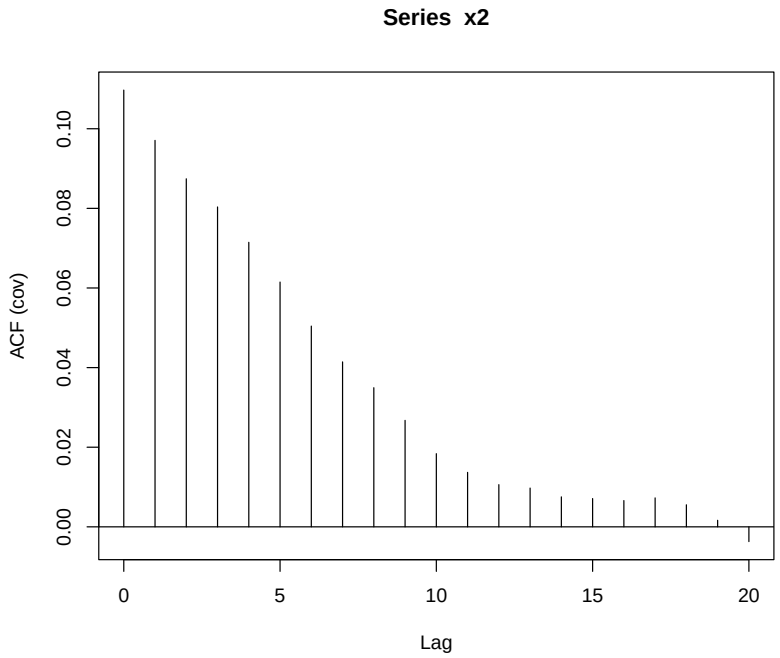
Ejemplo:

```
acf(datos)  # correlaciones
```



La función `acf` también permite estimar el covariograma[3].

```
covar <- acf(x2, type = "covariance")
```



A.2.5 Test de rachas

Permite contrastar si el orden de aparición de dos valores de una variable dicotómica es aleatorio. Supongamos que X toma los valores $+$ y $-$ y que observamos una muestra del tipo:

$$+++---+++--++++++---$$

[3]En algunos campos, como en estadística espacial, en lugar del covariograma se suele emplear el semivariograma $\gamma(k) = C(0) - C(k)$.

y nos interesa contrastar:

$$\begin{cases} H_0 : \textit{La muestra es aleatoria} \\ H_1 : \textit{La muestra no es aleatoria} \end{cases}$$

Una *racha* es una secuencia de observaciones iguales (o similares):

$$\underbrace{++++}_{1}\ \underbrace{---}_{2}\ \underbrace{+++}_{3}\ \underbrace{--}_{4}\ \underbrace{+++++}_{5}\ \underbrace{----}_{6}$$

- Una muestra con "muchas" o "pocas" rachas sugeriría que la muestra no es aleatoria (con dependencia negativa o positiva, respec.).
- Estadístico del contraste:

$$R = \text{"Número total de rachas en la muestra"}$$

- Bajo la hipótesis nula de aleatoriedad:

$$R \underset{aprox.}{\sim} N\left(1 + \frac{2n_1n_2}{n}, \frac{2n_1n_2(2n_1n_2 - n)}{n^2(n-1)}\right)$$

  siendo n_1 y n_2 el número de signos $+$ y $-$ en la muestra, respectivamente ($n_1 + n_2 = n$). Para tamaños muéstrales pequeños ($n < 40$), esta aproximación no es buena y conviene utilizar la distribución exacta (o utilizar corrección por continuidad). Los valores críticos de esta distribución están tabulados.

Este contraste se emplea también para variables continuas, se fija un punto de corte para dicotomizarlas. Normalmente se toma como punto de corte la mediana.

- En este caso si $k = n_1$ ($\simeq n_2$):

$$R \underset{aprox.}{\sim} N\left(k+1, \frac{k(k-1)}{2k-1}\right)$$

- Se rechaza la hipótesis nula de aleatoriedad si el número de rachas es significativamente pequeño o grande.
- Si el tamaño muestral es grande, el p-valor será:

$$p \simeq 2P\left(Z \geq \left|\frac{R - E(R)}{\sqrt{Var(R)}}\right|\right)$$

- Comandos R: `tseries::runs.test(as.factor(x > median(x)))`

Ejemplo:

```
library(tseries)
runs.test(as.factor(datos > median(datos)))
```

```
##
##  Runs Test
##
## data:  as.factor(datos > median(datos))
## Standard Normal = -0.4422, p-value = 0.6583
## alternative hypothesis: two.sided
```

Alternativamente, para evitar el cálculo del punto de corte (la mediana), requerido para dicotomizar la variable continua, se podría emplear una modificación de este contraste, el denominado test de rachas ascendentes y descendentes, en el que se generan los valores $+$ y $-$ dependiendo de si el valor de la secuencia es mayor o menor que el anterior (ver e.g. Downham, 1970). Este contraste es más adecuado para generadores aleatorios.

A.2.6 El contraste de Ljung-Box

Es un test muy utilizado (en series de tiempo) para contrastar la hipótesis de independencia. Se contrasta la hipótesis nula de que las primeras m autocorrelaciones son cero:

$$\begin{cases} H_0 : \rho_1 = \rho_2 = \ldots = \rho_m = 0 \\ H_1 : \rho_i \neq 0 \text{ para algún } i \end{cases}$$

- Se elige un m tal que la estimación $r(m)$ de $\rho_m = \rho(m)$ sea "fiable" (e.g. $10 \log_{10} n$).
- El estadístico del contraste:

$$Q = n(n+2) \sum_{k=1}^{m} \frac{r(k)^2}{n-k} \underset{aprox.}{\sim} \chi^2_m, \text{ si } H_0 \text{ es cierta.}$$

- Se rechaza H_0 si el valor observado es grande ($Q \geq \chi^2_{m,1-\alpha}$):

$$p = P\left(\chi^2_m \geq Q\right)$$

- Comandos R:

```
Box.test(x, type=Ljung)
Box.test(x, lag, type=Ljung)
```

Ejemplo:

```
Box.test(datos, type="Ljung") # Contrasta si la primera autocorrelación es nula
```

```
##
##  Box-Ljung test
##
## data:  datos
## X-squared = 0.0078317, df = 1, p-value = 0.9295
```

```
Box.test(datos, lag=5, type="Ljung") # Contrasta si las 5 primeras autocorrelaciones son nulas
```

```
##
##  Box-Ljung test
##
## data:  datos
## X-squared = 1.2556, df = 5, p-value = 0.9394
```

NOTA: Cuando se trabaja con residuos de un modelo lineal, para contrastar que la primera autocorrelación es cero, es preferible emplear el test de Durbin-Watson implementado en la función `dwtest()` del paquete `lmtest`.

A.3 Contrastes específicos para generadores aleatorios

Los contrastes generales anteriores pueden ser muy poco adecuados para testear generadores de números pseudoaleatorios (ver e.g. L'Ecuyer y Simard, 2007). Por ese motivo se han desarrollado contrastes específicos, principalmente con el objetivo de encontrar un generador con buenas propiedades criptográficas.

Muchos de estos contrastes están basados en la prueba chi-cuadrado y trabajan con enteros en lugar de los valores uniformes. El procedimiento habitual consiste en fijar un entero positivo K, y discretizar los valores uniformes $U_1, U_2, \ldots, U_n$, de la forma:

$$X_i = \lfloor K \cdot U_i \rfloor + 1,$$

donde $\lfloor u \rfloor$ denota la parte entera de u. De esta forma se consigue una sucesión de enteros aleatorios supuestamente independientes con distribución uniforme en $\{1, \ldots, K\}$.

En esta sección se describirán algunos de los métodos tradicionales en este campo con fines ilustrativos. Si realmente el objetivo es diagnosticar la calidad de un generador, la recomendación sería emplear las baterías de contrastes más recientes descritas en la Sección 2.3.2.

A.3.1 Contraste de frecuencias

Empleando la discretización anterior se simplifica notablemente el contraste chi-cuadrado de bondad de ajuste a una uniforme, descrito en la Sección A.1.4 e implementado en la función `chisq.cont.test()`. En este caso bastaría con contrastar la equiprobabilidad de la secuencia de enteros (empleando directamente la función `chisq.test()`) y este método de denomina *contraste de frecuencias* (frequency test). Por ejemplo:

```
# Generar
set.seed(1)
u <- runif(1000)
# Discretizar
k <- 10
x <- floor(k*u) + 1
# Test chi-cuadrado
f <- table(factor(x, levels = seq_len(k)))
chisq.test(f)
```

```
##
##  Chi-squared test for given probabilities
##
## data:  f
## X-squared = 10.26, df = 9, p-value = 0.3298
```

Este código está implementado en la función `simres::freq.test()` (fichero *test.R*) y podríamos emplear:

```
library(simres)
freq.test(u, nclass = k)
# Alternativamente
# chisq.cont.test(u, distribution = "unif", nclass = k, output = FALSE, min = 0, max = 1)
```

A.3.2 Contraste de series

El contraste anterior se puede generalizar a contrastar la uniformidad de las d-uplas $(X_{t+1}, X_{t+2}, \dots, X_{t+d})$ con $t = (i-1)d$, $i = 1, \dots, m$ siendo $m = \lfloor n/d \rfloor$. La idea es que troceamos el hipercubo $[0,1]^d$ en K^d celdas equiprobables. Considerando como categorías todos los posibles valores de las d-uplas, podemos emplear el estadístico chi-cuadrado para medir la discrepancia entre las frecuencias observadas y las esperadas, iguales todas a $\frac{m}{K^d}$. Las elecciones más frecuentes son $d = 2$ (contraste de pares seriados) y $K = 8$, 10 ó 20. Por ejemplo, la función `serial.test()` del paquete `randtoolbox` implementa este contraste para $d = 2$.

