

Simulación Estadística

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

2022-09-05

Índice general

Prólogo	5
1 Introducción a la simulación	7
1.1 Conceptos básicos	7
1.2 Aplicaciones de la simulación	9
1.3 Números aleatorios puros	9
1.4 Generación de números “aleatorios” mediante software	10
1.5 Números pseudoaleatorios	11
2 Números aleatorios en R	13
2.1 Opciones	14
2.2 Paquetes de R	14
2.3 Ejercicios	15
2.4 Tiempo de CPU	20
3 Generación de números pseudoaleatorios	23
3.1 Generadores congruenciales lineales	23
3.2 Extensiones	27
3.3 Análisis de la calidad de un generador	28
3.4 Ejercicios	38
4 Análisis de resultados de simulación	41
4.1 Convergencia	41
4.2 Estimación de la precisión	42
4.3 Teorema central del límite	44
4.4 Determinación del número de generaciones	45
4.5 El problema de la dependencia	46
4.6 Observaciones	54
5 Simulación de variables continuas	55
5.1 Método de inversión	55
5.2 Método de aceptación rechazo	59
5.3 Modificaciones del método de aceptación-rechazo	72
5.4 Método de composición (o de simulación condicional)	76
5.5 Métodos específicos para la generación de algunas distribuciones notables	77
6 Simulación de variables discretas	81
6.1 Método de la transformación cuantil	81
6.2 Método de la tabla guía	86
6.3 Método de Alias	88
6.4 Simulación de una variable discreta con dominio infinito	91
6.5 Cálculo directo de la función cuantil	92
6.6 Otros métodos	93
6.7 Métodos específicos para generación de distribuciones notables	96

7 Simulación de distribuciones multivariantes	97
7.1 Simulación de componentes independientes	97
7.2 El método de aceptación/rechazo	99
7.3 Factorización de la matriz de covarianzas	101
7.4 Método de las distribuciones condicionadas	103
7.5 Simulación condicional e incondicional	106
7.6 Simulación basada en cópulas	113
7.7 Simulación de distribuciones multivariantes discretas	118
8 Aplicaciones en Inferencia Estadística	125
8.1 Distribución en el muestreo	126
8.2 Intervalos de confianza	128
8.3 Contrastes de hipótesis	136
8.4 Comparación de estimadores	144
8.5 Remuestreo Bootstrap	148
9 Integración y Optimización Monte Carlo	157
9.1 Integración Monte Carlo (clásica)	157
9.2 Muestreo por importancia	161
9.3 Optimización Monte Carlo	167
9.4 Temple simulado	170
9.5 Algoritmos genéticos	173
10 Técnicas de reducción de la varianza	177
10.1 Variables antitéticas	177
10.2 Estratificación	182
10.3 Variables de control	185
10.4 Números aleatorios comunes	186
10.5 Ejercicios	187
Referencias	189
Bibliografía básica	189
Bibliografía complementaria	189
A Enlaces	191
B Bondad de Ajuste y Aleatoriedad	195
B.1 Métodos de bondad de ajuste	195
B.2 Diagnóstico de la independencia	204
B.3 Contrastados específicos para generadores aleatorios	213
C Integración numérica	217
C.1 Integración numérica unidimensional	217
C.2 Integración numérica bidimensional	220

Prólogo

IMPORTANTE: Este libro es la “primera edición” y puede no estar actualizado. Se está elaborando una **nueva edición** disponible en rubenfcasal.github.io/simbook2 (que se corresponde con el repositorio rubenfcasal/simbook2).

Este libro contiene los apuntes de la asignatura de Simulación Estadística del Máster en Técnicas Estadísticas.

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

<https://rubenfcasal.github.io/simbook/index.html>.

donde puede descargarse en formato pdf.

Para instalar los paquetes necesarios para poder ejecutar los ejemplos mostrados en el libro se puede emplear el siguiente comando:

```
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.



Este obra está bajo una licencia de Creative Commons Reconocimiento-NoComercial-SinObraDerivada 4.0 Internacional (esperamos poder liberarlo bajo una licencia menos restrictiva más adelante...).

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 9 se tratarán métodos de integración y optimización Monte Carlo.

1.1.1 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 destrutiva.
- 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 inconveniente 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.

¹Como 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.

²Estos 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.

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 extraer las conclusiones a otras situaciones.

1.2 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 8 y 9 nos centraremos en algunas de las aplicaciones de utilidad en Estadística.

1.3 Números aleatorios puros

El primer requisito para poder realizar simulación estocástica sería disponer de números aleatorios. Una sucesión de números aleatorios puros (*true random*) se caracteriza porque no existe ninguna regla o plan que nos permita conocer sus valores.

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 1.1).

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 1.1: 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 avanzando 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 inconvenientes. 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.

1.4 Generación de números “aleatorios” mediante software

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 las tablas. Se distingue entre dos tipos de secuencias:

- *números pseudo-aleatorios*: simulan realizaciones de una variable aleatoria (uniforme),
- *números cuasi-aleatorios*: secuencias deterministas con una distribución más regular en el rango considerado.

Algunos problemas, como la integración numérica (en el Capítulo 9 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 1.2).

```
library(randtoolbox)
n <- 2000
par.old <- par( mfrw=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')
```

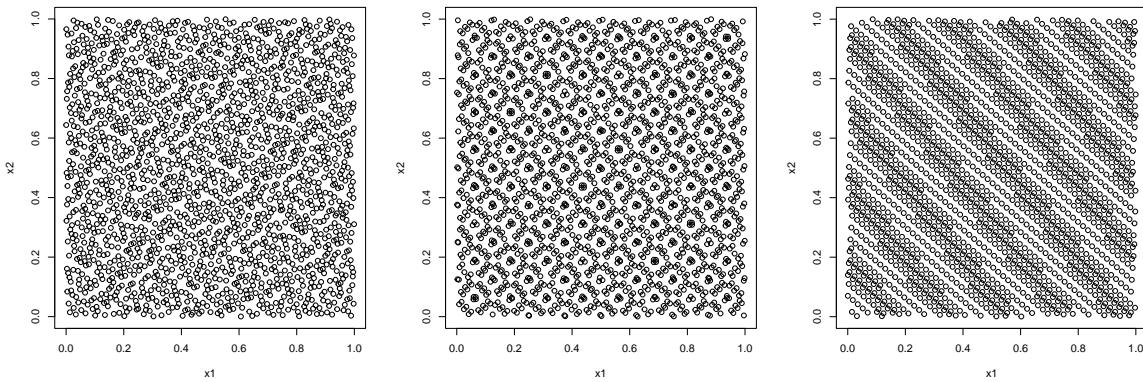


Figura 1.2: Secuencias cuasi-aleatorias bidimensionales obtenidas con los métodos de Halton (izquierda), Sobol (centro) y Torus (derecha).

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

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

1.5 Números pseudoaleatorios

La mayoría de los métodos de simulación se basan en la posibilidad de generar números pseudoaleatorios que imiten las propiedades de valores independientes de una distribución $\mathcal{U}(0, 1)$, es decir, que imiten las propiedades de una muestra aleatoria simple³.

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

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

donde:

- k es el orden del generador.
- $(x_0, x_1, \dots, x_{k-1})$ 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.

³Aunque 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).

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).
- 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 3.1. 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.

Capítulo 2

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 6).
- `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 `globalenv`) en `.Random.seed`; es un vector de enteros cuya dimensión depende del tipo de generador:

- No debe ser modificado manualmente; se guarda con el entorno de trabajo (por ejemplo, si se guarda al terminar la sesión en un fichero `.RData`, se restaurará la semilla al iniciar una nueva sesión y podremos continuar con las simulaciones).
- Si no se especifica con `set.seed` (o no existe) se genera a partir del reloj del sistema.
- Puede ser recomendable almacenarla 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:n sim) {
  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
...
}
```

2.1 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 ligeramente¹ 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.

2.2 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.

¹Para evitar problemas de redondeo con tamaños extremadamente grandes; ver bug PR#17494.

- `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`, `rngwell119937`, `randaes`, `SuppDists`, `lhs`, `mc2d`, `fOptions`, ...

2.3 Ejercicios

Ejercicio 2.1

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 2.1) (los puntos con color negro tienen distribución uniforme en el círculo unidad; esto está relacionado con el método de aceptación-rechazo, ver Ejemplo 7.3, 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 círculo
symbols(0, 0, circles = 1, inches = FALSE, add = TRUE)
symbols(0, 0, squares = 2, inches = FALSE, add = TRUE)
```

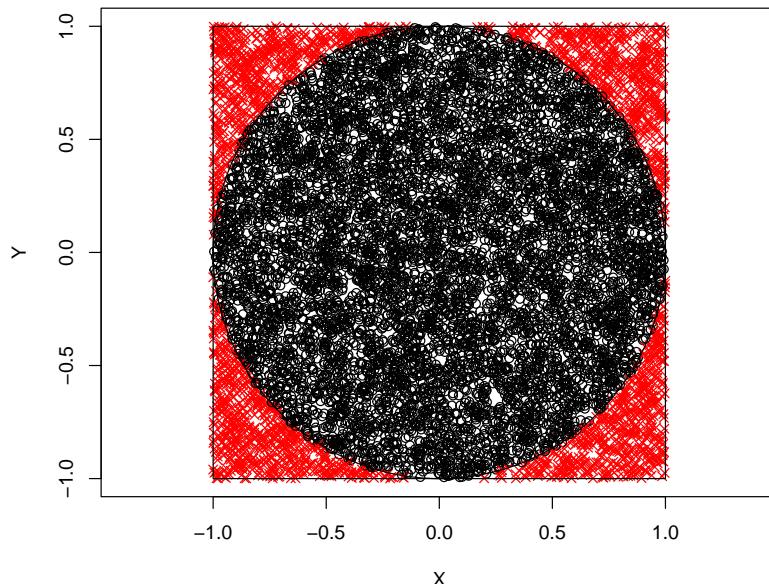


Figura 2.1: Valores generados con distribución uniforme bidimensional, con colores y símbolos indicando si están dentro del círculo unidad.

Ejercicio 2.2 (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.

```
set.seed(1)
nsim <- 1000
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
```

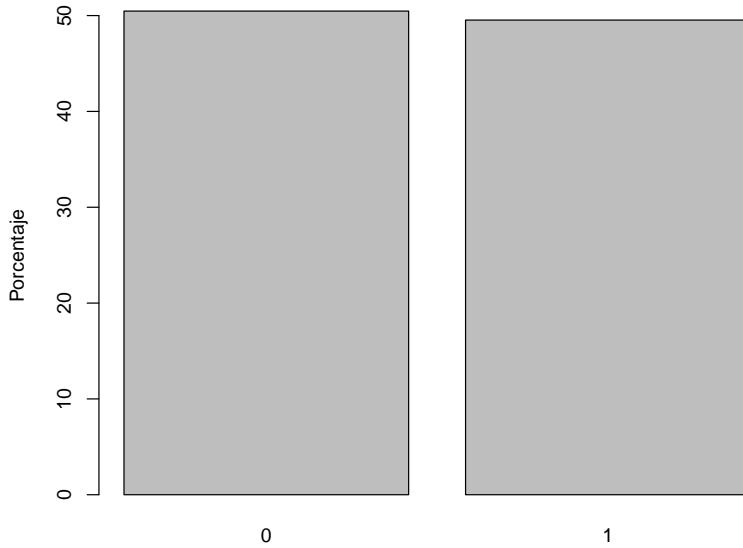


Figura 2.2: Frecuencias relativas de los valores generados con distribución Bernoulli (aproximaciones por simulación de las probabilidades teóricas).

- b) En R pueden generarse valores de la distribución de Bernoulli mediante la función `rbinom(nsim, size=1, prob)`. Generar un gráfico de líneas 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")
```

Ejercicio 2.3 (Simulación de un circuito)
 Simular el paso de corriente a través del siguiente circuito, donde figuran las probabilidades de que pase corriente por cada uno de los interruptores:

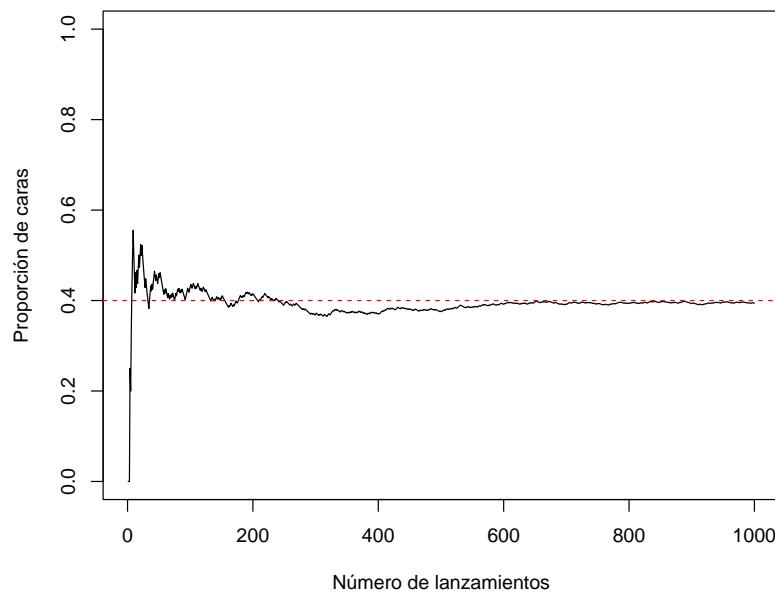
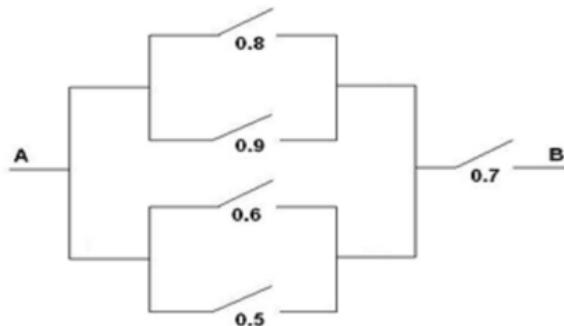


Figura 2.3: Gráfico de convergencia de la aproximación por simulación a la probabilidad teórica.



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
```

Ejercicio 2.4 (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
```

Ejercicio 2.5 (variación del problema del coleccionista)

Supongamos que tenemos un álbum con $n = 75$ cromos y para completarlo hay que comprar sobres con $m = 6$ cromos. A partir de $nsim = 1000$ simulaciones de coleccionistas de cromos, obtener una aproximación por simulación a la respuesta de las siguientes cuestiones:

- a) ¿Cuál será en media el número de sobres que hay que comprar para completar la colección?
 ¿Cuál sería el gasto medio si cada sobre cuesta 0.8€?
- b) ¿Cuál sería el número mínimo de sobres para asegurar de que se completa la colección un 95% de las veces?

2.4 Tiempo de CPU

La velocidad del generador suele ser una característica importante (también medir los tiempos, de cada iteración y de cada procedimiento, 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()
  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
##   0.14    0.00   0.14
## Tiempo total operación:
##   user  system elapsed
##   0.14    0.00   0.14
result2 <- funtest(10^3)
CPUTimeprint()
```

```
## Tiempo última operación:
##   user  system elapsed
##   0.17    0.00   0.17
## Tiempo total operación:
##   user  system elapsed
##   0.31    0.00   0.31
```

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()`.
- La función `cpu.time()` del paquete `npsp`:
 - `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).

```
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: 0.01 sec elapsed
toc() # middle

## middle: 0.01 sec elapsed
toc() # outer

## outer: 0.11 sec elapsed
## Timing in a loop and analyzing the results later using tic.log().
tic.clearlog()
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 4.9 4.9 4.9 4.9 4.9 4.9 4.9 4.9 4.9 4.92
## $ toc: num 4.9 4.9 4.9 4.9 4.9 4.9 4.9 4.9 4.92 4.92
## $ msg: chr "1" "2" "3" "4" ...
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.
## 0.000 0.000 0.000 0.002 0.000 0.020
```

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 qué 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`).

Capítulo 3

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.

3.1 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:

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

donde a (*multiplicador*), c (*incremento*) y m (*módulo*) son enteros positivos¹ 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 en el siguiente código, imitando el funcionamiento del generador uniforme de R (aunque de un forma no muy eficiente²):

```
# -----
# Generador congruencial de números pseudoaleatorios
# -----
```

¹Se 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.

²Para evitar problemas computacionales, se recomienda realizar el cálculo de los valores empleando el método de Schrage (ver Bratley *et al.*, 1987; L'Ecuyer, 1988).

```

# initRANDC(semilla, a, c, m)
# -----
#   Selecciona los parámetros del generador congruencial
#   Por defecto RANDU de IBM con semilla del reloj
#   No se hace ninguna verificación de los parámetros
initRANDC <- function(semilla=as.numeric(Sys.time()), a=2^16+3, c=0, m=2^31) {
  .semilla <- as.double(semilla) %% m  #Cálculos en doble precisión
  .a <- a
  .c <- c
  .m <- m
  return(invisible(list(semilla = .semilla, a = .a, c = .c, m = .m))) #print(initRANDC())
}

# RANDC()
# -----
#   Genera un valor pseudoaleatorio con el generador congruencial
#   Actualiza la semilla (si no existe llama a initRANDC)
RANDC <- function() {
  if (!exists(".semilla", envir = globalenv())) initRANDC()
  .semilla <- (.a * .semilla + .c) %% .m
  return(.semilla/.m)
}

# RANDCN(n)
# -----
#   Genera un vector de valores pseudoaleatorios con el generador congruencial
#   (por defecto de dimensión 1000)
#   Actualiza la semilla (si no existe llama a initRANDC)
RANDCN <- function(n = 1000) {
  x <- numeric(n)
  for(i in 1:n) x[i] <- RANDC()
  return(x)
  # return(replicate(n,RANDC())) # Alternativa más rápida
}

initRANDC(543210)      # Fijar semilla 543210 para reproducibilidad

```

Ejemplos:

- $c = 0, a = 2^{16} + 3 = 65539$ 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 3.1 (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. $a \equiv 1 \pmod{q}$, para todo q factor primo de m).
 3. Si m es múltiplo de 4, entonces $a - 1$ también lo ha de ser (i.e. $m \equiv 0 \pmod{4} \Rightarrow a \equiv 1 \pmod{4}$).
-

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 3.2

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

1. m es primo.
 2. a es una raíz primitiva de m (i.e. el menor entero q tal que $a^q \equiv 1 \pmod{m}$ 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 3.1)³. Por ejemplo, el conjunto de datos `randu` contiene 400 tripletas de números sucesivos obtenidos con la implementación de VAX/VMS 1.5 (1977).

```
system.time(u <- RANDCN(9999))

##    user  system elapsed
##  0.02    0.00    0.01

# 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)
```

En general todos los generadores de este tipo van a presentar estructuras reticulares. Marsaglia (1968) demostró que las k -uplas de un generador 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 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 3.3) 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.

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

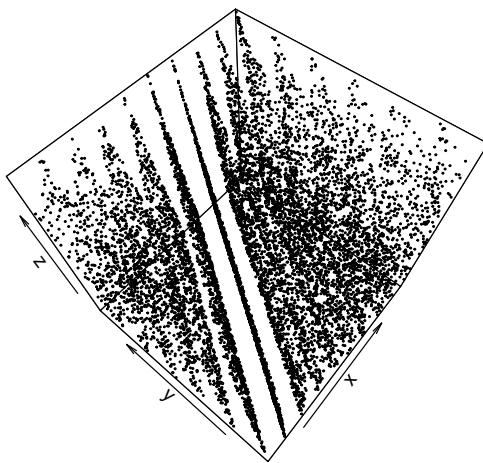


Figura 3.1: Grafico de dispersión de tripletas del generador RANDU de IBM (contenidas en 15 planos).

- 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 3.1

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

$$\begin{aligned}x_{n+1} &= (5x_n + 1) \bmod 512, \\ u_{n+1} &= \frac{x_{n+1}}{512}, \quad n = 0, 1, \dots\end{aligned}$$

- 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.

```
initRANDC(321, 5, 1, 512)           # Establecer semilla y parámetros
nsim <- 500
system.time(u <- RANDCN(nsim))
```

```
##      user    system elapsed
##        0        0       0
hist(u, freq = FALSE)               # Densidad uniforme
abline(h = 1)
```

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 3.2 en Sección 3.3).

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

La aproximación por simulación de la media teórica es:

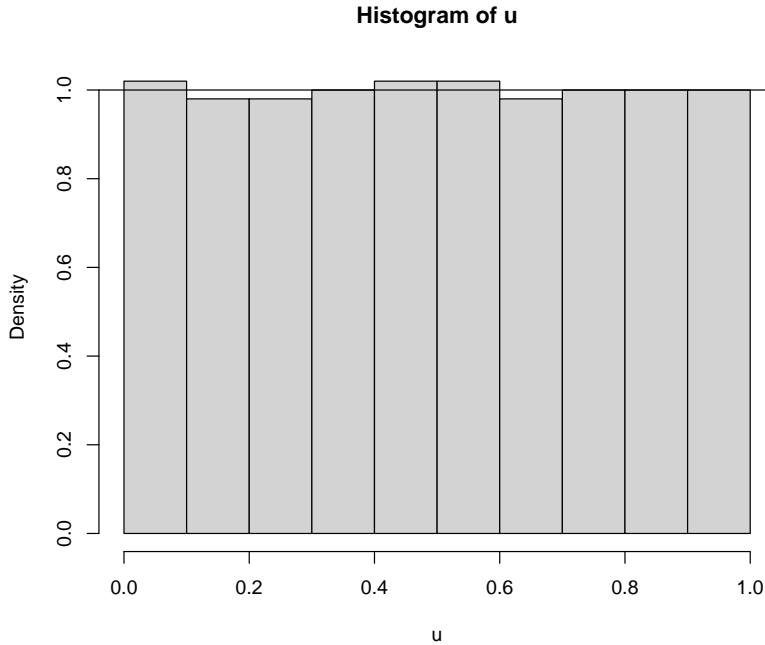


Figura 3.2: Histograma de los valores generados.

```
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
```

3.2 Extensiones

Se han considerado diversas extensiones del generador congruencial lineal simple:

- Lineal múltiple: $x_i = a_0 + a_1x_{i-1} + a_2x_{i-2} + \dots + a_kx_{i-k} \pmod{m}$, con periodo $p \leq m^k - 1$.
- No lineal: $x_i = f(x_{i-1}, x_{i-2}, \dots, x_{i-k}) \pmod{m}$. Por ejemplo $x_i = a_0 + a_1x_{i-1} + a_2x_{i-1}^2 \pmod{m}$.
- Matricial: $x_i = A_0 + A_1x_{i-1} + A_2x_{i-2} + \dots + A_kx_{i-k} \pmod{m}$.

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

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

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 combinación 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 concuenciales lineales múltiples de orden $k = 3$ (el período aproximado es 2^{191}).

3.3 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 B 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 3.3.2.

Hay que destacar algunas diferencias entre el uso de este tipo de métodos en inferencia y en simulación. Por ejemplo, si empleamos un condraste 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 siguiente función (que imita a las incluidas en R):

```
#-----
# chisq.test.cont(x, distribution, nclasses, output, nestpar,...)
#-----
# Realiza el test chi-cuadrado de bondad de ajuste para una distribución
# continua discretizando en intervalos equiprobables.
# Parámetros:
#   distribution = "norm", "unif", etc
#   nclasses = floor(length(x)/5)
#   output = TRUE
#   nestpar = 0 = número de parámetros estimados
#   ... = parámetros distribución
# Ejemplo:
#   chisq.test.cont(x, distribution = "norm", nestpar = 2,
#                   mean = mean(x), sd = sqrt((nx - 1) / nx) * sd(x))
#-----

chisq.test.cont <- function(x, distribution = "norm",
  nclasses = floor(length(x)/5), output = TRUE, nestpar = 0, ...) {
  # Funciones distribución
  q.distrib <- eval(parse(text = paste("q", distribution, sep = "")))
  d.distrib <- eval(parse(text = paste("d", distribution, sep = "")))
  # Puntos de corte
  q <- q.distrib((1:(nclasses - 1))/nclasses, ...)
  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")
    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)/nclasses
DNAME <- deparse(substitute(x))
METHOD <- "Pearson's Chi-squared test"
STATISTIC <- sum((O - E)^2/E)
names(STATISTIC) <- "X-squared"
PARAMETER <- nclasses - nestpar - 1
names(PARAMETER) <- "df"
PVAL <- pchisq(STATISTIC, PARAMETER, lower.tail = FALSE)
# Preparar resultados
classes <- format(xbreaks)
classes <- paste("(, classes[-(nclasses + 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")
}

```

Ejemplo 3.2 (análisis de un generador congruencial, continuación)

Continuando con el generador congruencial del Ejemplo 3.1:

```

initRANDC(321, 5, 1, 512)
nsim <- 500
system.time(u <- RANDCN(nsim))

```

```

##    user  system elapsed
##      0       0       0

```

Al aplicar el test chi-cuadrado obtendríamos:

```

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

```

```

##
## 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

```

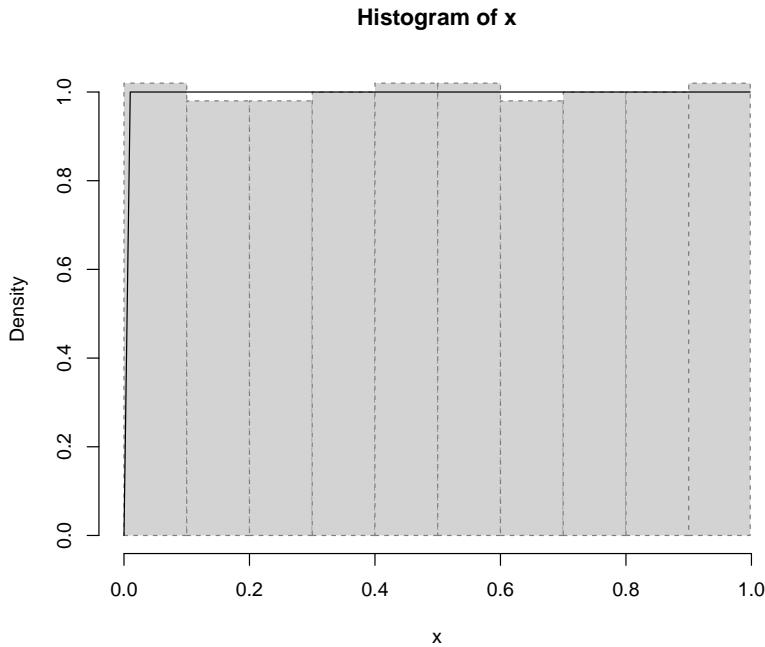


Figura 3.3: Gráfico resultante de aplicar la función ‘chisq.test.cont()’ comparando el histograma de los valores generados con la densidad uniforme.

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

Como se muestra en la Figura 3.3 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 B.1.5). Este contraste de hipótesis compara la función de distribución bajo la hipótesis nula con la función de distribución empírica (ver Sección B.1.2), representadas en la Figura 3.4:

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

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 B.1 se describen con más detalle estos contrastes de bondad de ajuste.

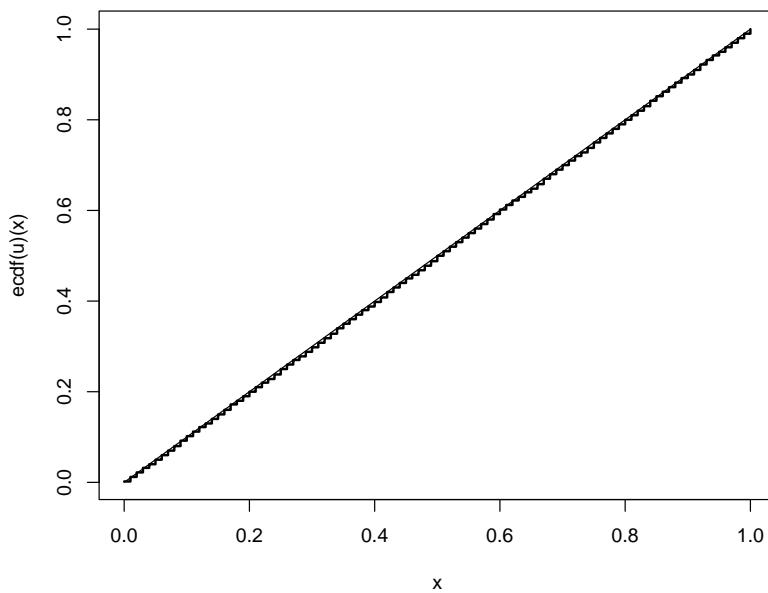


Figura 3.4: Comparación de la distribución empírica de la secuencia generada con la función de distribución uniforme.

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

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

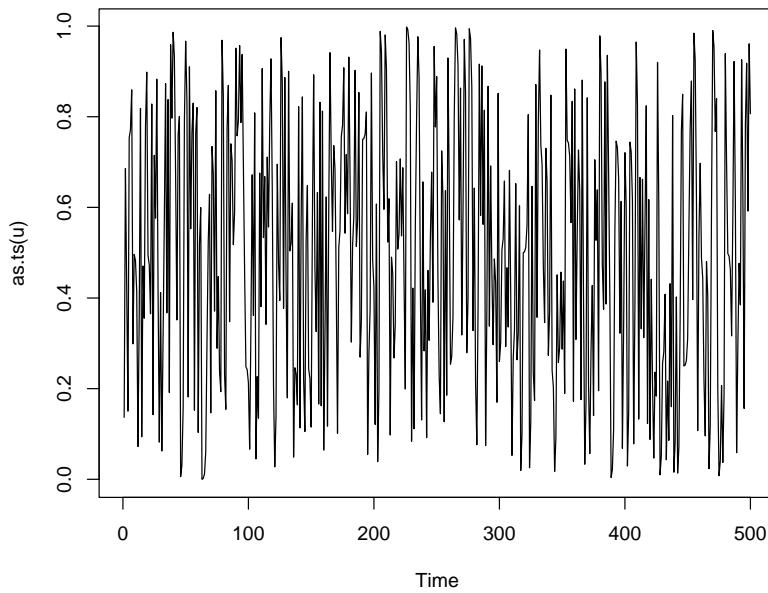


Figura 3.5: Gráfico secuencial de los valores generados.

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

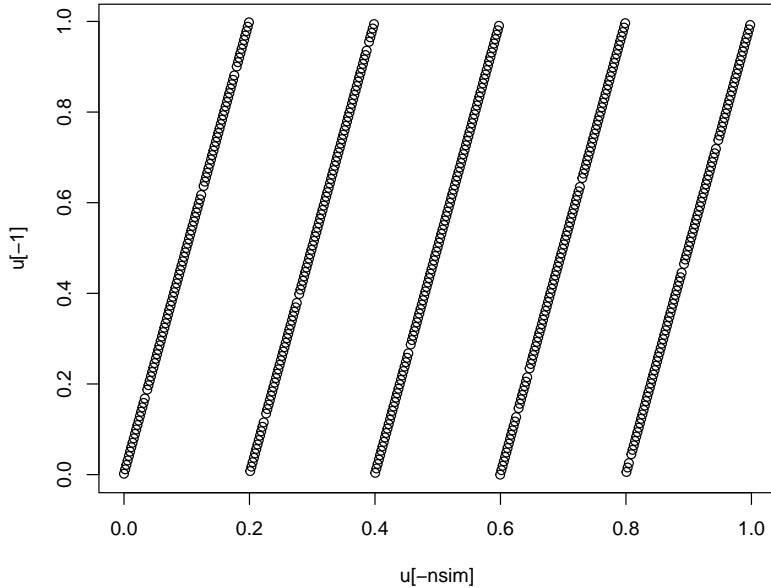


Figura 3.6: Gráfico de dispersión retardado de los valores generados.

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

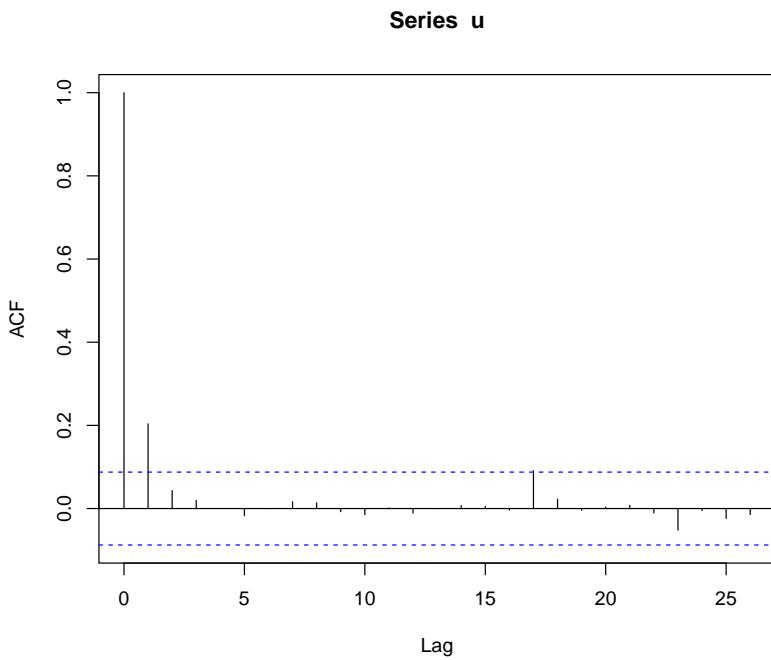


Figura 3.7: Autocorrelaciones de los valores generados.

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

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
```