Para que la prueba chi-cuadrado sea fiable el valor de n debería ser grande en comparación con el número de categorías K^d (e.g. $n \geq 5dK^d$). Si se considera un valor $d \geq 3$ puede ser necesario reducir considerablemente el valor de K para evitar considerar demasiadas categorías. Alternativamente se podrían considerar distintas técnicas para agrupar estas categorías, por ejemplo como se hace en el contraste del poker o del coleccionista descritos a continuación.

A.3.3 El contraste del poker

En el contrate del poker "clásico" se consideran conjuntos sucesivos de cinco enteros ($d = 5$) y, para cada uno, se determina cuál de las siguientes posibilidades se da:

1. Un mismo entero se repite cinco veces (abreviadamente, $AAAAA$).
2. Un mismo entero se repite cuatro veces y otro distinto aparece una vez ($AAAAB$).

3. Un entero se repite tres veces y otro distinto se repite dos ($AAABB$).
4. Un entero se repite tres veces y otros dos distintos aparecen una vez cada uno ($AAABC$).
5. Un entero se repite dos veces, otro distinto se repite también dos veces y un tercer entero diferente aparece una sóla vez ($AABBC$).
6. Un entero se repite dos veces y otros tres distintos aparecen una vez cada uno ($AABCD$).
7. Los cinco enteros que aparecen son todos distintos ($ABCDE$).

Bajo las hipótesis de aleatoriedad y uniformidad, se pueden calcular las probabilidades de estas modalidades. Por ejemplo para $K = 10$ obtendríamos:

$$\begin{aligned}
&P(AAAAA) = 0.0001, P(AAAAB) = 0.0045, P(AAABB) = 0.0090,\\
&P(AAABC) = 0.0720, P(AABBC) = 0.1080, P(AABCD) = 0.5040,\\
&P(ABCDE) = 0.3024.
\end{aligned}$$

Es frecuente que las clases $AAAAA$ y $AAAAB$ se agrupen a la hora de aplicar el test chi-cuadrado, ya que, en caso contrario, la restricción habitual $e_i \geq 5$ llevaría a que $0.0001 \cdot \frac{n}{5} \geq 5$, es decir, $n \geq 250\,000$.

Es habitual simplificar el contraste anterior para facilitar su implementación definiendo las categorías según el número de enteros distintos de entre los cinco observados. Así obtendríamos:

$$\begin{aligned}
&P(1 \text{ entero diferente}) = 0.0001, P(2 \text{ enteros diferentes}) = 0.0135,\\
&P(3 \text{ enteros diferentes}) = 0.1800, P(4 \text{ enteros diferentes}) = 0.5040,\\
&P(5 \text{ enteros diferentes}) = 0.3024,
\end{aligned}$$

procediendo también a agrupar las dos primeras modalidades.

En el caso general de considerar d-uplas (manos de d cartas con K posibilidades cada una), la probabilidad de obtener c valores (cartas) diferentes es (e.g. Knuth, 2002, Sección 3.3.2, p. 64):

$$P(C = c) = \frac{K!}{(K-c)!K^d} S(d, c),$$

donde $S(d, c)$ es el número de Stirling de segunda clase, definido como el número de formas que existen de hacer una partición de un conjunto de d elementos en c subconjuntos:

$$S(d, c) = \frac{1}{c!} \sum_{i=0}^{c} (-1)^i \binom{c}{i} (c-i)^d.$$

Por ejemplo, la función `poker.test()` del paquete `randtoolbox` implementa este contraste para el caso de $d = K$.

A.3.4 El contraste del coleccionista

Por simplicidad describiremos el caso de $d = 1$ (con K categorías). Considerando la sucesión de enteros aleatorios se procede (como un coleccionista) a contabilizar cuál es el número, Q, (aleatorio) de valores consecutivos hasta que se completa la colección de todos los enteros entre 1 y K. Obviamente, bajo las hipótesis de aleatoriedad y uniformidad, cada posible entero entre 1 y K tiene la misma probabilidad de aparecer en cada generación y, por tanto, resulta posible calcular la distribución de probabilidad de Q. De esta forma podemos utilizar los valores calculados de las probabilidades

$$P(Q = K), P(Q = K + 1), \ldots, P(Q = M - 1), P(Q \geq M),$$

para obtener las frecuencias esperadas de cada clase y confrontarlas con las observadas vía el estadístico chi-cuadrado (e.g. Knuth, 2002, Sección 3.3.2, p. 65).

Existen varias elecciones comunes de K y M. Tomando $K = 5$ con clases $Q = 5$, $Q = 6$, ..., $Q = 19$, $Q \geq 20$, las probabilidades vendrían dadas por:

$$\begin{aligned}
P(Q = 5) &= 0.03840000, & P(Q = 6) &= 0.07680000,\\
P(Q = 7) &= 0.09984000, & P(Q = 8) &= 0.10752000,\\
P(Q = 9) &= 0.10450944, & P(Q = 10) &= 0.09547776,\\
P(Q = 11) &= 0.08381645, & P(Q = 12) &= 0.07163904,\\
P(Q = 13) &= 0.06011299, & P(Q = 14) &= 0.04979157,\\
P(Q = 15) &= 0.04086200, & P(Q = 16) &= 0.03331007,\\
P(Q = 17) &= 0.02702163, & P(Q = 18) &= 0.02184196,\\
P(Q = 19) &= 0.01760857, & P(Q \geq 20) &= 0.07144851.
\end{aligned}$$

Para $K = 10$ se podrían considerar las siguientes categorías (con sus correspondientes probabilidades):

$$\begin{aligned}
P(10 \leq Q \leq 19) &= 0.17321155, & P(20 \leq Q \leq 23) &= 0.17492380,\\
P(24 \leq Q \leq 27) &= 0.17150818, & P(28 \leq Q \leq 32) &= 0.17134210,\\
P(33 \leq Q \leq 39) &= 0.15216056, & P(Q \geq 40) &= 0.15685380.
\end{aligned}$$

Apéndice B

Integración numérica

En muchos casos nos puede interesar la aproximación de una integral definida. En estadística, además del caso de Inferencia Bayesiana (que se trató en el Capítulo 11 empleando Integración Monte Carlo y MCMC), nos puede interesar por ejemplo aproximar mediante simulación el error cuadrático integrado medio (MISE) de un estimador. Por ejemplo, en el caso de una densidad univariante sería de la forma:

$$MISE\left(\hat{f}\right) = \int E\left[\left(\hat{f}(x) - f(x)\right)^2\right] dx$$

Cuando el numero de dimensiones es pequeño, nos puede interesar emplear un método numérico para aproximar este tipo de integrales.

B.1 Integración numérica unidimensional

Supongamos que nos interesa aproximar una integral de la forma:

$$I = \int_a^b h(x)dx.$$

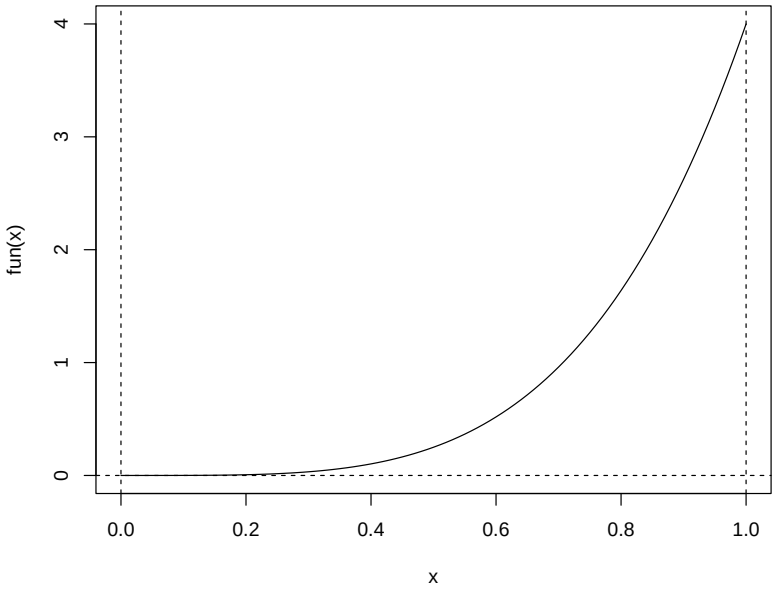
.