3.3.1 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 8.3).

Ejemplo 3.3

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

```
# Valores iniciales
initRANDC(543210)    # Fijar semilla para reproducibilidad
# set.seed(543210)
n <- 500
nsim <- 1000
estadistico <- numeric(nsim)
pvalor <- numeric(nsim)

# Realizar contrastes
for(isim in 1:nsim) {
  u <- RANDCN(n)      # Generar
  # u <- runif(n)
  tmp <- chisq.test.cont(u, distribution="unif",
                         nclasses=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)
```

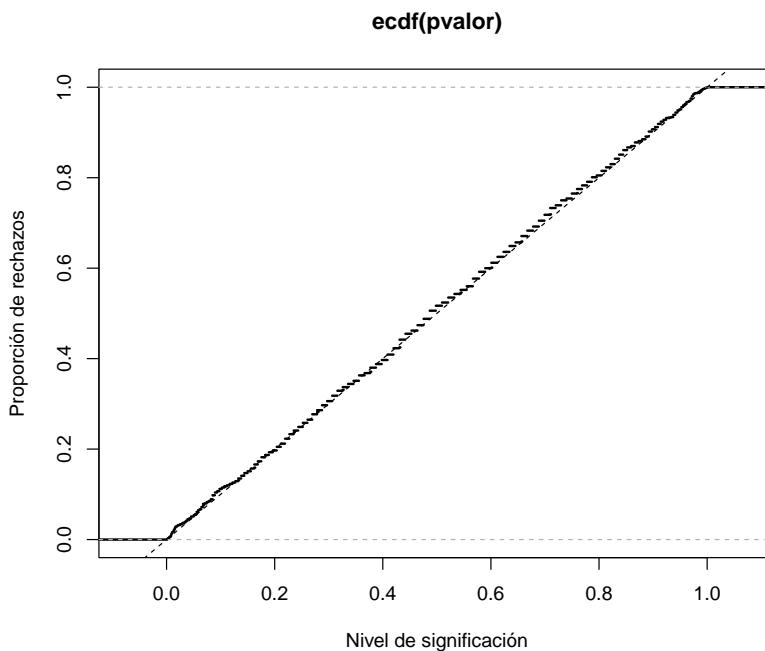


Figura 3.8: Proporción de rechazos con los distintos niveles de significación.

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.test.estadistico, distribution="chisq", nclasses=20, nestpar=0, df=99
# Test de Kolmogorov-Smirnov
ks.test(estadistico, "pchisq", df = 99)

##
```

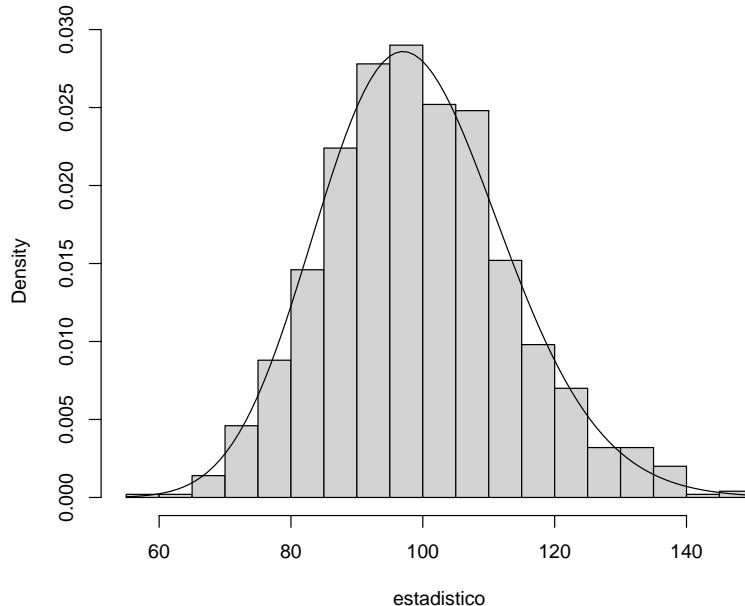


Figura 3.9: Distribución del estadístico del contraste.

```
## 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.

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

# Test chi-cuadrado
# chisq.test.cont(pvalor, distribution="unif", nclasses=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 esta-

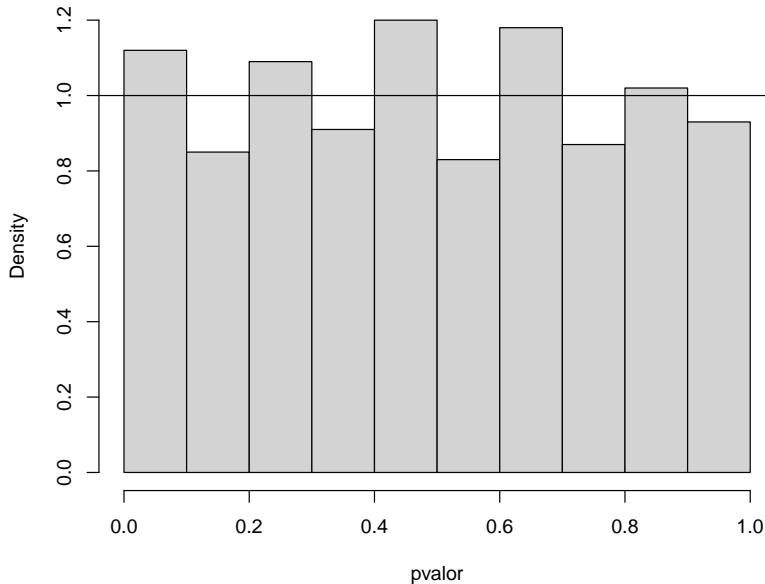


Figura 3.10: Distribución del p -valor del contraste.

dístico.

3.3.2 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 ejemplo⁴:

- 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

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

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.

3.4 Ejercicios

Ejercicio 3.1 (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:

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

El algoritmo está implementado en el siguiente código:

```
# -----
# Generador Von Neumann de números pseudoaleatorios
# -----
```

```
# initRANDVN(semilla,n)
# -----
#   Inicia el generador
#   n número de dígitos centrales, 4 por defecto (debe ser un número par)
#   Por defecto semilla del reloj
#   OJO: No se hace ninguna verificación de los parámetros
initRANDVN <- function(semilla = as.numeric(Sys.time()), n = 4) {
  .semilla <- as.double(semilla) %% 10^n # Cálculos en doble precisión
  .n <- n
  .aux <- 10^(2*n-n/2)
  .aux2 <- 10^(n/2)
  return(invisible(list(semilla=.semilla,n=.n)))
}

# RANDVN()
# -----
#   Genera un valor pseudoaleatorio con el generador de Von Neumann.
#   Actualiza la semilla (si no existe llama a initRANDVN).
RANDVN <- function() {
  if (!exists(".semilla", envir=globalenv())) initRANDVN()
  z <- .semilla^2
  .semilla <- trunc((z-trunc(z/.aux)*.aux)/.aux2)
  return(.semilla/10^.n)
}
```

```
# RANDVNN(n)
# -----
#   Genera un vector de valores pseudoaleatorios, de dimensión `n`
#   con el generador de Von Neumann.
#   Actualiza la semilla (si no existe llama a initRANDVN).
RANDVNN <- function(n = 1000) {
  x <- numeric(n)
  for(i in 1:n) x[i] <- RANDVN()
  return(x)
  # return(replicate(n, RANDVN())) # Alternativa más rápida
}
```

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

Ejercicio 3.2

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 4

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).

4.1 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, \dots, 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) de los grandes números*:

- Si X_1, X_2, \dots es una secuencia de variables aleatorias independientes con:

$$E(X_i) = \mu \text{ y } \text{Var}(X_i) = \sigma^2 < \infty,$$

entonces $\bar{X}_n = (X_1 + \dots + X_n)/n$ converge en probabilidad a μ , i.e. para cualquier $\varepsilon > 0$:

$$\lim_{n \rightarrow \infty} P(|\bar{X}_n - \mu| < \varepsilon) = 1.$$

- *La ley fuerte* establece la convergencia casi segura.

Ejemplo 4.1 (Aproximación de una probabilidad)

Simulamos una distribución de Bernoulli de parámetro $p = 0.5$:

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

La aproximación por simulación de p será:

```
mean(rx)
```

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

Podemos generar un gráfico con la evolución de la aproximación con el siguiente código:

```
plot(cumsum(rx)/1:n sim, 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)
```

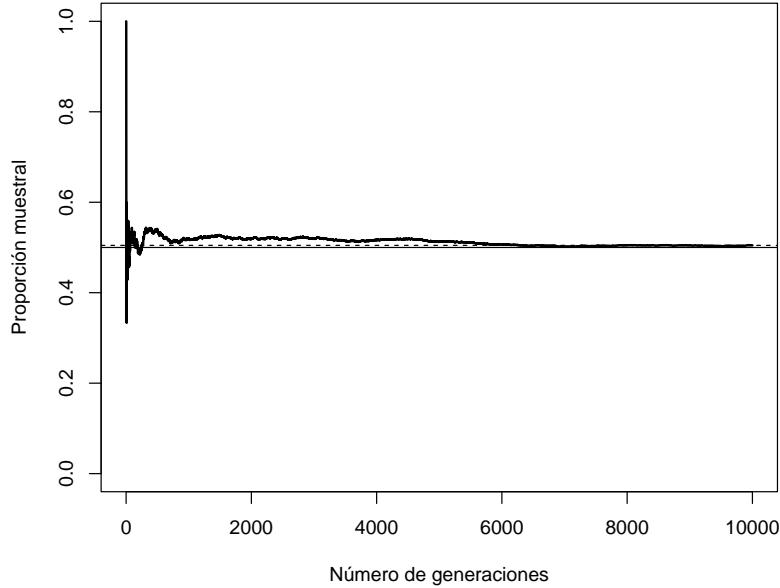


Figura 4.1: Aproximación de la proporción en función del número de generaciones.