Consideraremos como ejemplo:

$$\int_0^1 4x^4 dx = \frac{4}{5}$$

.

```
fun <- function(x) return(4 * x^4)
curve(fun, 0, 1)
abline(h = 0, lty = 2)
abline(v = c(0, 1), lty = 2)
```



B.1.1 Método del trapezoide

La regla de los trapecios es una forma de aproximar la integral utilizando n trapecios. Si se consideran n subintervalos en $[a, b]$ de longitud $h = \frac{b-a}{n}$ (i.e. $n+1$ puntos regularmente espaciados cubriendo el dominio), y se aproxima linealmente la función en cada subintervalo, se obtiene que:

$$\int_a^b f(x)\,dx \approx \frac{h}{2}[f(a) + 2f(a+h) + 2f(a+2h) + ... + f(b)]$$

```
trapezoid.vec <- function(f.vec, h = 0.01) {
# Integración numérica unidimensional entre a y b
# utilizando el método del trapezoide
# (se aproxima f linealmente en cada intervalo)
  n <- length(f.vec)
  return(h*(f.vec[1]/2 + sum(f.vec[2:(n-1)]) + f.vec[n]/2))
}

trapezoid <- function(fun, a = 0, b = 1, n = 100) {
# Integración numérica de fun (función unidimensional) entre a y b
# utilizando el método del trapezoide con n subdivisiones
# (se aproxima f linealmente en cada intervalo)
# Se asume a < b y n entero positivo
  h <- (b-a)/n
  x.vec <- seq(a, b, by = h)
  f.vec <- sapply(x.vec, fun)
  return(trapezoid.vec(f.vec, h))
}

trapezoid(fun, 0, 1, 20)
```

```
## [1] 0.8033325
```

El error en esta aproximación se corresponde con:

$$\frac{(b-a)^3}{12n^2} f''(\xi),$$

para algún $a \leq \xi \leq b$ (dependiendo del signo de la segunda derivada, i.e. de si la función es cóncava o convexa, el error será negativo ó positivo). El error máximo absoluto es $\frac{(b-a)^3}{12n^2} \max_{a\leq\xi\leq b} |f''(\xi)|$. En el caso general multidimensional sería $O(n^{-\frac{2}{d}})$.

B.1.2 Regla de Simpson

Se divide el intervalo n subintervalos de longitud $h = \frac{b-a}{n}$ (con n par), considerando $n+1$ puntos regularmente espaciados $x_i = a + ih$, para $i = 0, 1, ..., n$. Aproximando de forma cuadrática la función en cada subintervalo $[x_{j-1}, x_{j+1}]$ (considerando 3 puntos), se obtiene que:

$$\int_a^b f(x)\,dx \approx \frac{h}{3}\left[f(x_0) + 2\sum_{j=1}^{(n/2)-1} f(x_{2j}) + 4\sum_{j=1}^{n/2} f(x_{2j-1}) + f(x_n)\right],$$

```
simpson <- function(fun, a, b, n = 100) {
# Integración numérica de fnt entre a y b
# utilizando la regla de Simpson con n subdivisiones
# (se aproxima fun de forma cuadrática en cada par de intervalos)
# fnt es una función de una sola variable
# Se asume a < b y n entero positivo par
  n <- max(c(2*(n %/% 2), 4))
  h <- (b-a)/n
  x.vec1 <- seq(a+h, b-h, by = 2*h)
  x.vec2 <- seq(a+2*h, b-2*h, by = 2*h)
  f.vec1 <- sapply(x.vec1, fun)
  f.vec2 <- sapply(x.vec2, fun)
  return(h/3*(fun(a) + fun(b) + 4*sum(f.vec1) + 2*sum(f.vec2)))
}

simpson(fun, 0, 1, 20)
```

```
## [1] 0.8000033
```

El máximo error (en el caso unidimensional) viene dado por la expresión:

$$\frac{(b-a)^5}{180n^4} \max_{a\leq\xi\leq b} \left|f^{(4)}(\xi)\right|.$$

En el caso general multidimensional sería $O(n^{-\frac{4}{d}})$.

B.1.3 Cuadratura adaptativa

En lugar de evaluar la función en una rejilla regular (muestrear por igual el dominio), puede interesar ir añadiendo puntos sólo en los lugares donde se mejore la aproximación (en principio donde hay mayor área).

```
quadrature <- function(fun, a, b, tol=1e-8) {
    # numerical integration using adaptive quadrature

  simpson2 <- function(fun, a, b) {
    # numerical integral using Simpson's rule
    # assume a < b and n = 2
    return((b-a)/6 * (fun(a) + 4*fun((a+b)/2) + fun(b)))
  }

    quadrature_internal <- function(S.old, fun, a, m, b, tol, level) {
        level.max <- 100
        if (level > level.max) {
```

```
            cat ("recursion limit reached: singularity likely\n")
            return (NULL)
        }
        S.left <- simpson2(fun, a, m)
        S.right <- simpson2(fun, m, b)
        S.new <- S.left + S.right
        if (abs(S.new-S.old) > tol) {
            S.left <- quadrature_internal(S.left, fun,
                                          a, (a+m)/2, m, tol/2, level+1)
            S.right <- quadrature_internal(S.right, fun,
                                           m, (m+b)/2, b, tol/2, level+1)
            S.new <- S.left + S.right
        }
        return(S.new)
    }

    level = 1
    S.old <- (b-a) * (fun(a) + fun(b))/2
    S.new <- quadrature_internal(S.old, fun,
                                 a, (a+b)/2, b, tol, level+1)
    return(S.new)
}

quadrature(fun, 0, 1)
```

```
## [1] 0.8
```

Fuente: r-blogger Guangchuang Yu

B.1.4 Comandos de R

La función `integrate()` implementa un método de cuadratura adaptativa que admite límites infinitos

```
integrate(fun, 0, 1)  # Cuidado: fun debe ser vectorial...
```

```
## 0.8 with absolute error < 8.9e-15
```

Alternativamente, para dominios acotados, se puede emplear la función `MASS::area()` (suele dar muy buenos resultados, aunque los autores la desarrollaron inicialmente para fines ilustrativos):

```
require(MASS)
area(fun, 0, 1)
```

```
## [1] 0.8000001
```

B.2 Integración numérica bidimensional

Supongamos que nos interesa aproximar una integral de la forma:

$$I = \int_{a_x}^{b_x} \int_{a_y}^{b_y} f(x,y)dydx$$

.

Consideraremos como ejemplo:

$$\int_{-1}^{1} \int_{-1}^{1} \left(x^2 - y^2\right) dxdy = 0$$

.

```
f2d <- function(x,y) x^2 - y^2
```

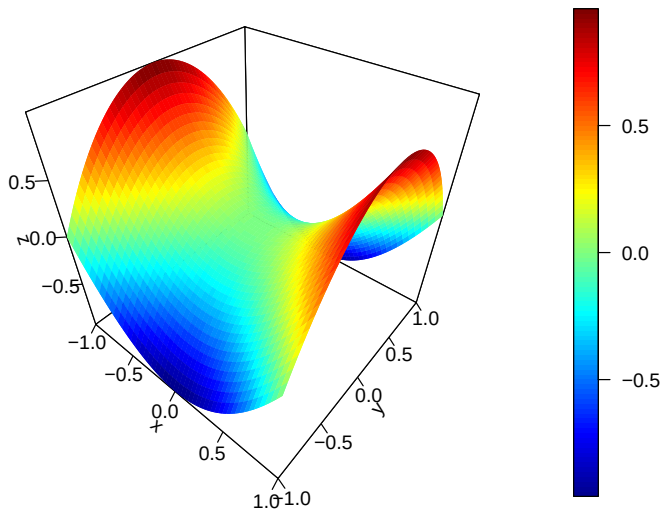
Es habitual (especialmente en simulación) que la función se evalúe en una rejilla:

```
ax = -1
ay = -1
bx = 1
by = 1
nx = 21
ny = 21
x <- seq(ax, bx, length = nx)
y <- seq(ay, by, length = ny)
z <- outer(x, y, f2d)

hx <- x[2]-x[1]
hy <- y[2]-y[1]
```

B.2.1 Representación gráfica

Puede ser de utilidad las herramientas de los paquetes `plot3D` y `plot3Drgl` (también se pueden utilizar las funciones `spersp`, `simage`, `spoints` y `splot` del paquete `npsp`).

```
if(!require(plot3D)) stop('Required pakage `plot3D` not installed.')

# persp3D(z = z, x = x, y = y)

persp3D.f2d <- function(f2d, ax=-1, bx=1, ay=-1, by=1, nx=21, ny=21, ...) {
  x <- seq(ax, bx, length = nx)
  y <- seq(ay, by, length = ny)
  z <- outer(x, y, f2d)
  persp3D(x, y, z, ...)
}

persp3D.f2d(f2d, -1, 1, -1, 1, 50, 50, ticktype = "detailed")
```



B.2.2 Método del trapezoide

Error $O(n^{-\frac{2}{d}})$.

```
trapezoid.mat <- function(z, hx, hy) {
# Integración numérica bidimensional
# utilizando el método del trapezoide (se aproxima f linealmente)
  f.vec <- apply(z, 1, function(x) trapezoid.vec(x, hx))
  return(trapezoid.vec(f.vec, hy))
}

# trapezoid.mat(z, hx, hy)

trapezoid.f2d <- function(f2d, ax=-1, bx=1, ay=-1, by=1, nx=21, ny=21) {
  x <- seq(ax, bx, length = nx)
  y <- seq(ay, by, length = ny)
  hx <- x[2]-x[1]
  hy <- y[2]-y[1]
  z <- outer(x, y, f2d)
  trapezoid.mat(z, hx, hy)
}

trapezoid.f2d(f2d, -1, 1, -1, 1, 101, 101)
```

```
## [1] -8.881784e-18
```

B.2.3 Comandos de R

Suponiendo que la función es vectorial, podemos emplear:

```
integrate( function(y) {
    sapply(y, function(y) {
      integrate(function(x) f2d(x,y), ax, bx)$value }) },
    ay, by)
```

```
## -2.775558e-17 with absolute error < 1.1e-14
```

Si la función no es vectorial y solo admite parámetros escalares:

```
integrate(function(y) {
    sapply(y, function(y) {
      integrate(function(x) {
        sapply(x, function(x) f2d(x,y)) }, ax, bx)$value }) },
    ay, by)
```

Fuente: tolstoy.newcastle.edu.au.