4.1.1 Detección de problemas de convergencia

Una suposición crucial es que las variables X_i deben tener varianza finita (realmente esta suposición puede relajarse: $E(|X_i|) < \infty$). 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)
plot(cumsum(rx)/1:n sim, type="l", lwd=2,
      xlab="Número de generaciones", ylab="Media muestral")
```

Para detectar problemas de convergencia es 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)
```

4.2 Estimación de la precisión

En el caso de la media muestral \bar{X}_n , un estimador insesgado de $Var(\bar{X}_n) = \sigma^2/n$ es la cuasi-varianza muestral:

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

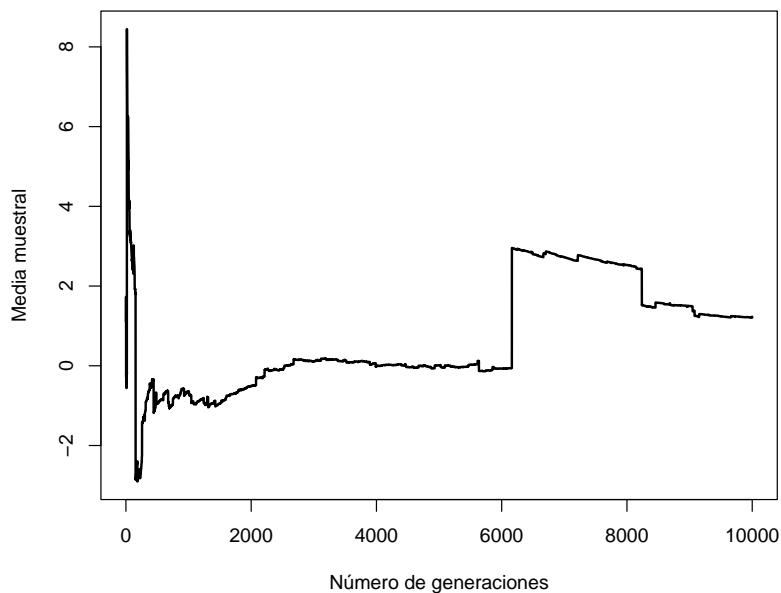


Figura 4.2: Evolución de la media muestral de una distribución de Cauchy en función del número de generaciones.

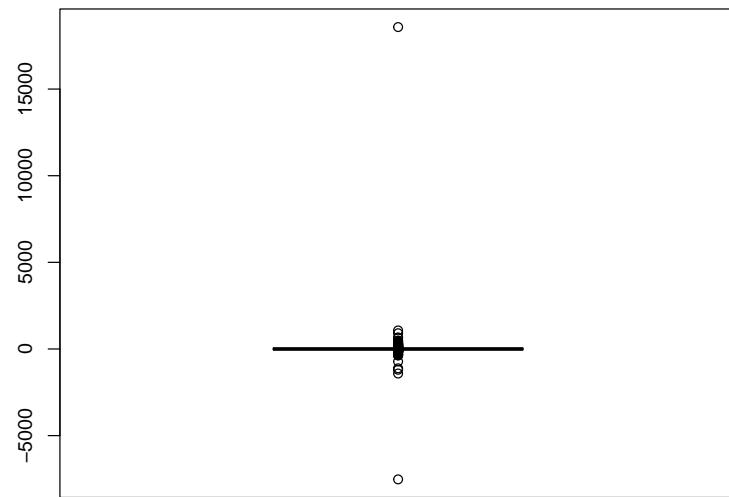


Figura 4.3: Gráfico de cajas de 10000 generaciones de una distribución de Cauchy.

con:

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

En el caso de una proporción \hat{p}_n :

$$\widehat{Var}(\hat{p}_n) = \frac{\hat{p}_n(1 - \hat{p}_n)}{n - 1},$$

aunque se suele emplear la varianza muestral.

Los valores obtenidos servirían como medidas básicas de la precisión de la aproximación, aunque su principal aplicación es la construcción de intervalos de confianza. Para ello podemos emplear el resultado mostrado a continuación.

4.3 Teorema central del límite

Si X_1, X_2, \dots es una secuencia de variables aleatorias independientes con $E(X_i) = \mu$ y $Var(X_i) = \sigma^2 < \infty$, entonces:

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

i.e. $\lim_{n \rightarrow \infty} F_{Z_n}(z) = \Phi(z)$. Por tanto, un intervalo de confianza asintótico para μ es:

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

Podemos considerar que $z_{1-\alpha/2} \frac{\hat{S}_n}{\sqrt{n}}$ es la precisión obtenida (con nivel de confianza $1 - \alpha$).

La convergencia de la aproximación, además de ser aleatoria, se podría considerar lenta. La idea es que para doblar la precisión (disminuir el error a la mitad), necesitaríamos un número de generaciones cuatro veces mayor. 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).

Ejemplo 4.2 (Aproximación de la media de una distribución normal)

```
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 considerar (se suele denominar error de la aproximación):

```
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) # Varianzas muestrales
esterr <- sqrt((cumsum(rx^2)/n - est^2)/(n-1)) # Errores estándar
```

```

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)

```

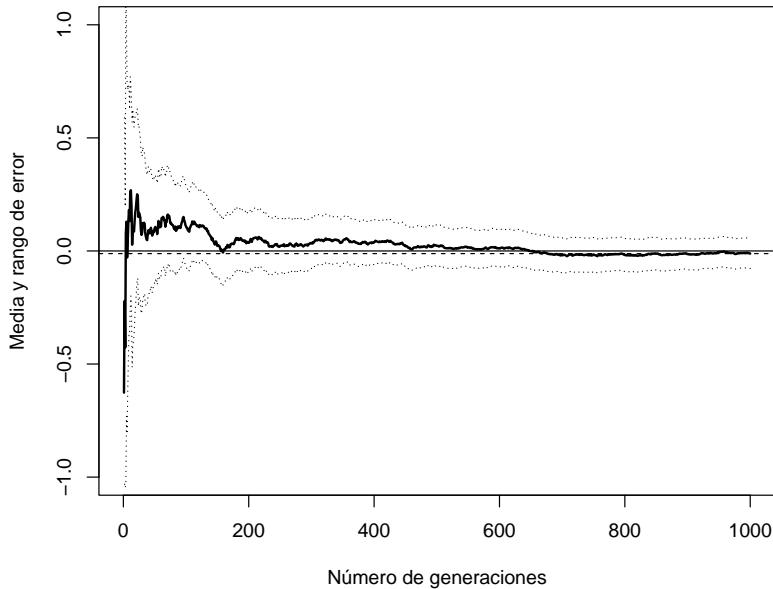


Figura 4.4: Gráfico de convergencia incluyendo el error de la aproximación.

4.4 Determinación del número de generaciones

Normalmente el valor de n se toma 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 simulaciones del orden de $n = 100, 200, 500$. Sin embargo, en otros casos 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 podría ser el siguiente:

1. Hacer $j = 0$ y fijar un tamaño inicial n_0 (e.g. 30 ó 60).
2. Generar $\{X_i\}_{i=1}^{n_0}$ y calcular \bar{X}_{n_0} y \widehat{S}_{n_0} .
3. Mientras $z_{1-\alpha/2} \widehat{S}_{n_j} / \sqrt{n_j} > \varepsilon$ hacer:
 - 3.1. $j = j + 1$.
 - 3.2. $n_j = \left\lceil \left(z_{1-\alpha/2} \widehat{S}_{n_{j-1}} / \varepsilon \right)^2 \right\rceil$.

3.3. Generar $\{X_i\}_{i=n_{j-1}+1}^{n_j}$ y calcular \bar{X}_{n_j} y \hat{S}_{n_j} .

4. Devolver \bar{X}_{n_j} y $z_{1-\alpha/2} \frac{\hat{S}_{n_j}}{\sqrt{n_j}}$.

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

$$z_{1-\alpha/2} \frac{\hat{S}_n}{\sqrt{n}} < \varepsilon |\bar{X}_n|.$$

4.5 El problema de la dependencia

En el caso de dependencia, la estimación de la precisión se complica:

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

Ejemplo 4.3 (aproximación de una proporción bajo dependencia, cadena de Markov)

Supongamos que en A Coruña llueve de media 1/3 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
```

Se podría pensar en emplear las expresiones anteriores:

```
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") # Prob. teor. cadenas Markov
```

La aproximación de la proporción sería correcta (es consistente):

```
est[nsim]
```

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

Sin embargo, al ser datos dependientes esta aproximación del error estandar no es adecuada:

```
esterr[nsim]
```

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

En este caso al haber dependencia positiva se produce una subestimación del verdadero error estandar.

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

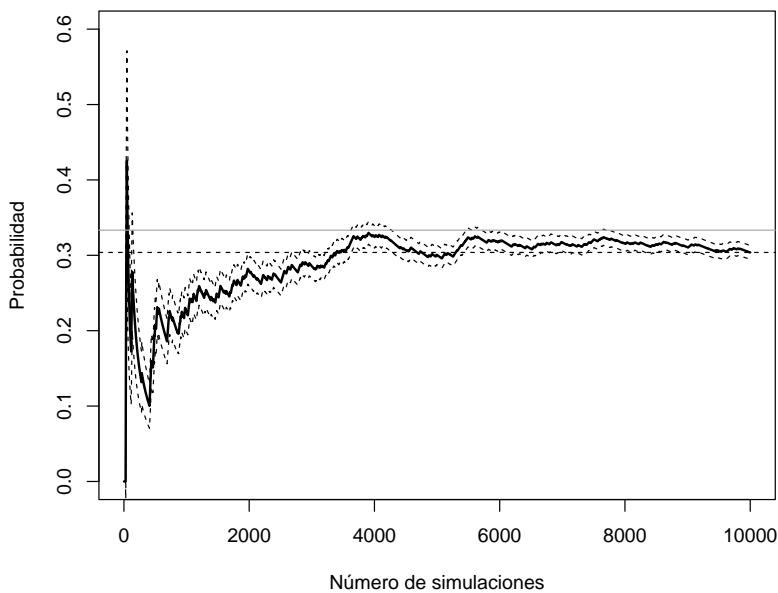


Figura 4.5: Gráfico de convergencia incluyendo el error de la aproximación (calculado asumiendo independencia).

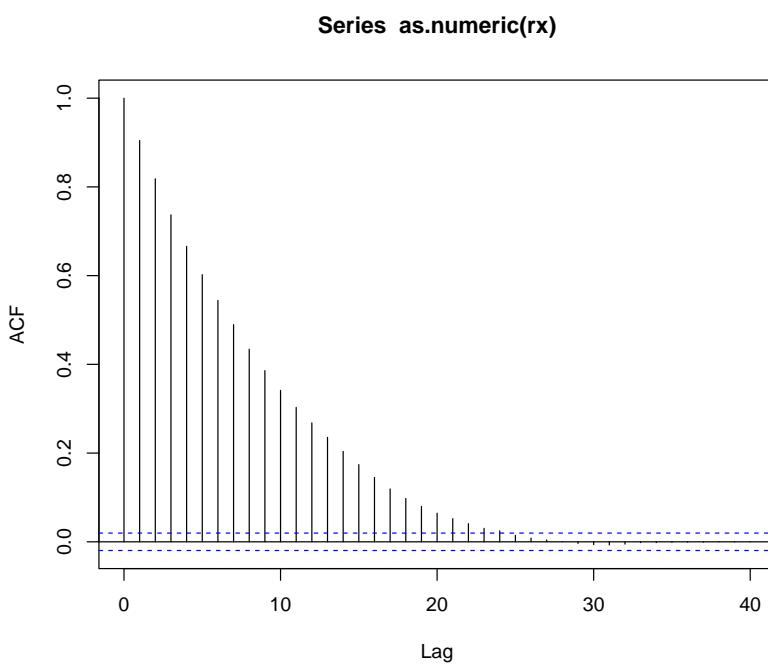


Figura 4.6: Correlograma de la secuencia indicadora de días de lluvia.

El gráfico de autocorrelaciones sugiere que si tomamos 1 de cada 25 podemos suponer independencia.

```
lag <- 24
xlag <- c(rep(FALSE, lag), TRUE)
rxi <- rx[xlag]
```

```
acf(as.numeric(rxi))
```

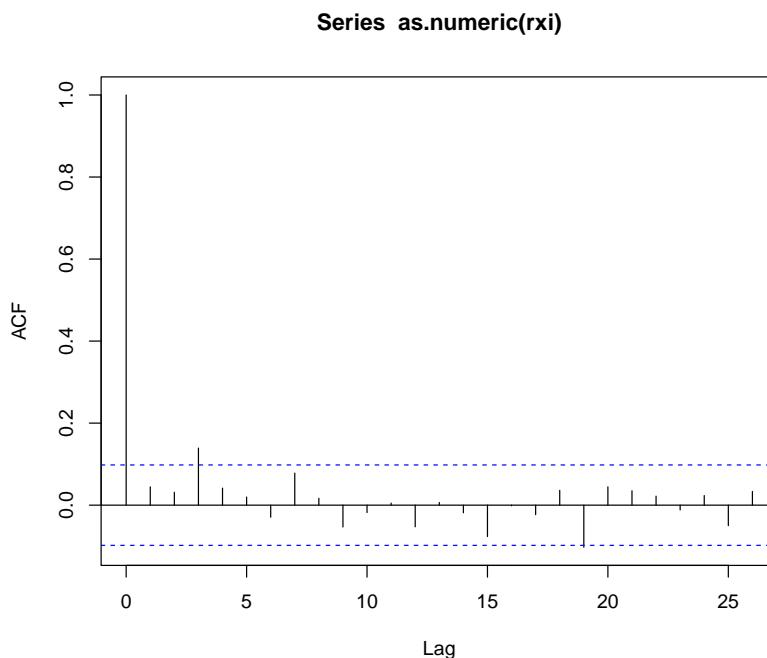


Figura 4.7: Correlograma de la subsecuencia de días de lluvia obtenida seleccionando uno de cada 25 valores.

```
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
```

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

```
esterr[nrxi]
```

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

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

Por ejemplo, se podría pensar en considerar las medias de grupos de 24 valores consecutivos y suponer que hay independencia entre ellas:

```
r xm <- rowMeans(matrix(rx, ncol = lag, byrow = TRUE))
nr xm <- length(r xm)
n <- 1:nr xm
est <- cumsum(r xm)/n
esterr <- sqrt((cumsum(r xm^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(r xm)], lty=2)
```

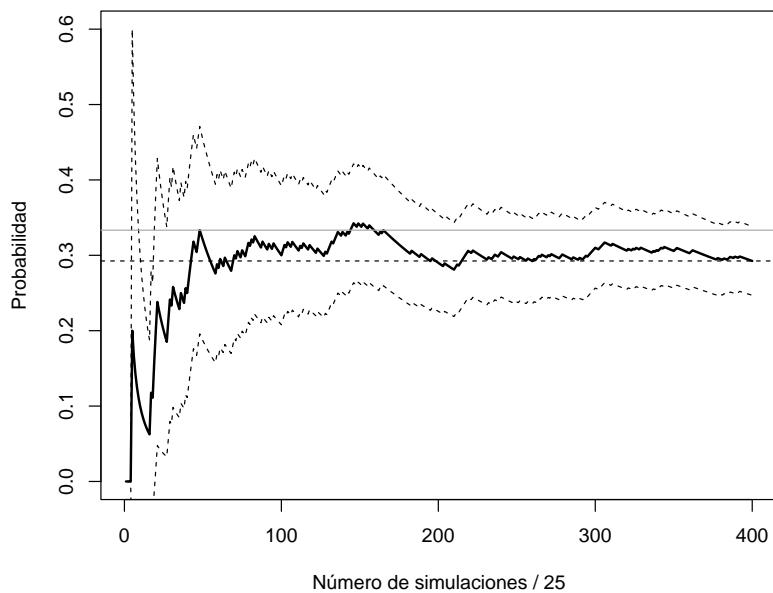


Figura 4.8: Gráfico de convergencia de la aproximación de la probabilidad a partir de la subsecuencia de días de lluvia (calculando el error de aproximación asumiendo independencia).

```
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
```

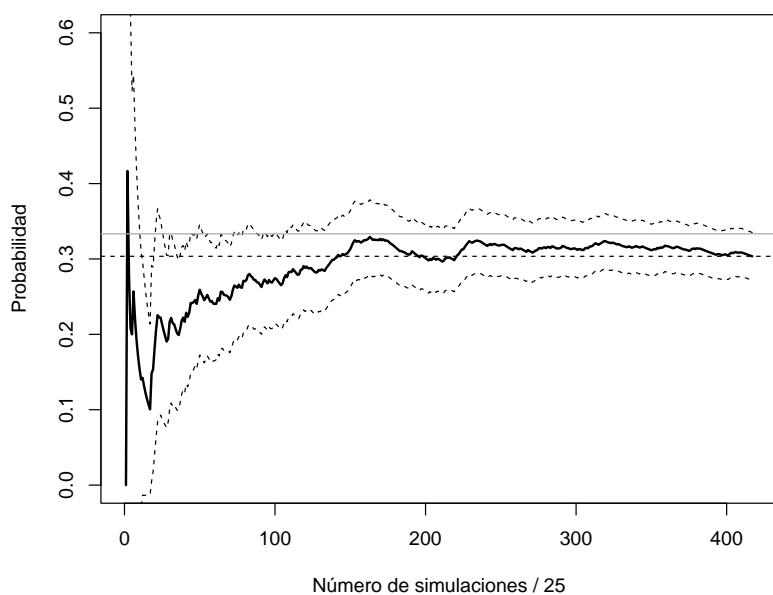


Figura 4.9: Gráfico de convergencia de las medias por lotes.

Esta es la idea del método de medias por lotes (*batch means; macro-micro replicaciones*) para la estimación de la varianza. En el ejemplo anterior se calcula el error estándar de la aproximación por simulación de la proporción:

```
esterr[nrxm]
```

```
## [1] 0.01582248
```

pero si el objetivo es la aproximación de la varianza (de la variable y no de las medias por lotes), habrá que reescalarlo adecuadamente. 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:

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

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

```
var.aprox <- nsim * esterr[length(rxm)]^2
var.aprox
```

```
## [1] 2.50351
```

Obtenida asumiendo independencia entre las medias por lotes, y que será una mejor aproximación que asumir independencia entre las generaciones de la variable:

```
var(rx)
```

```
## [1] 0.2115267
```

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")

```

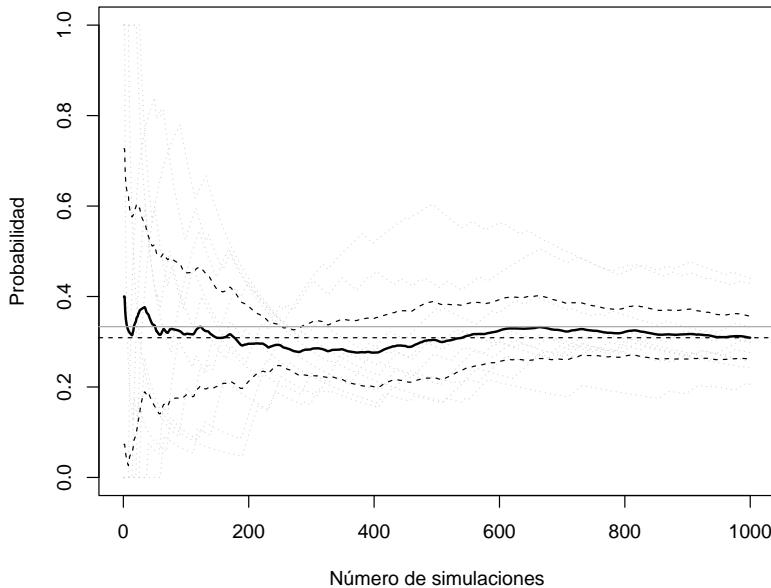


Figura 4.10: Gráfico de convergencia de la media de 10 secuencias generadas de forma independiente.

```

# 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 de simulación Monte Carlo de Cadenas de Markov (MCMC). Aparecen también en la simulación dinámica (por eventos o cuantos).

4.5.1 Periodo de calentamiento

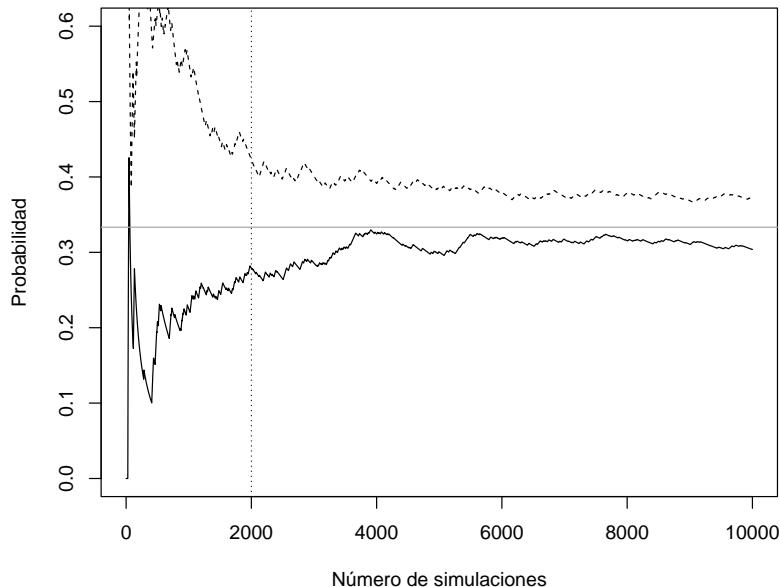
En el caso de simulación de datos dependientes (simulación dinámica) pueden aparecer problemas de estabilización. Puede ocurrir que el sistema evolucione lentamente en el tiempo hasta alcanzar su distribución estacionaria, siendo muy sensible a las condiciones iniciales con las que se comenzó la simulación. 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 establezca la distribución de probabilidad.

Como ejemplo comparamos la simulación del Ejemplo 4.3 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 4.4 (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:

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

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

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

o lo fijamos a la media (en este caso nos alejamos un poco de la distribución estacionaria, para que el “periodo de calentamiento” sea mayor). Después generamos los siguientes valores de forma recursiva:

```

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)

```

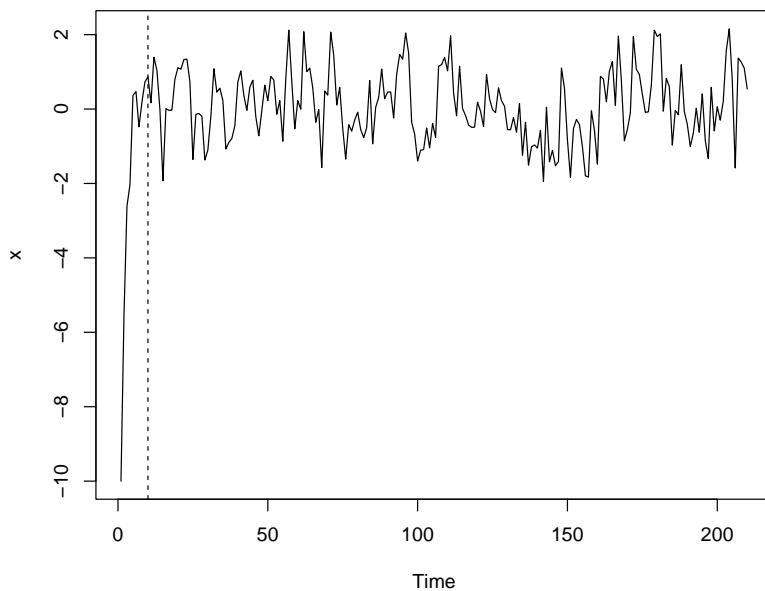


Figura 4.11: Ejemplo de una simulación de una serie de tiempo autorregresiva.

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.

4.6 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, las “bandas de confianza” obtenidas con el TCL son puntuales (si generamos nuevas secuencias de simulación es muy probable que no estén contenidas).
- 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 Sección 8.5).

Capítulo 5

Simulación de variables continuas

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

5.1 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:

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

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 5.1 (Método de inversión)

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

Ejemplo 5.1 (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 $X = -\frac{\ln(1-U)}{\lambda}$.

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*.

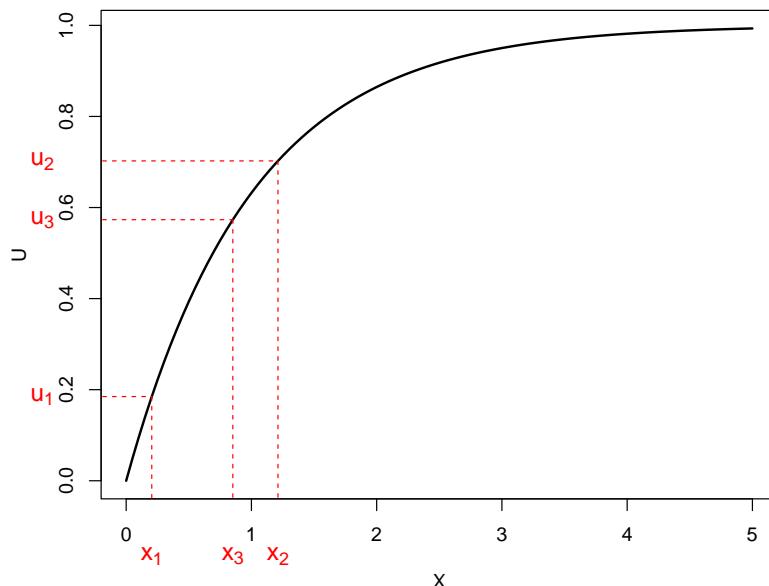


Figura 5.1: Ilustración de la simulación de una distribución exponencial por el método de inversión.

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
##      0.00     0.01    0.02

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 5.2 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.

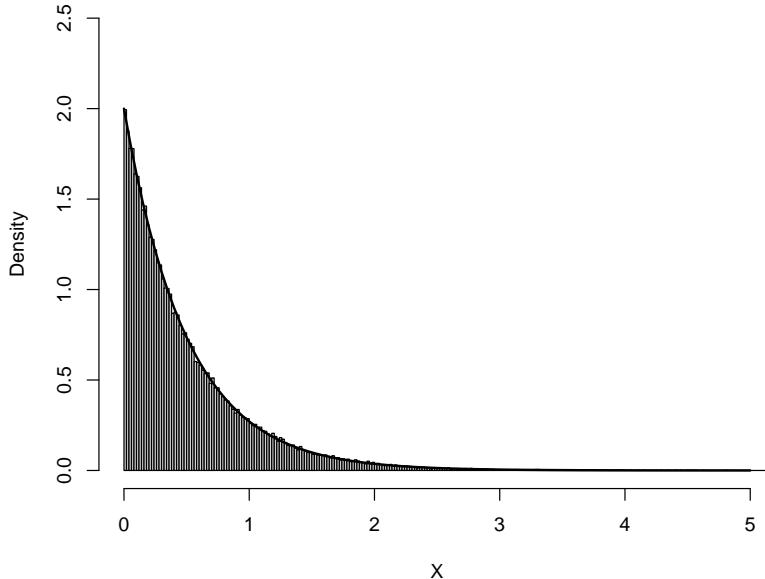


Figura 5.2: Distribución de los valores generados de una exponencial mediante el método de inversión.

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

Ejercicio 5.1 (distribución doble exponencial)

La distribución doble exponencial (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:

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

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

```

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)

```

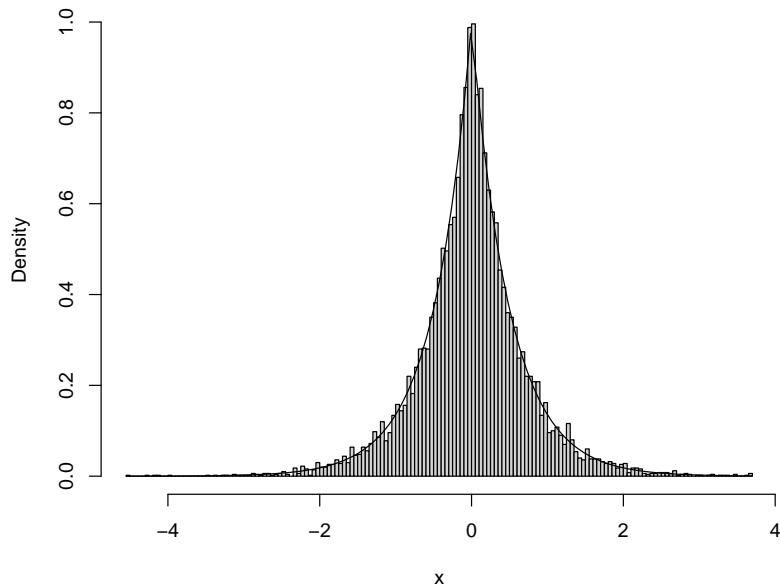


Figura 5.3: Distribución de los valores generados de una doble exponencial mediante el método de inversión.

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.

Nota: Esta distribución también se puede generar fácilmente simulando una distribución exponencial y asignando un signo positivo o negativo con equiprobabilidad (ver Ejemplo 5.6).

5.1.2 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 6.1, 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 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 2.1, R emplea por defecto este método para la generación de la distribución normal).

5.1.3 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 auxiliar¹:

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

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.

Algoritmo 5.2 (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).

5.2 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 $f(x)$.

¹R emplea una aproximación similar, basada en el algoritmo de Wichura (1988) más preciso, y que está implementado en el fichero fuente qnorm.c.

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):

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

De esta forma la primera componente tendrá la distribución deseada (Figura 5.4):

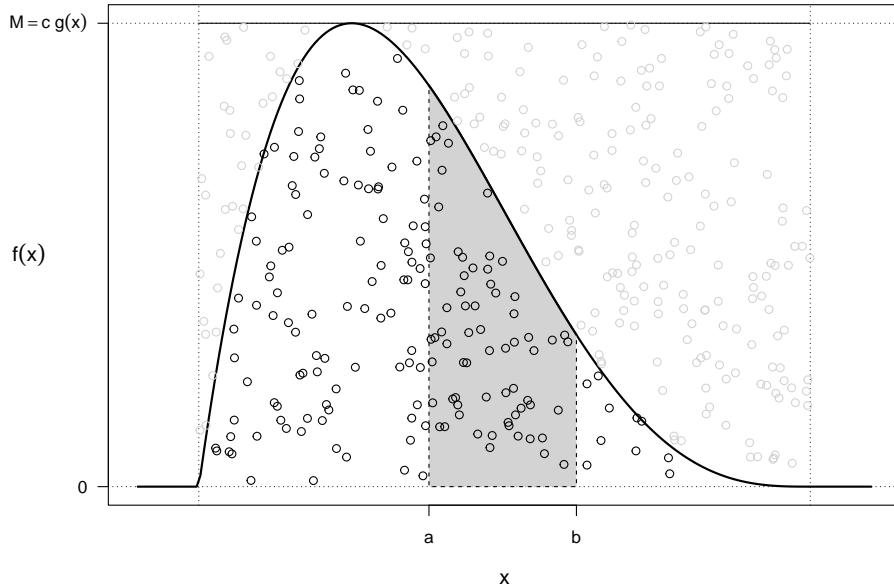


Figura 5.4: Puntos con distribución uniforme en el hipografo de una función de densidad.