Alternativamente se podría emplear la función `cubintegrate()` del paquete `cubature`.

Apéndice C

Introducción al procesamiento en paralelo en R

En este apéndice se pretenden mostrar las principales herramientas para el procesamiento en paralelo disponibles en R y dar una idea de su funcionamiento. Para más detalles se recomienda ver CRAN Task View: High-Performance and Parallel Computing with R (además de HPC, High Performance Computing, también incluye herramientas para computación distribuida)[1].

C.1 Introducción

Emplearemos la siguiente terminología:

- **Núcleo**: término empleado para referirse a un procesador lógico de un equipo (un equipo puede tener un único procesador con múltiples núcleos que pueden realizar operaciones en paralelo). También podría referirse aquí a un equipo (nodo) de una red (clúster de equipos; en la práctica cada uno puede tener múltiples núcleos). *Un núcleo puede ejecutar procesos en serie.*
- **Clúster**: colección de núcleos en un equipo o red de equipos. *Un clúster puede ejecutar varios procesos en paralelo.*

Por defecto la versión oficial de R emplea un único núcleo, aunque se puede compilar de forma que realice cálculos en paralelo (e.g. librería LAPACK). También están disponibles otras versiones de R que ya implementan por defecto procesamiento en paralelo (multithread): - Microsoft R Open: versión de R con rendimiento mejorado.

Métodos simples de paralelización[2]:

- *Forking*: Copia el proceso de R a un nuevo núcleo (se comparte el entorno de trabajo). Es el más simple y eficiente pero **no está disponible en Windows**.
- *Socket*: Lanza una nueva versión de R en cada núcleo, como si se tratase de un cluster de equipos comunicados a través de red (hay que crear un entorno de trabajo en cada núcleo). Disponible en todos los sistemas operativos.

C.2 Paquetes en R

Hay varios paquetes que se pueden usar para el procesamiento paralelo en R, entre ellos podríamos destacar:

[1] También puede ser de interés la presentación R y HPC (uso de R en el CESGA) de Aurelio Rodríguez en las VI Xornada de Usuarios de R en Galicia

[2] Realmente las herramientas estándar son *OpenMP* para el procesamiento en paralelo con memoria compartida en un único equipo y *MPI* para la computación distribuida en múltiples nodos.

- `parallel`: forma parte de la instalación base de R y fusiona los paquetes `multicore` (forking) y `snow` (sockets; Simple Network of Workstations). Además incluye herramientas para la generación de números aleatorios en paralelo (cada proceso empleara una secuencia y los resultados serán reproducibles).

  Incluye versiones "paralelizadas" de la familia `*apply()`: `mclapply()` (forking), `parLapply()` (socket), …

- `foreach`: permite realizar iteraciones y admite paralelización con el operador `%dopar%`, aunque requiere paquetes adicionales como `doSNOW` o `doParallel` (recomendado).

- `rslurm`: permite la ejecución distribuida en clústeres Linux que implementen SLURM (Simple Linux Utility for Resource Management), un gestor de recursos de código abierto muy empleado.

C.3 Ejemplos

Si se emplea el paquete `parallel` en sistemas tipo Unix (Linux, Mac OS X, …), se podría evaluar en paralelo una función llamando directamente a `mclapply()`. Por defecto empleará todos los núcleos disponibles, pero se puede especificar un número menor mediante el argumento `mc.cores`.

```
library(parallel)
ncores <- detectCores()
ncores
```

```
## [1] 8
```

```
func <- function(k) {
  i_boot <- sample(nrow(iris), replace = TRUE)
  lm(Petal.Width ~ Petal.Length, data = iris[i_boot, ])$coefficients
}

RNGkind("L'Ecuyer-CMRG") # Establecemos Pierre L'Ecuyer's RngStreams...
set.seed(1)

system.time(res.boot <- mclapply(1:100, func)) # En Windows llama a lapply() (mc.cores = 1)
```

```
##    user  system elapsed
##    0.07    0.00    0.08
```

```
# res.boot <- mclapply(1:100, func, mc.cores = ncores - 1) # En Windows genera un error si mc.cores
```

En Windows habría que crear previamente un cluster, llamar a una de las funciones `par*apply()` y finalizar el cluster:

```
cl <- makeCluster(ncores - 1, type = "PSOCK")
clusterSetRNGStream(cl, 1) # Establecemos Pierre L'Ecuyer's RngStreams con semilla 1...

system.time(res.boot <- parSapply(cl, 1:100, func))
```

```
##    user  system elapsed
##    0.00    0.00    0.03
```

```
# stopCluster(cl)

str(res.boot)
```

```
##  num [1:2, 1:100] -0.415 0.429 -0.363 0.42 -0.342 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:2] "(Intercept)" "Petal.Length"
##   ..$ : NULL
```

Esto también se puede realizar en Linux (`type = "FORK"`), aunque podríamos estar trabajando ya en un cluster de equipos…

También podríamos emplear balance de carga si el tiempo de computación es variable (e.g. `parLapplyLB()` o `clusterApplyLB()`) pero no sería recomendable si se emplean números pseudo-aleatorios (los resultados no serían reproducibles).

Además, empleando las herramientas del paquete `snow` se puede representar el uso del cluster (*experimental* en Windows):

```
# library(snow)
ctime <- snow::snow.time(snow::parSapply(cl, 1:100, func))
ctime
plot(ctime)
```

Hay que tener en cuenta la sobrecarga adicional debida a la comunicación entre nodos al paralelizar (especialmente con el enfoque de socket).

C.3.1 Procesamiento en paralelo con la función `boot()`

La función `boot::boot()` incluye parámetros para el procesamiento en paralelo: `parallel = c("no", "multicore", "snow")`, `ncpus`, `cl`. Si `parallel = "snow"` se crea un clúster en la máquina local durante la ejecución, salvo que se establezca con el parámetro `cl`.

Veamos un ejemplo empleando una muestra simulada:

```
n <- 100
rate <- 0.01
mu <- 1/rate
muestra <- rexp(n, rate = rate)
media <- mean(muestra)
desv <- sd(muestra)

library(boot)
statistic <- function(data, i){
  remuestra <- data[i]
  c(mean(remuestra), var(remuestra)/length(remuestra))
}
B <- 2000
set.seed(1)

system.time(res.boot <- boot(muestra, statistic, R = B))
```

```
##    user  system elapsed
##    0.05    0.00    0.05
```

```
# system.time(res.boot <- boot(muestra, statistic, R = B, parallel = "snow"))
system.time(res.boot <- boot(muestra, statistic, R = B, parallel = "snow", cl = cl))
```

```
##    user  system elapsed
##    0.06    0.00    0.06
```

C.3.2 Estudio de simulación

Si se trata de un estudio más complejo, como por ejemplo un estudio de simulación en el que se emplea bootstrap, la recomendación sería tratar de paralelizar en el nivel superior para minimizar la sobrecarga debida a la comunicación entre nodos.