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

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

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

$$(X, c \cdot U \cdot f(x)) \sim \mathcal{U}(A_{cf})$$

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

- Recíprocamente si $(X, Y) \sim \mathcal{U}(A_{cf})$ entonces² $X \sim f$.

Para generar valores de una variable aleatoria bidimensional con distribución uniforme en A_f (o en A_{f^*}), se emplea el resultado anterior para generar valores en $A_{cg} \supset A_f$, siendo g una densidad auxiliar (preferiblemente fácil de simular y similar a f). Teniendo en cuenta además que:

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

Por tanto, 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 T será f .

5.2.1 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:

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

²Emplearemos también $X \sim f$ para indicar que X es una variable aleatoria con función de densidad f .

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

Algoritmo 5.3 (Método de aceptación-rechazo; Von Neuman, 1951)

1. Generar $U \sim \mathcal{U}(0, 1)$.
2. Generar $T \sim g$.
3. Si $c \cdot U \cdot g(T) \leq f(T)$ devolver $X = T$,
en caso contrario volver al paso 1.

5.2.2 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 $c = M(b - a)$ 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 $T = a + (b - a)V$.
3. Si $M \cdot U \leq f(T)$ devolver $X = T$,
en caso contrario volver al paso 1.

Nota: no confundir M con $c = M(b - a)$.

Ejemplo 5.2 (simulación de distribución beta a partir de la uniforme)

Para simular una variable con función de densidad $Beta(\alpha, \beta)$:

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

(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 mayores³ 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 5.5), 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)
```

³Si 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.

```
c <- dbeta((s1 - 1)/(s1 + s2 - 2), s1, s2)
# abline(h = c, lty = 2)
segments(0, c, 1, c, lty = 2, lwd = 2)
abline(v = 0, lty = 3)
abline(v = 1, lty = 3)
abline(h = 0, lty = 3)
```

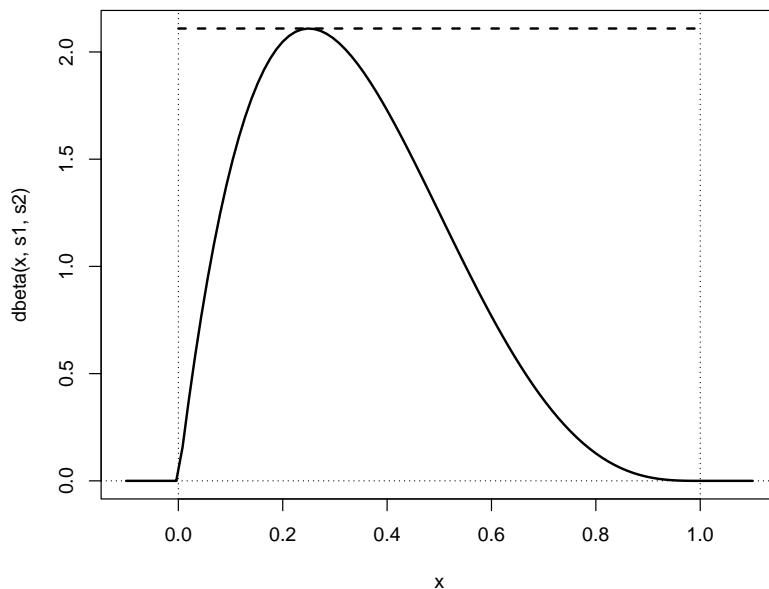


Figura 5.5: Densidad objetivo beta (línea continua) y densidad auxiliar uniforme reescalada (línea discontinua).

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

```
nge <- 0

rbeta2 <- function(s1 = 2, s2 = 2) {
  # Simulación por aceptación-rechazo
  # Beta a partir de uniforme
  c <- dbeta((s1 - 1)/(s1 + s2 - 2), s1, s2)
  while (TRUE) {
    U <- runif(1)
    X <- runif(1)
    nge <- nge+1
    if (c*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 $\text{Beta}(2, 4)$ (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   0.04
```

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)
```

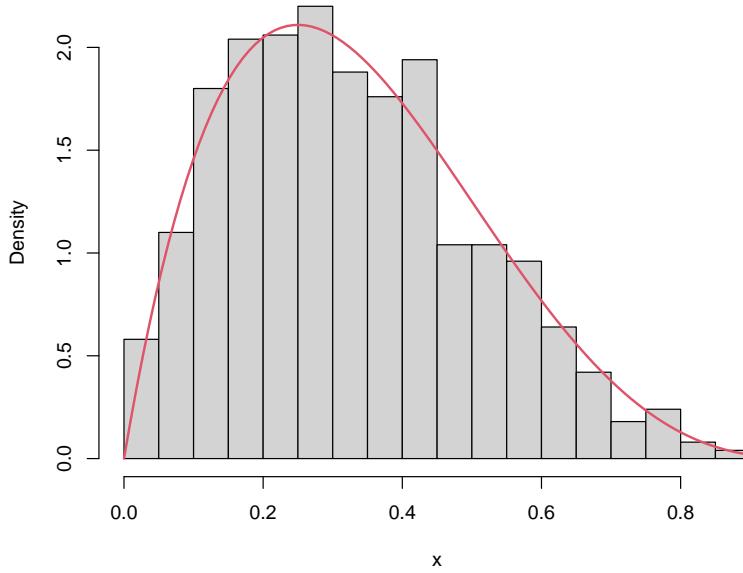


Figura 5.6: Distribución de los valores generados mediante el método de aceptación-rechazo.

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 5.2

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.

5.2.3 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:

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

Por tanto:

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

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 5.3). Basta con tener en cuenta que, si $f(x) = f^*(x)/k$:

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

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

Ejemplo 5.3 (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 5.1, 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 $[0, \infty]$:

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

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 5.7), 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)
```

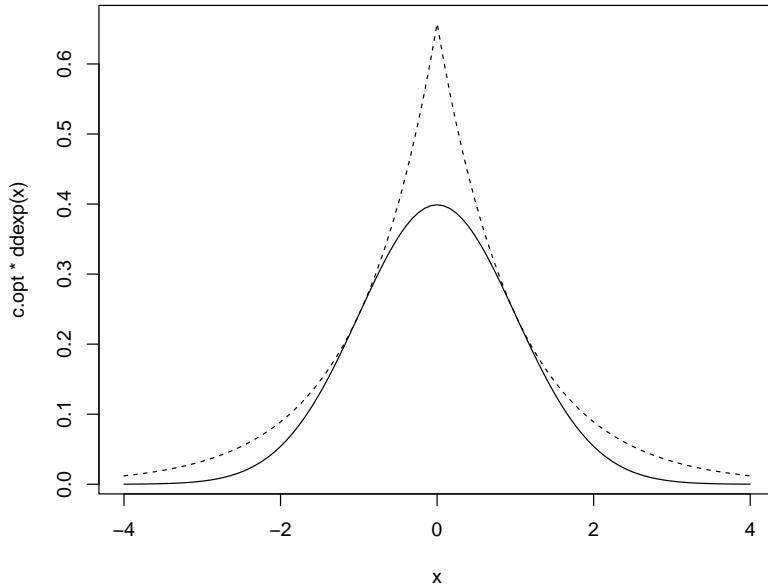


Figura 5.7: Densidad objetivo (normal estandarizada, línea continua) y densidad auxiliar (doble exponencial, línea discontinua).

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
```

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 se puede tener en cuenta que:

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

aunque en general puede ser recomendable emplear $c \cdot U \cdot g(T) \leq f(T)$.

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)
```

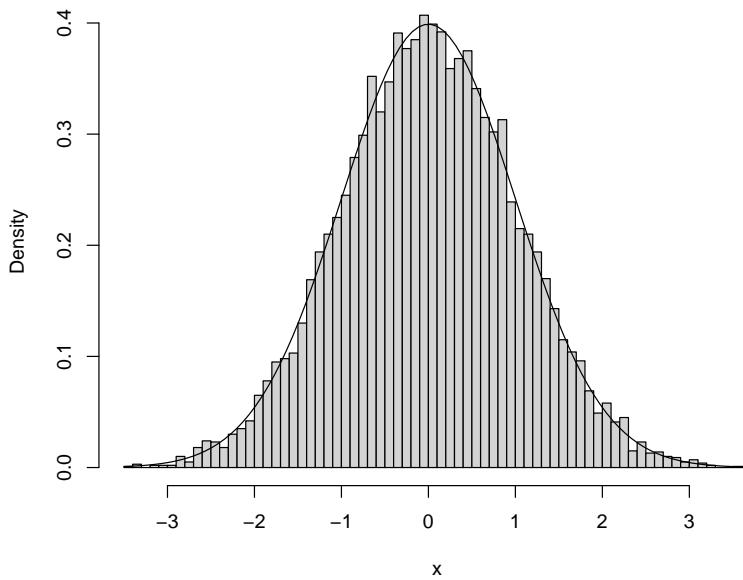


Figura 5.8: Distribución de los valores generados mediante el método de aceptación-rechazo.

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).

5.2.4 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 5.4 (simulación de la normal mediante la doble exponencial, continuación)

Continuando con el Ejemplo 5.3 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 $\lambda > 0$:

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

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:

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

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)
```

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 5.3.

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
```

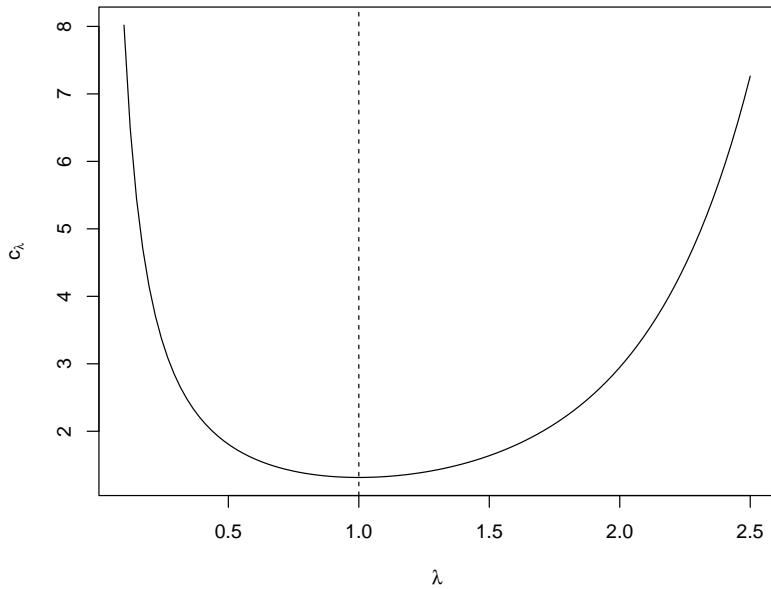


Figura 5.9: Representación de la cota óptima dependiendo del valor del parámetro.

```
## [1] 0.9999987
c.opt2
## [1] 1.315489
```

5.2.5 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.
- $\mathbf{x} = (x_1, \dots, x_n)^\top$ 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 pensar en 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 MV 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 5.3.1).

Ejercicio 5.3 (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%.

```
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)
```

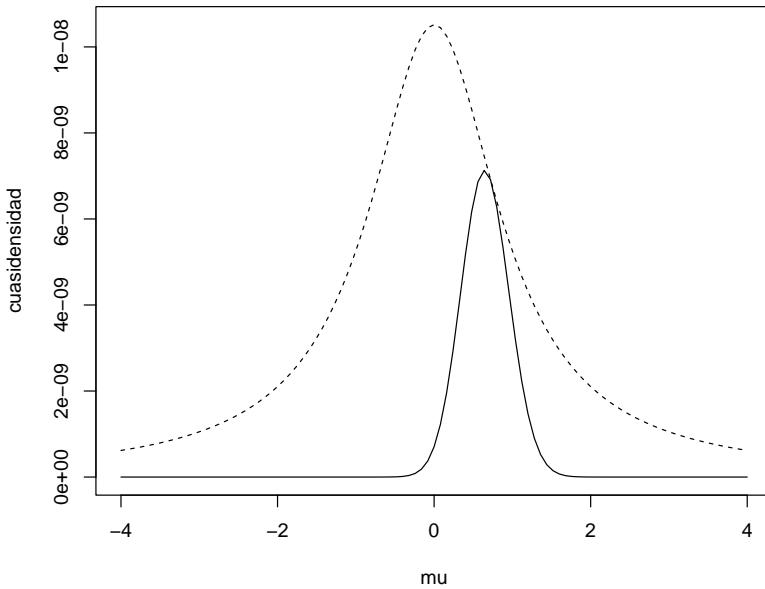


Figura 5.10: Comparación de la cuasidensidad a posteriori (línea continua) con la densidad a priori reescalada (línea discontinua).

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)
```

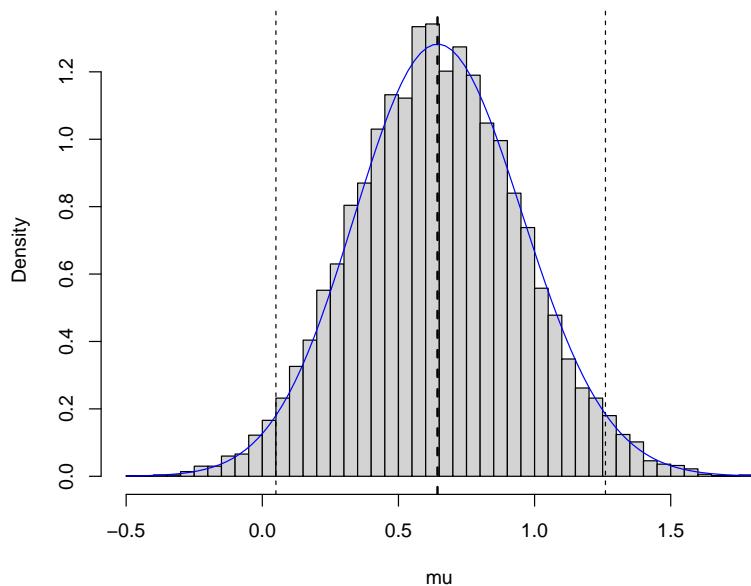


Figura 5.11: Distribución de los valores generados y aproximación del intervalo de credibilidad.

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

5.3 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 4.

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 approxime la densidad por abajo (una envolvente inferior):

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

Algoritmo 5.4 (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.

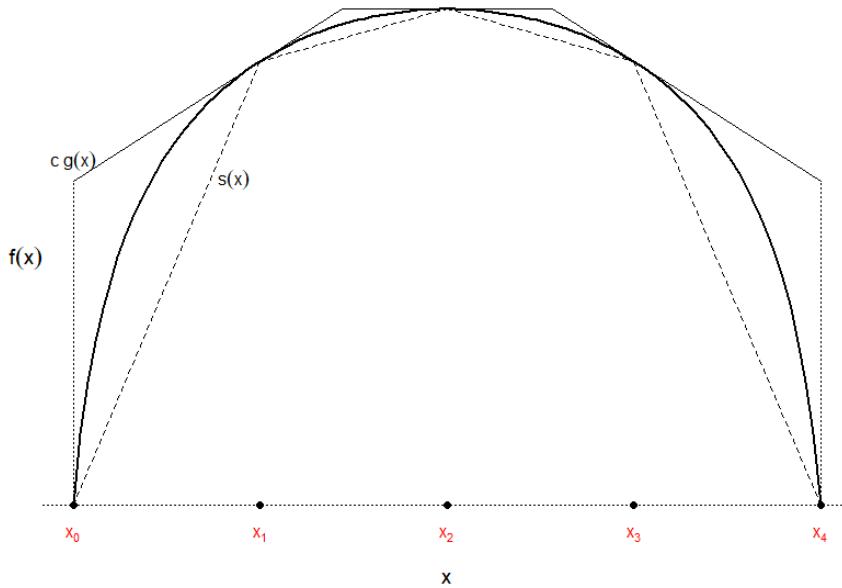


Figura 5.12: Ilustración del algoritmo de aceptación-rechazo con envolvente inferior (función "squeeze").

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.

5.3.1 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 $S_n = \{x_i : i = 0, \dots, n + 1\}$ 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 $\log f(x)$:

- $\underline{\phi}_n(x) = L_{i,i+1}(x)$

- $\bar{\phi}_n(x) = \min \{L_{i-1,i}(x), L_{i+1,i+2}(x)\}$

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

- $s_n(x) = \exp(\underline{\phi}_n(x))$
- $G_n(x) = \exp(\bar{\phi}_n(x))$

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 5.4) con el método de inversión.

Algoritmo 5.5 (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 5.12).

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

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

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

5.3.2 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:

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

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 $\mathcal{U}(C_{f^*})$ a partir de componentes 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 5.5 (simulación de la distribución de Cauchy mediante cociente de uniformes)

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

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

Por tanto, podemos emplear el siguiente código para generar valores de la densidad objetivo:

```
rcauchy.rou <- function(nsim) {
  # Cauchy mediante cociente de uniformes
  ngen <- nsim
  u <- runif(nsim, 0, 1)
  v <- runif(nsim, -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)
}

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

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

# 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
```

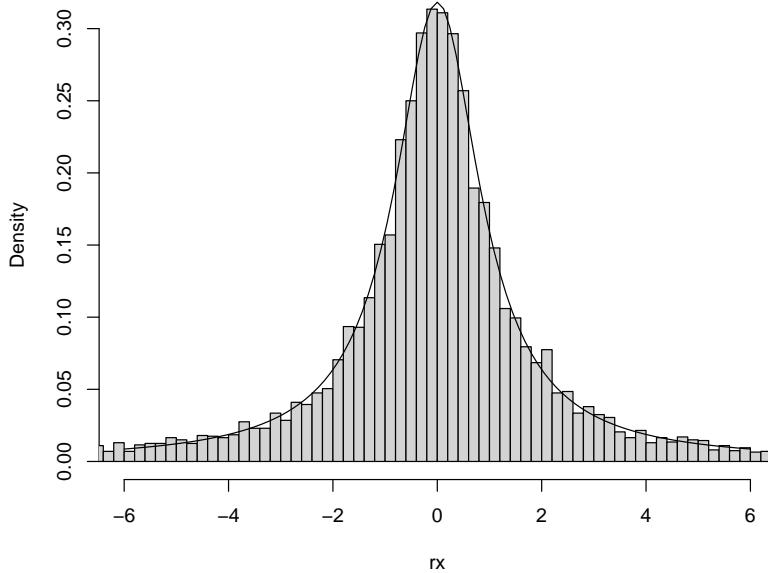


Figura 5.13: Distribución de los valores generados mediante el método de cociente de uniformes.

5.4 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 o variables aleatorias, incluyendo el caso discreto).

Algoritmo 5.6 (simulación de una mixtura discreta)

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

Ejemplo 5.6 (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:

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

El algoritmo resultante sería el siguiente (empleando dos números pseudo aleatorios 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 $X = \ln(V) / \lambda$.

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 3.3 del libro Técnicas de Remuestreo). 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 5.7 (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 6.7), ya que el resultado sería válido cambiando las funciones de densidad f y g por funciones de masa de probabilidad.

5.5 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)⁴, 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 (pero que se puede generalizar para otras distribuciones y dependencia).
- Algoritmos de Jöhnk (1963) y Cheng (1978) para la generación de la distribución beta (como ejemplo de las eficiencias de los métodos de aceptación-rechazo).

⁴Cuidado 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.

5.5.1 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 $\sqrt{2E} \sin 2\pi U$ son $\mathcal{N}(0, 1)$ independientes.

Algoritmo 5.8 (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 $X_1 = W_1 \cos W_2$, $X_2 = W_1 \sin W_2$.

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 exponentiales 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.

5.5.2 Simulación de la distribución beta

Existen multitud de algoritmos para simular la distribución $\text{Beta}(a, b)$. 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 $Y \sim \text{Gamma}(a, s)$ y $Z \sim \text{Gamma}(b, s)$ son independientes, entonces

$$X = \frac{Y}{Y + Z} \sim \text{Beta}(a, b).$$

Como la distribución resultante no depende de s y $\chi_n^2 \stackrel{d}{=} \text{Gamma}\left(\frac{n}{2}, \frac{1}{2}\right)$, se podría considerar $Y \sim \chi_{2a}^2$ y $Z \sim \chi_{2b}^2$ 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:

$$U_{(k)} \sim \text{Beta}(k, n + 1 - k).$$

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 5.9 (de Fox, 1963)

1. Generar $U_1, U_2, \dots, U_{a+b-1} \sim \mathcal{U}(0, 1)$.
2. Ordenar: $U_{(1)} \leq U_{(2)} \leq \dots \leq U_{(a+b-1)}$.
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 5.10 (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 implementar⁵ pero mucho más eficiente.

Algoritmo 5.11 (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 $U_1, U_2 \sim \mathcal{U}(0, 1)$.
 2. Hacer $V = \beta \cdot \ln\left(\frac{U_1}{1-U_1}\right)$ y $W = a \cdot e^V$.
 3. Si $\alpha \cdot \ln\left(\frac{\alpha}{b+W}\right) + \gamma V - \ln 4 \geq \ln(U_1^2 U_2)$ devolver $X = \frac{W}{b+W}$,
en caso contrario volver al paso 1.
-

⁵R implementa este algoritmo en el fichero fuente rbeta.c.

Capítulo 6

Simulación de variables discretas

Se trata de simular una variable aleatoria discreta X con función de masa de probabilidad (f.m.p.):

$$\begin{array}{c|cccccc} x_i & x_1 & x_2 & \cdots & x_n & \cdots \\ \hline P(X = x_i) & p_1 & p_2 & \cdots & p_n & \cdots \end{array}$$

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 6.3.

6.1 Método de la transformación cuantil

Este método es una adaptación del método de inversión (válido para el caso continuo) a distribuciones discretas. 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 , $i = 1, 2, \dots$). 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:

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

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

Teorema 6.1 (de inversión generalizada)

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

Demostración: Bastaría ver que:

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

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 6.1 (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 ($x_1 < x_2 < \dots$), la función cuantil será:

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

Para encontrar este valor se puede emplear el siguiente algoritmo:

Algoritmo 6.2 (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:

```
rfmp0 <- function(x, prob = 1/length(x), nsim = 1000) {
  # Simulación nsim v.a. discreta a partir de fmp
  # por inversión generalizada (transformación cuantil)
  X <- numeric(nsim)
  U <- runif(nsim)
  for(j in 1:nsim) {
    i <- 1
    Fx <- prob[1]
    while (Fx < U[j]) {
      i <- i + 1
      Fx <- Fx + prob[i]
```

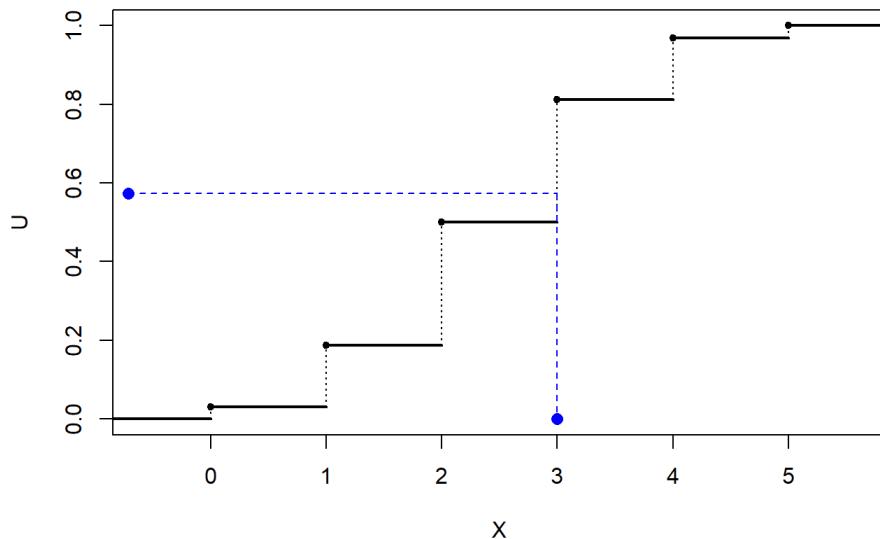


Figura 6.1: Ilustración de la simulación de una distribución discreta mediante transformación cuantil.

```

    }
    X[j] <- x[i]
}
return(X)
}

```

Adicionalmente, para disminuir el tiempo de computación, se puede almacenar las probabilidades acumuladas en una tabla. Si también se quiere obtener el número de comparaciones se puede considerar una variable global `ncomp`:

```

ncomp <- 0

rfmp <- function(x, prob = 1/length(x), nsim = 1000) {
  # Simulación nsim v.a. discreta a partir de fmp
  # por inversión generalizada (transformación cuantil)
  # Inicializar FD
  Fx <- cumsum(prob)
  # Simular
  X <- numeric(nsim)
  U <- runif(nsim)
  for(j in 1:nsim) {
    i <- 1
    while (Fx[i] < U[j]) i <- i + 1
    X[j] <- x[i]
    ncomp <<- ncomp + i
  }
  return(X)
  # return(factor(X, levels = x))
}

```

Ejercicio 6.1 (Simulación de una binomial mediante el método de la transformación cuantil)

Generar, por el método de la transformación cuantil usando búsqueda secuencial, una muestra de $nsim = 10^5$ observaciones de una variable $\mathcal{B}(10, 0.5)$. 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 np) y el número medio de comparaciones para generar cada observación.

Empleamos la rutina anterior para generar las simulaciones:

```
set.seed(1)
n <- 10
p <- 0.5
nsim <- 10^5

x <- 0:n
fmp <- dbinom(x, n, p)

ncomp <- 0
system.time( rx <- rfmp(x, fmp, nsim) )

##      user    system elapsed
##      0.06     0.00     0.06
```

Aproximación de la media:

```
mean(rx)
```

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

El valor teórico es $n*p = 5$.

Número medio de comparaciones:

```
ncomp/nsim
```

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

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

Análisis de los resultados:

```
res <- table(rx)/nsim
# res <- table(factor(rx, levels = x))/nsim
plot(res, ylab = "frecuencia relativa", xlab = "valores")
points(x, fmp, pch = 4) # Comparación teórica

res <- as.data.frame(res)
names(res) <- c("x", "psim")
res$pteor <- fmp
res

##      x      psim      pteor
## 1  0 0.00100 0.0009765625
## 2  1 0.00981 0.0097656250
## 3  2 0.04457 0.0439453125
## 4  3 0.11865 0.1171875000
## 5  4 0.20377 0.2050781250
## 6  5 0.24477 0.2460937500
## 7  6 0.20593 0.2050781250
## 8  7 0.11631 0.1171875000
## 9  8 0.04415 0.0439453125
## 10 9 0.01021 0.0097656250
```

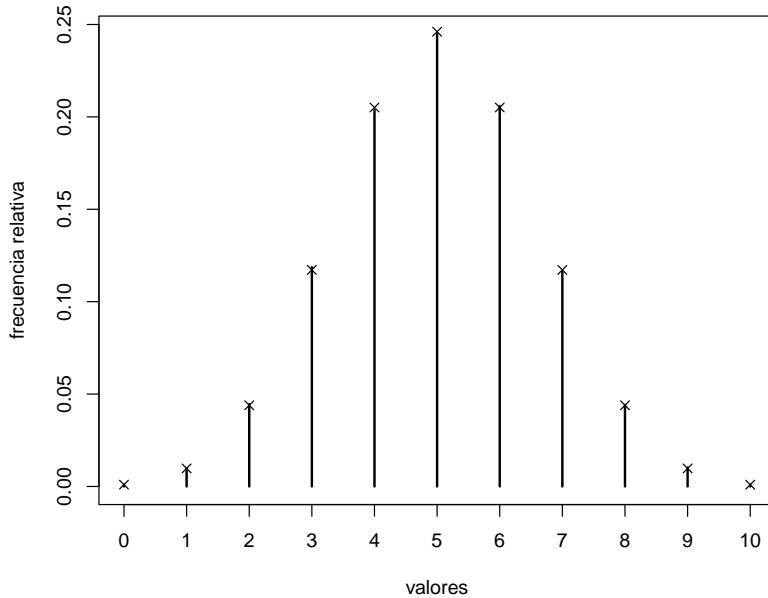


Figura 6.2: Comparación de las frecuencias relativas de los valores generados, mediante el método de la transformación cuantil, con las probabilidades teóricas.

```
## 11 10 0.00083 0.0009765625
# Errores
max(abs(res$psim - res$ptheor))

## [1] 0.0014625
max(abs(res$psim - res$ptheor) / res$ptheor)

## [1] 0.15008
```

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$ptheor <- fmp` gererará un error. Una posible solución sería trabajar con factores (hacer que la función `rfmp()` devuelva `factor(X, levels = x)`) o emplear `res <- table(factor(rx, levels = x))/nsim`.