Por ejemplo, a continuación se realiza un estudio de simulación comparando las probabilidades de cobertura y las longitudes de los intervalos de confianza implementados en la función `boot.ci()`.

```
t.ini <- proc.time()

nsim <- 500

getSimulation <- function(isim, B = 2000, n = 30, alfa = 0.1, mu = 100) {
    rate <- 1/mu # 0.01
    resnames <- c("Cobertura", "Longitud")
    # intervals <- c("Normal", "Percentil", "Percentil-t", "Percentil-t simetrizado")
    intervals <- c("Normal", "Basic", "Studentized", "Percentil", "BCa")
    names(intervals) <- c("normal","basic", "student", "percent", "bca")
    intervals <- intervals[1:4]
    resultados <- array(dim = c(length(resnames), length(intervals)))
    dimnames(resultados) <- list(resnames, intervals)
    # for (isim in 1:nsim) { # isim <- 1
    muestra <- rexp(n, rate = 0.01)
    media <- mean(muestra)
    desv <- sd(muestra)
    # boot()
    library(boot)
    statistic <- function(data, i){
      remuestra <- data[i]
      c(mean(remuestra), var(remuestra)/length(remuestra))
    }
    res.boot <- boot(muestra, statistic, R = B)
    res <- boot.ci(res.boot, conf = 1 - alfa)
    # Intervalos
    res <- sapply(res[names(intervals)], function(x) {
      l <- length(x)
      x[c(l-1, l)]
    })
    # resultados
    resultados[1, ] <- apply(res, 2,
                                 function(ic) (ic[1] < mu) && (mu < ic[2])) # Cobertura
    resultados[2, ] <- apply(res, 2, diff) # Longitud
    resultados
}

parallel::clusterSetRNGStream(cl)
result <- parLapply(cl, 1:nsim, getSimulation)
# stopCluster(cl)

# result
t.fin <- proc.time() - t.ini
print(t.fin)
```

```
##    user  system elapsed
##    0.00    0.00    7.74
```

```
resnames <- c("Cobertura", "Longitud")
intervals <- c("Normal", "Basic", "Studentized", "Percentil", "BCa")
names(intervals) <- c("normal","basic", "student", "percent", "bca")
intervals <- intervals[1:4]
resultados <- sapply(result, function(x) x)
dim(resultados) <- c(length(resnames), length(intervals), nsim)
dimnames(resultados) <- list(resnames, intervals, NULL)

res <- t(apply(resultados, c(1, 2), mean))
```

```
res
```

```
##              Cobertura Longitud
## Normal           0.866 57.05639
## Basic            0.860 56.97389
## Studentized      0.900 65.72609
## Percentil        0.868 56.97389
```

```
knitr::kable(res, digits = 3)
```

	Cobertura	Longitud
Normal	0.866	57.056
Basic	0.860	56.974
Studentized	0.900	65.726
Percentil	0.868	56.974

El último paso es finalizar el cluster:

```
stopCluster(cl)
```

Apéndice D

Soluciones ejercicios

A continuación se muestran soluciones de algunos de los ejercicios no resueltos en el texto.

D.1 Capítulo 1 Introducción a la simulación

D.1.1 Ejercicio 1.1

Enunciado 1.1:

Sea (X, Y) es un vector aleatorio con distribución uniforme en el cuadrado $[-1, 1] \times [-1, 1]$ de área 4.

a) Aproximar mediante simulación $P(X + Y \leq 0)$ y compararla con la probabilidad teórica (obtenida aplicando la regla de Laplace $\frac{\text{área favorable}}{\text{área posible}}$).

Generamos `nsim = 10000` valores del proceso bidimensional:

```
set.seed(1)
nsim <- 10000
x <- runif(nsim, -1, 1)
y <- runif(nsim, -1, 1)
```

La probabilidad teórica es 1/2 y la aproximación por simulación es la frecuencia relativa del suceso en los valores generados (para calcularla podemos aprovechar que R maneja internamente los valores lógicos como 1, `TRUE`, y 0, `FALSE`):

```
indice <- (x + y < 0)
sum(indice)/nsim
```

```
## [1] 0.4996
```

Alternativamente (la frecuencia relativa es un caso particular de la media) se puede obtener de forma más simple como:

```
mean(indice)
```

```
## [1] 0.4996
```

---

b) Aproximar el valor de π mediante simulación a partir de $P(X^2 + Y^2 \leq 1)$.

```
set.seed(1)
n <- 10000
x <- runif(n, -1, 1)
```

```
y <- runif(n, -1, 1)
indice <- (x^2+y^2 < 1)
mean(indice)
```

```
## [1] 0.7806
```

```
pi/4
```

```
## [1] 0.7853982
```

```
pi_aprox <- 4*mean(indice)
pi_aprox
```

```
## [1] 3.1224
```

Generamos el correspondiente gráfico (ver Figura D.1) (los puntos con color negro tienen distribución uniforme en el círculo unidad; esto está relacionado con el método de aceptación-rechazo, ver Ejemplo 6.3, o con el denominado método *hit-or-miss*).

```
# Colores y símbolos dependiendo de si el índice correspondiente es verdadero:
color <- ifelse(indice, "black", "red")
simbolo <- ifelse(indice, 1, 4)
plot(x, y, pch = simbolo, col = color,
     xlim = c(-1, 1), ylim = c(-1, 1), xlab="X", ylab="Y", asp = 1)
     # asp = 1 para dibujar circulo
symbols(0, 0, circles = 1, inches = FALSE, add = TRUE)
symbols(0, 0, squares = 2, inches = FALSE, add = TRUE)
```

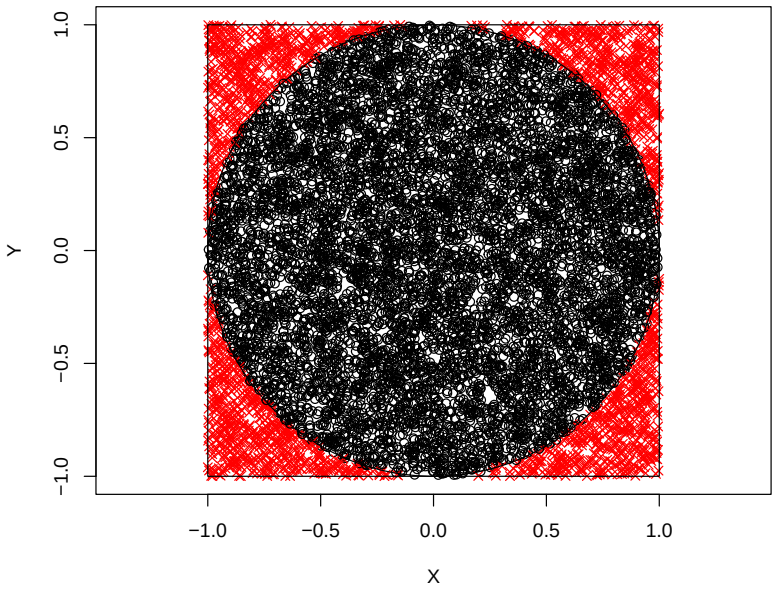


Figura D.1: Valores generados con distribución uniforme bidimensional, con colores y símbolos indicando si están dentro del círculo unidad.

D.1.2 Ejercicio 1.2

Enunciado 1.2:

Consideramos el experimento de Bernoulli consistente en el lanzamiento de una moneda.

a) Empleando la función `sample`, obtener 1000 simulaciones del lanzamiento de una moneda `(0 = cruz, 1 = cara)`, suponiendo que no está trucada. Aproximar la probabilidad de cara a partir de las simulaciones.

```
set.seed(1)
nsim <- 10000
x <- sample(c(cara = 1, cruz = 0), nsim, replace = TRUE, prob = c(0.5,0.5))
mean(x)
```

```
## [1] 0.4953
```

```
barplot(100*table(x)/nsim, ylab = "Porcentaje") # Representar porcentajes
```

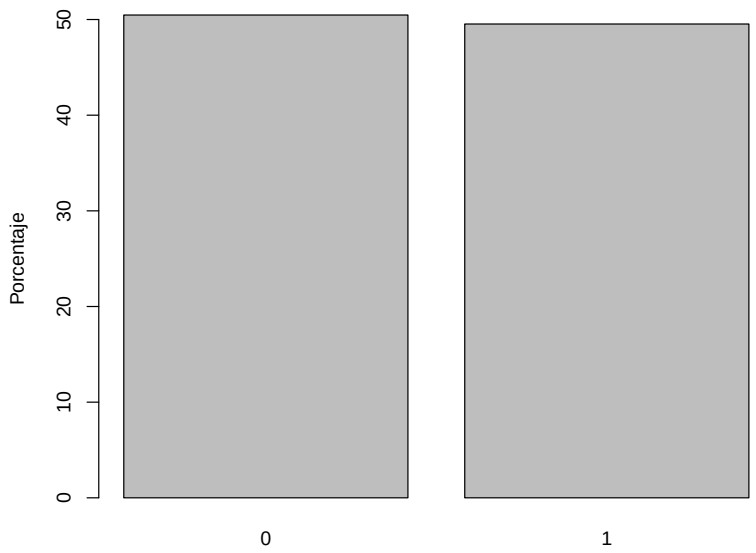


Figura D.2: Frecuencias relativas de los valores generados con distribución Bernoulli (aproximaciones por simulación de las probabilidades teóricas).

---

b) En R pueden generarse valores de la distribución de Bernoulli mediante la función `rbinom(nsim, size=1, prob)`. Generar un gráfico de lineas considerando en el eje X el número de lanzamientos (de 1 a 10000) y en el eje Y la frecuencia relativa del suceso cara (puede ser recomendable emplear la función `cumsum`).

```
set.seed(1)
nsim <- 1000
p <- 0.4
x <- rbinom(nsim, size = 1, prob = p) # Simulamos una Bernoulli
# Alternativa programación: x <- runif(nsim) < p
mean(x)
```

```
## [1] 0.394
```

```
n <- 1:nsim
plot(n, cumsum(x)/n, type="l", ylab="Proporción de caras",
     xlab="Número de lanzamientos", ylim=c(0,1))
abline(h=p, lty=2, col="red")
```

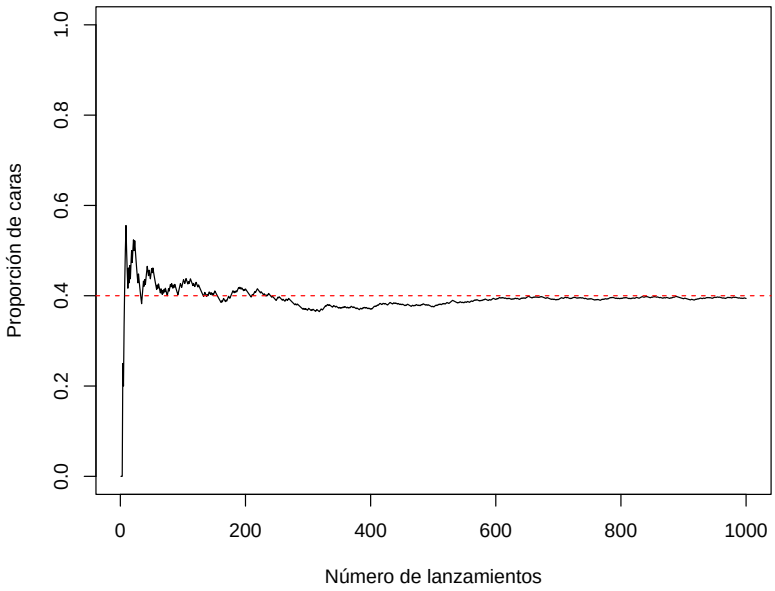
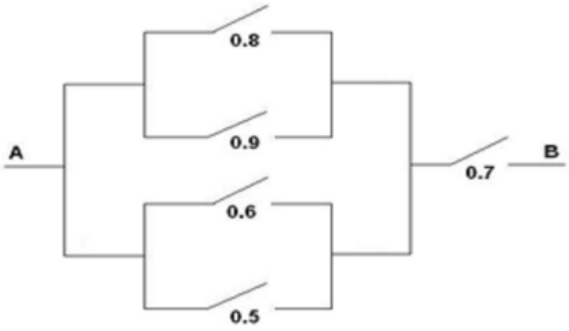


Figura D.3: Gráfico de convergencia de la aproximación por simulación a la probabilidad teórica.

D.1.3 Ejercicio 1.3

Enunciado 1.3:

Simular el paso de corriente a través del siguiente circuito, donde figuran las probabilidades de que pase corriente por cada uno de los interruptores:



Considerar que cada interruptor es una variable aleatoria de Bernoulli independiente para simular 1000 valores de cada una de ellas.

Nota: R maneja internamente los valores lógicos como 1 (`TRUE`) y 0 (`FALSE`). Recíprocamente, cualquier número puede ser tratado como lógico (al estilo de C). El entero 0 es equivalente a `FALSE` y cualquier entero distinto de 0 a `TRUE`.

```
set.seed(1)
nsim <- 10000
x1 <- rbinom(nsim, size=1, prob=0.8)
x2 <- rbinom(nsim, size=1, prob=0.9)
z1 <- x1 | x2   # Operador lógico "O"
x3 <- rbinom(nsim, size=1, prob=0.6)
x4 <- rbinom(nsim, size=1, prob=0.5)
z2 <- x3 | x4
z3 <- z1 | z2
x5 <- rbinom(nsim, size=1, prob=0.7)
fin <- z3 & x5  # Operador lógico "Y"
mean(fin)
```

```
## [1] 0.692
```

D.1.4 Ejercicio 1.4

Enunciado 1.4 (el problema del Caballero de Méré):

En 1651, el Caballero de Méré le planteó a Pascal una pregunta relacionada con las apuestas y los juegos de azar: ¿es ventajoso apostar a que en cuatro lanzamientos de un dado se obtiene al menos un seis? Este problema generó una fructífera correspondencia entre Pascal y Fermat que se considera, simbólicamente, como el nacimiento del Cálculo de Probabilidades.

a) Escribir una función que simule el lanzamiento de n dados. El parámetro de entrada es el número de lanzamientos n, que toma el valor 4 por defecto, y la salida debe ser `TRUE` si se obtiene al menos un 6 y `FALSE` en caso contrario.

```
deMere <- function(n = 4){
  lanz <- sample(1:6, replace=TRUE, size=n)
  return(6 %in% lanz)
}

n <- 4
lanz <- sample(1:6, replace=TRUE, size=n)
lanz
```

```
## [1] 3 5 1 6
```

```
6 %in% lanz
```

```
## [1] TRUE
```

---

b) Utilizar la función anterior para simular $nsim = 10000$ jugadas de este juego y calcular la proporción de veces que se gana la apuesta (obtener al menos un 6 en n lanzamientos), usando $n = 4$. Comparar el resultado con la probabilidad teórica $1 - (5/6)^n$.

```
set.seed(1)
n <- 4
nsim <- 10000
mean(replicate(nsim, deMere(n)))
```

```
## [1] 0.5148
```

```
1-(5/6)^n
```

```
## [1] 0.5177469
```

D.1.5 Ejercicio 1.5

Enunciado 1.5 (variación del problema del coleccionista, cadena de Markov):

Continuando con el ejemplo de la Sección 1.1.1 (álbum con $n = 75$ cromos y sobres con $m = 6$). A partir de $nsim = 2000$ simulaciones de coleccionistas de cromos, aproximar por simulación la evolución del proceso de compra de un coleccionista (número de cromos distintos dependiendo de los sobres comprados).

Generamos `nsim = 2000` simulaciones de coleccionistas de cromos:

```
# Parámetros
n <- 75 # Número total de cromos
m <- 6  # Número de cromos en cada sobre
repe <- TRUE # Repetición de cromos en cada sobre

# Número de simulaciones
nsim <- 2000

# Resultados simulación
nsobres <- numeric(nsim)      # Número de sobres
evol <- vector("list", nsim)  # Evolución del número de cromos
# Por comodidad se podría haber fijado un número máximo de cromos
# evol <- matrix(nrow = max_len, ncol = nsim)

# Fijar semilla
set.seed(1)
# Bucle simulación
for (isim in 1:nsim) {
  # seed <- .Random.seed # .Random.seed <- seed
  # Simular
  album <- logical(n)
  evolucion <- c()
  i <- 0 # Número de sobres
  repeat{
    i <- i + 1
    sobre <- sample(n, m, replace = repe)
    album[sobre] <- TRUE
    ncromos <- sum(album)
    evolucion <- c(evolucion, ncromos)
    if (ncromos == n) {
      nsobres[isim] <- i
      evol[[isim]] <- evolucion
      break
    }
  }
}
# simres::plot.sr(nsobres)
```

`evol` contiene las realizaciones de la cadena de Markov.

```
# plot(evol[[1]], type = "l")
```

Combinar realizaciones del proceso (evoluciones del número de cromos):

```
# Se extienden a la máxima longitud
max_len <- max(lengths(evol)) # max(sapply(evol, length))
evol <- sapply(evol, function(x) c(x, rep(n, max_len - length(x))))
```

```
str(evol)
```

```
##  num [1:167, 1:2000] 6 12 16 21 23 25 30 34 37 38 ...
```

Aproximar cuantiles (intervalos de predicción):

```
alpha <- 0.05
limits <- apply(evol, 1, quantile, probs = c(alpha, 0.5, 1-alpha))
str(limits)
```

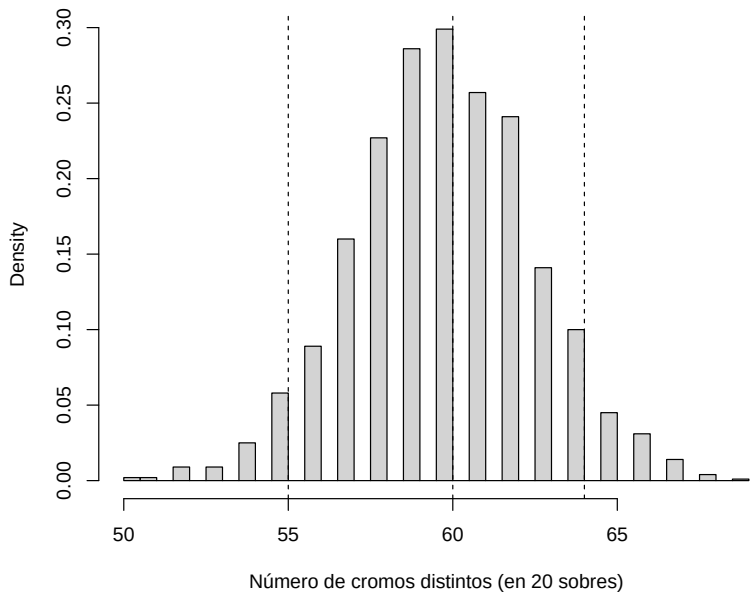
```
##  num [1:3, 1:167] 5 6 6 10 11 12 14 16 18 18 ...
##  - attr(*, "dimnames")=List of 2
##   ..$ : chr [1:3] "5%" "50%" "95%"
##   ..$ : NULL
```

Ejemplo, aproximación de los límites (y mediana) para el número de cromos en el álbum después de comprar 20 sobres:

```
limits[, 20]
```

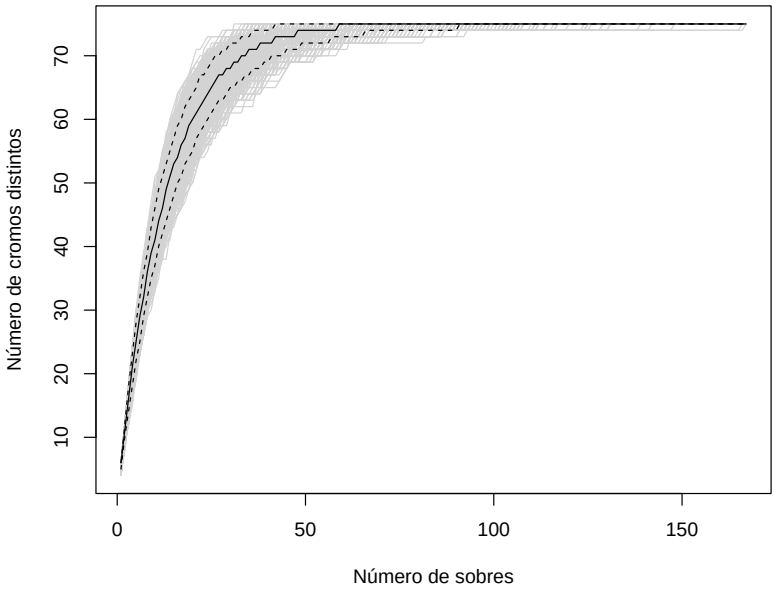
```
##  5% 50% 95%
##  55  60  64
```

```
hist(evol[20, ], breaks = "FD", freq = FALSE,
     main = "", xlab = "Número de cromos distintos (en 20 sobres)")
abline(v = limits[, 20], lty = 2)
```



Representar las realizaciones del proceso y los intervalos de predicción puntuales:

```
matplot(1:max_len, evol, type = "l", col = "lightgray", lty = 1,
    xlab="Número de sobres", ylab="Número de cromos distintos")
matlines(1:max_len, t(limits), lty = c(2, 1, 2), col = 1)
```



D.2 Capítulo 2 Generación de números pseudoaleatorios

D.2.1 Ejercicio 2.1

Enunciado 2.1:

Uno de los primeros generadores utilizados fue el denominado método de los cuadrados medios propuesto por Von Neumann (1946). Con este procedimiento se generan números pseudoaleatorios de 4 dígitos de la siguiente forma:

i. Se escoge un número de cuatro dígitos x_0 (semilla).

ii. Se eleva al cuadrado (x_0^2) y se toman los cuatro dígitos centrales (x_1).

iii. Se genera el número pseudo-aleatorio como

$$u_1 = \frac{x_1}{10^4}.$$

iv. Volver al paso ii y repetir el proceso.

Para obtener los k (número par) dígitos centrales de x_i^2 se puede utilizar que:

$$x_{i+1} = \left\lfloor \left(x_i^2 - \left\lfloor \frac{x_i^2}{10^{(2k-\frac{k}{2})}} \right\rfloor 10^{(2k-\frac{k}{2})} \right) / 10^{\frac{k}{2}} \right\rfloor$$

Este algoritmo está implementado en la función `simres::rvng()` (ver también `simres::rng()`; fichero *rng.R*):

```
simres::rvng
```

```
## function(n, seed = as.numeric(Sys.time()), k = 4) {
##   seed <- seed %% 10^k
##   aux <- 10^(2*k-k/2)
##   aux2 <- 10^(k/2)
##   u <- numeric(n)
##   for(i in 1:n) {
```

```
##      z <- seed^2
##      seed <- trunc((z - trunc(z/aux)*aux)/aux2)
##      u[i] <- seed/10^k
##    }
##    # Almacenar semilla y parámetros
##    assign(".rng", list(seed = seed, type = "vm", parameters = list(k = k)),
##        envir = globalenv())
##    # .rng <<- list(seed = seed, type = "vm", parameters = list(k = k))
##    # Para continuar con semilla y parámetros:
##    #   with(.rng, rvng(n, seed, parameters$k))
##    # Devolver valores
##    return(u)
## }
## <bytecode: 0x000000003cf95ca8>
## <environment: namespace:simres>
```

Estudiar las características del generador de cuadrados medios a partir de una secuencia de 500 valores. Emplear únicamente métodos gráficos.

D.3 Capítulo 5 Simulación de variables discretas

D.3.1 Ejercicio 5.1

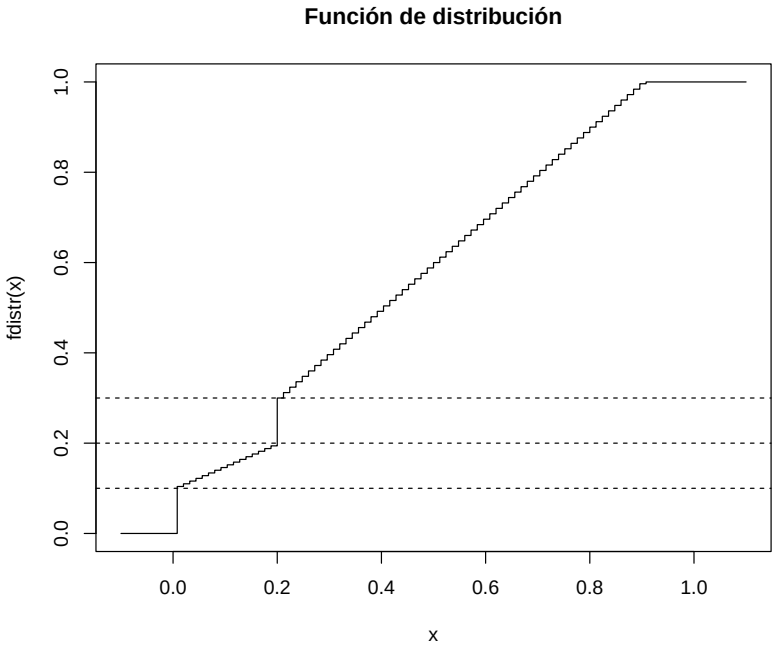
Enunciado 5.1 (Simulación de una distribución mixta mediante el método de inversión generalizado):

Consideramos la variable aleatoria con función de distribución dada por:

$$F(x) = \begin{cases} 0 & \text{si } x < 0 \\ \frac{x}{2} + \frac{1}{10} & \text{si } x \in [0, \frac{1}{5}) \\ x + \frac{1}{10} & \text{si } x \in [\frac{1}{5}, \frac{9}{10}] \\ 1 & \text{en otro caso} \end{cases}$$

Esta función está implementada en el siguiente código:

```r
fdistr <- function(x) {
ifelse(x < 0, 0,
    ifelse(x < 1/5, x/2 + 1/10,
        ifelse(x <= 9/10, x + 1/10, 1) ) )
}
# Empleando ifelse la función es vectorial (y podemos emplear curve...)
curve(fdistr, from = -0.1, to = 1.1, type = 's',
      main = 'Función de distribución')
# Discontinuidades en 0 y 1/5
abline(h = c(1/10, 2/10, 3/10), lty = 2)
```



Nota: Esta variable toma los valores 0 y 1/5 con probabilidad 1/10.

a) Diseñar un algoritmo basado en el método de inversión generalizado para generar observaciones de esta variable.

b) Implementar el algoritmo en una función que permita generar *nsim* valores de esta variable.

---

a) El algoritmo general es siempre el mismo. Empleando la función cuantil:

$$Q(u) = \inf\{x \in \mathbb{R} : F(x) \geq u\},$$

el algoritmo sería:

1. Generar $U \sim \mathcal{U}(0,1)$
2. Devolver $X = Q(U)$

En este caso concreto:

1. Generar $U \sim \mathcal{U}(0,1)$
2. Si $U < \frac{1}{10}$ devolver $X = 0$
3. Si $U < \frac{2}{10}$ devolver $X = 2(U - \frac{1}{10})$
4. Si $U < \frac{3}{10}$ devolver $X = \frac{2}{10}$
5. En caso contrario devolver $X = U - \frac{1}{10}$

b) El algoritmo de simulación se puede implementar a partir de la función cuantil (vectorial):

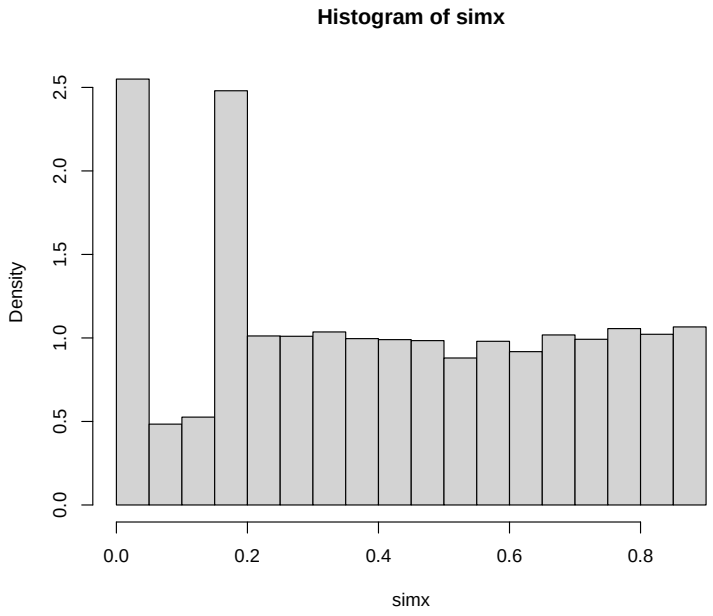
```
# Función cuantil:
fquant <- function(u)
  ifelse(u < 1/10, 0,
         ifelse(u < 2/10, 2*(u - 1/10),
                ifelse(u < 3/10, 1/5, u - 1/10) ) )
# Función para generar nsim valores:
rx <- function(nsim) fquant(runif(nsim))
```

Ejemplo:

```
set.seed(1)
nsim <- 10^4
system.time(simx <- rx(nsim))
```

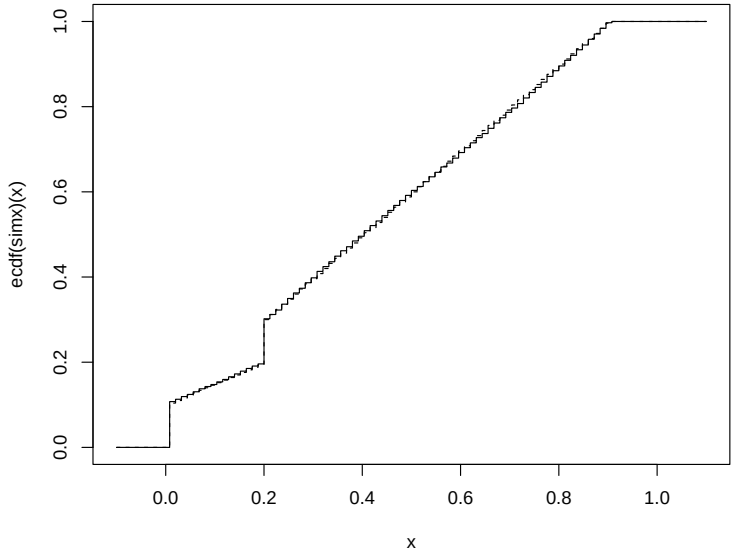
```
##    user  system elapsed
##       0       0       0
```

```
hist(simx, breaks = "FD", freq = FALSE)
```



En este caso como no es una variable absolutamente continua mejor emplear la función de distribución para compararla con la teórica:

```
curve(ecdf(simx)(x), from= -0.1, to = 1.1, type = "s")
curve(fdistr(x), type = "s", lty = 2, add = TRUE)
```


ecdf(simx)(x)
0.0
0.2
0.4
0.6
0.8
1.0
0.0
0.2
0.4
0.6
0.8
1.0
x

Referencias

Bibliografía básica

Cao, R. (2002). *Introducción a la simulación y a la teoría de colas.* NetBiblo.

Cao R. y Fernández-Casal R. (2020). *Técnicas de Remuestreo.* https://rubenfcasal.github.io/book_remuestreo.

Gentle, J.E. (2003). *Random number generation and Monte Carlo methods.* Springer-Verlag.

Jones, O. et al. (2009). *Introduction to Scientific Programming and Simulation Using R.* CRC.

Ripley, B.D. (1987). *Stochastic Simulation.* John Wiley & Sons.

Robert, C.P. y G. Casella (2010). *Introducing Monte Carlo Methods with R.* Springer.

Ross, S.M. (1999).*Simulación.* Prentice Hall.

Suess, E.A. y Trumbo, B.E. (2010). *Introduction to probability simulation and Gibbs sampling with R.* Springer.

Bibliografía complementaria

Bratley, P., Fox, B.L. y Schrage L.E. (1987). *A guide to simulation.* Springer-Verlag.

Davison, A.C. y Hinkley, D.V. (1997). *Bootstrap Methods and Their Application.* Cambridge University Press.

Devroye, L. (1986). *Non-uniform random variate generation.* Springer-Verlag.

Evans, M. y Swartz, T. (2000). *Approximating integrals via Monte Carlo and determinstic methods.* Oxford University Press.

Gentle, J.E. (1998). *Random number generation and Monte Carlo methods.* Springer-Verlag.

Hörmann, W. et al. (2004). *Automatic Nonuniform Random Variate Generation.* Springer.

Moeschlin, O., Grycko, E., Pohl, C. y Steinert, F. (1998). *Experimental stochastics.* Springer-Verlag.

Pardo, L. y Valdés, T. (1987). *Simulación. Aplicaciones prácticas a la empresa.* Díaz de Santos.

Robert, C.P. y G. Casella (2004). *Monte Carlo statistical methods.* Springer. Shao, J. (2003). *Mathematical statistics.* Springer.

Enlaces

Repositorio: rubenfcasal/simbook2

Recursos para el aprendizaje de R: En este post se muestran algunos recursos que pueden ser útiles para el aprendizaje de R y la obtención de ayuda.

Bookdown:

- Cao R. y Fernández-Casal R. (2021). *Técnicas de Remuestreo.* github.

- Introducción a RMarkdown, apéndice del libro: Fernández-Casal R. y Cotos-Yáñez T.R. (2018). *Escritura de libros con bookdown*, github.
- Hyndman, R.J., y Athanasopoulos, G. (2021). *Forecasting: principles and practice.* OTexts.
- Wickham, H. (2019). *Advanced R, 2ª edición*, Chapman & Hall, 1ª edición..
- Wickham, H. (2015). *R packages: organize, test, document, and share your code* (actualmente 2ª edición en desarrollo con H. Bryan), O'Reilly, 1ª edición.

Bibliografía completa

Azarang, M. R. y García Dunna, E. (1996). *Simulación y análisis de modelos estocásticos.* McGraw-Hill.

Demirhan, H. y Bitirim, N. (2016). CryptRndTest: an R package for testing the cryptographic randomness. *The R Journal*, 8(1), 233-247.

Downham, D.Y. (1970). Algorithm AS 29: The runs up and down test. *Journal of the Royal Statistical Society.* Series C (Applied Statistics), 19(2), 190-192.

Gilks, W.R. y Wild, P. (1992). Adaptive rejection sampling for Gibbs sampling. *Journal of the Royal Statistical Society.* Series C (Applied Statistics), 41(2), 337-348.

Hall, S.W. (1994). Analysis of defectivity of semiconductor wafers by contingency table, *Proceedings of the Institute of Environmental Sciences*, **1**, 177-183.

Hyndman, R.J. y Athanasopoulos, G. (2018). *Forecasting: principles and practice.* OTexts. Disponible online: 2nd edition (forecast), third edition (fable).

Hofert, M. (2018). *Elements of Copula Modeling with R*, Springer.

Kinderman, A.J. y Monahan, J.F. (1977). Computer generation of random variables using the ratio of uniform deviates. *ACM Transactions on Mathematical Software (TOMS)*, 3(3), 257-260.

Knuth, D.E. (1969). *The Art of Computer Programming.* Volume 2. Addison-Wesley.

Knuth, D.E. (2002). *The Art of Computer Programming.* Volume 2, third edition, ninth printing. Addison-Wesley.

L'Ecuyer, P. (1999). Good parameters and implementations for combined multiple recursive random number generators. *Operations Research*, 47, 159–164.

L'Ecuyer, P. y Simard, R. (2007). TestU01: A C library for empirical testing of random number generators. *ACM Transactions on Mathematical Software* (TOMS), 33(4), 1-40.

Law, A.M. y Kelton, W.D. (1991). *Simulation, modeling and analysis.* McGraw-Hill.

Liu, J.S. (2004). *Monte Carlo strategies in scientific computing.* Springer.

Marsaglia, G. y Tsang, W.W. (2002). Some difficult-to-pass tests of randomness. *Journal of Statistical Software*, 7(3), 1-9.

Marsaglia, G., Zaman, A. y Tsang, W.W. (1990). Toward a universal random number generator. *Stat. Prob. Lett.*, 9(1), 35-39.

Matsumoto, M. y Nishimura, T. (1998). Mersenne Twister: A 623-dimensionally equidistributed uniform pseudo-random number generator, *ACM Transactions on Modeling and Computer Simulation*, 8, 3–30.

Nelsen, R.B. (2006). *An introduction to copulas*, second edition, Springer.

Nelson, R. (1995). *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modelling.* Springer-Verlag.

Odeh, R.E. y Evans, J.O. (1974). The percentage points of the normal distribution. *Journal of the Royal Statistical Society.* Series C (Applied Statistics), 23(1), 96-97.

Patefield, W.M. (1981). Algorithm AS 159: An efficient method of generating random r x c tables with given row and column totals. *Applied Statistics*, 30, 91–97.

Park, S.K. y Miller , K.W. (1988). Random number generators: good ones are hard to find. *Communications of the ACM*, 31(10), 1192-1201.

Park, S.K., Miller, K.W. y Stockmeyer, P.K. (1993). Technical correspondence. *Communications of the ACM*, 36(7), 108-110.

Wichura, M.J. (1988) Algorithm AS 241: The Percentage Points of the Normal Distribution. *Applied Statistics*, 37, 477–484.