6.1.1 Eficiencia del algoritmo

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

$$\frac{i}{P(\mathcal{J} = i)} \begin{array}{c|ccccc} & 1 & 2 & \dots & n & \dots \\ \hline P(\mathcal{J} = i) & p_1 & p_2 & \dots & p_n & \dots \end{array}$$

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:

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

Realmente, cuando la variable toma un número finito de valores: x_1, x_2, \dots, 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:

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

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:

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

Ejercicio 6.2 (Simulación de una binomial, continuación)

Repetir el Ejercicio 6.1 anterior ordenando previamente las probabilidades en orden decreciente y también empleando la función `sample` de R.

```
set.seed(1)
tini <- proc.time()

ncomp <- 0
ind <- order(fmp, decreasing=TRUE)
rx <- rfmp(x[ind], fmp[ind], nsim)

tiempo <- proc.time() - tini
tiempo

##      user    system elapsed
##      0.04     0.00     0.04

# Número de comparaciones
ncomp/nsim

## [1] 3.08369
sum((1:length(x))*fmp[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 6.3):

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

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

6.2 Método de la tabla guía

La idea consiste en construir m subintervalos equiespaciados contenidos en $[0, 1]$ de la forma:

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

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

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

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

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

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

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

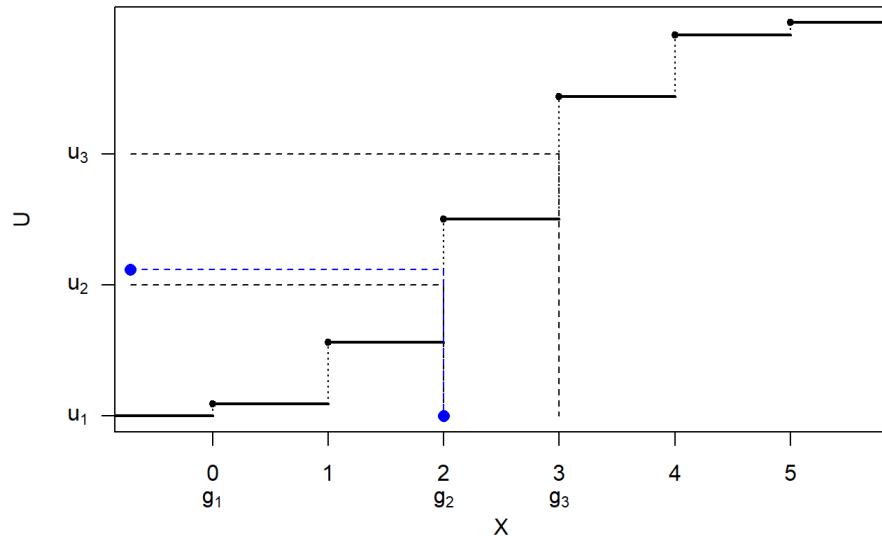


Figura 6.3: Ilustración de la simulación de una distribución discreta mediante tabla guía.

Algoritmo 6.3 (de simulación mediante una tabla guía)

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 $X = x_i$.

Ejercicio 6.3 (Simulación de una binomial mediante tabla guía)

Diseñar una rutina que permita generar $nsim$ valores de una distribución discreta usando una tabla guía. Repetir el Ejercicio 6.1 anterior empleando esta rutina con $m = n - 1$.

```

rfmp.tabla <- function(x, prob = 1/length(x), m, nsim = 1000) {
  # Simulación v.a. discreta a partir de función de masa de probabilidad
  # por tabla guía de tamaño m
  # 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
  }
  # Generar valores
  X <- numeric(nsim)
  U <- runif(nsim)
  for(j in 1:nsim) {
    i <- g[floor(U[j]*m)+1]
    while (Fx[i] < U[j]) i <- i + 1
    X[j] <- x[i]
  }
  return(X)
}

set.seed(1)
system.time( rx <- rfmp.tabla(x, fmp, n=1, nsim) )

##      user    system elapsed
##      0.03     0.00     0.03

```

Análisis de los resultados:

```

res <- table(rx)/nsim
plot(res, ylab = "frecuencia relativa", xlab = "valores")
points(x, fmp, pch = 4)  # Comparación teórica

```

6.3 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 6.4 (“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 vacíos terminar.
4. Seleccionar $l \in L$ y $h \in H$.
5. Hacer $a_l = h$.
6. Eliminar l de L .

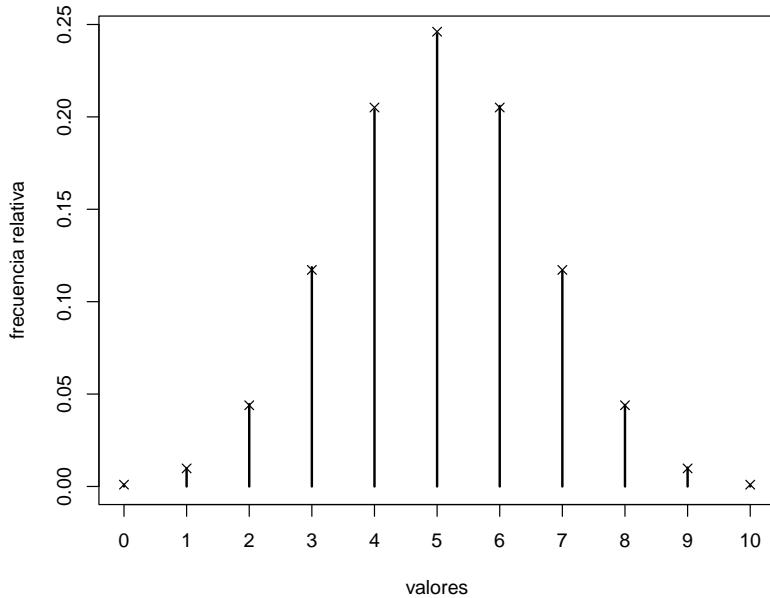


Figura 6.4: Comparación de las frecuencias relativas de los valores generados, mediante el método de la tabla guía, con las probabilidades teóricas.

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.
-

El algoritmo para generar las simulaciones es el estándar del método de composición:

Algoritmo 6.5 (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 R¹.

Ejercicio 6.4 (Simulación de una binomial mediante en método de Alias)

Diseñar una rutina que permita generar $nsim$ valores de una distribución discreta usando el método de Alias. Repetir el Ejercicio 6.1 anterior empleando esta rutina.

¹R 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 2.1).

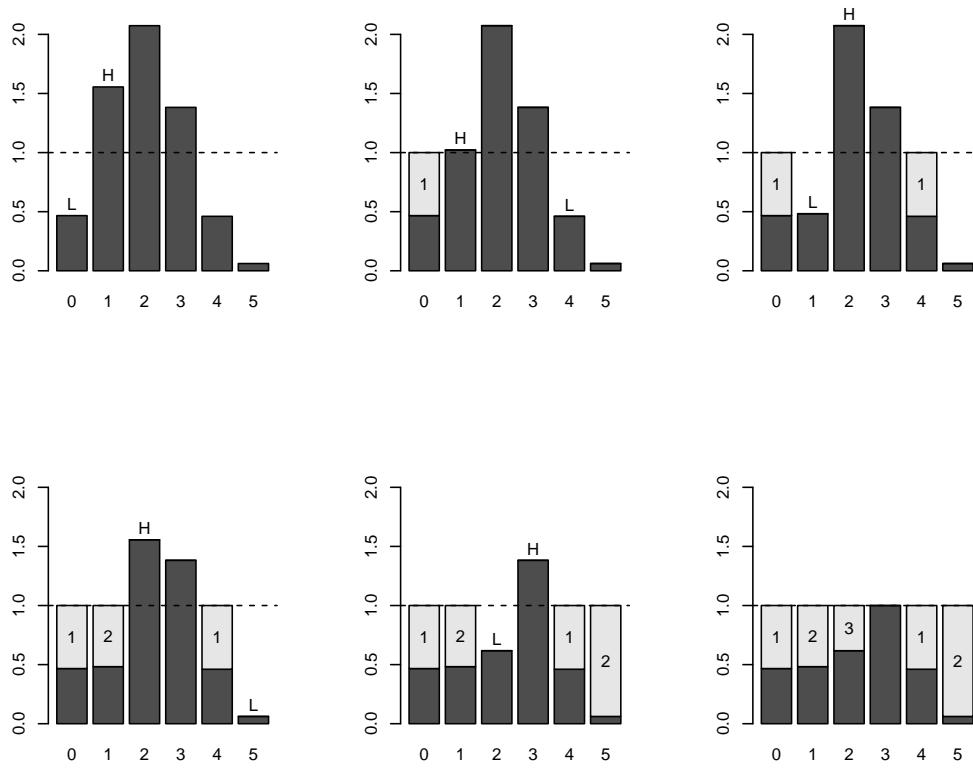


Figura 6.5: Pasos del algoritmo de inicialización del método Alias.

```

rfmp.alias <- function(x, prob = 1/length(x), nsim = 1000) {
  # 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(nsim)
  i <- floor(runif(nsim)*length(x)) + 1
  return( x[ ifelse( V < q[i], i, a[i]) ] )
}

n <- 10

```

```

p <- 0.5
nsim <- 10^5
x <- 0:n
fmp <- dbinom(x, n, p)
set.seed(1)
system.time( rx <- rfmp.alias(x, fmp, nsim) )

##    user  system elapsed
##    0.03    0.00    0.03

```

Análisis de los resultados:

```

res <- table(rx)/nsim
plot(res, ylab = "frecuencia relativa", xlab = "valores")
points(x, fmp, pch = 4) # Comparación teórica

```

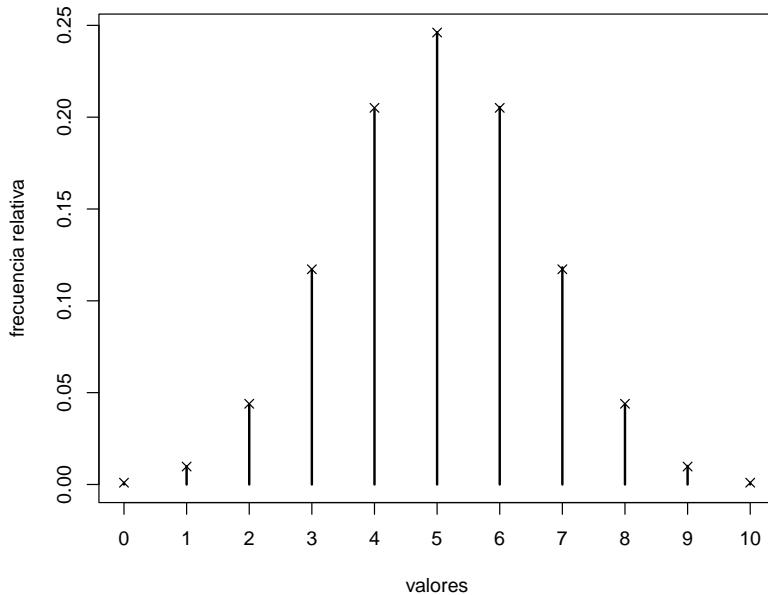


Figura 6.6: Comparación de las frecuencias relativas de los valores generados, mediante el método de alias, con las probabilidades teóricas.

6.4 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 6.1 (distribución de Poisson)

Una variable X con distribución de Poisson de parámetro λ , toma los valores $x_1 = 0, x_2 = 1, \dots$ con probabilidades:

$$p_j = P(X = x_j) = P(X = j - 1) = \frac{e^{-\lambda} \lambda^{j-1}}{(j-1)!}, \quad j = 1, 2, \dots$$

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, e.g. incluyendo una búsqueda secuencial en la cola de la distribución, para estos casos.

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), de esta forma la distribución de las simulaciones será aproximada.

6.5 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 6.2 (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 6.3 (distribución geométrica)

La función de masa de probabilidad de una distribución geométrica es:

$$P(X = j) = P(I = j + 1) = p(1 - p)^j, \quad j = 0, 1, \dots$$

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:

$$\begin{aligned} G(i) - G(i-1) &= 1 - e^{-\lambda i} - (1 - e^{-\lambda(i-1)}) = e^{-\lambda(i-1)} - e^{-\lambda i} \\ &= e^{-\lambda(i-1)}(1 - e^{-\lambda}) = (1 - e^{-\lambda})(e^{-\lambda})^{i-1} \\ &= p(1 - p)^{i-1}, \end{aligned}$$

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 $X = \lfloor T \rfloor$.

6.6 Otros métodos

- Aceptación-Rechazo: Este método también sería directamente aplicable al caso discreto. En principio habría que considerar una variable auxiliar discreta con el mismo soporte, pero también hay modificaciones para variables auxiliares continuas.
- Método de composición: este es otro método directamente aplicable y que se emplea en el método de Alias (descrito en la Sección 6.3)
- 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).

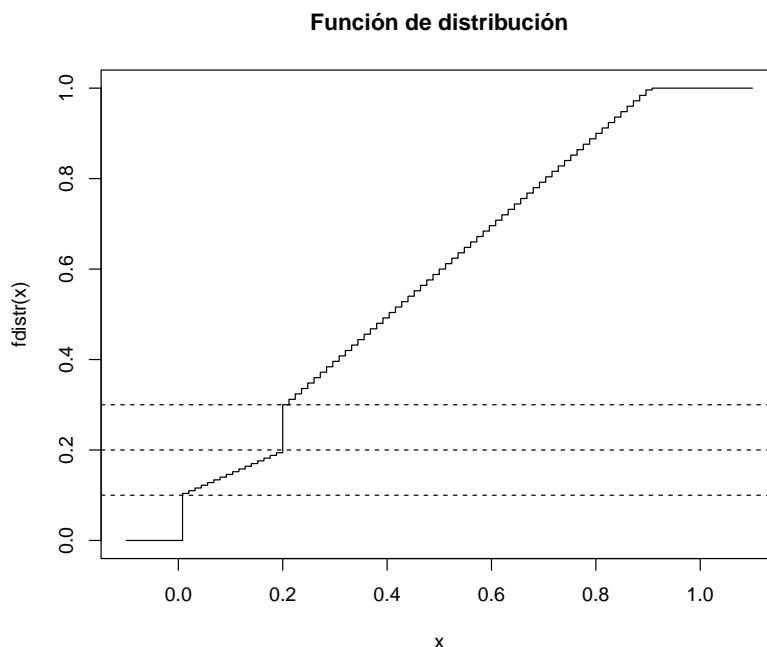
Considera la variable aleatoria con función de distribución dada por:

Ejercicio 6.5 (Simulación de una distribución mixta mediante el método de inversión generalizado)

$$F(x) = \begin{cases} 0 & \text{si } x < 0 \\ \frac{x}{2} + \frac{1}{10} & \text{si } x \in [0, \frac{1}{5}) \\ x + \frac{1}{10} & \text{si } x \in [\frac{1}{5}, \frac{9}{10}] \\ 1 & \text{en otro caso} \end{cases}$$

Función de distribución:

```
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 se complica un poco más pero el resultado es una función vectorial.
curve(fdistr(x), 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) Diseña un algoritmo basándote en el método de inversión generalizado para generar observaciones de esta variable.

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) Implementa el algoritmo en una función que permita generar $nsim$ valores de la variable.

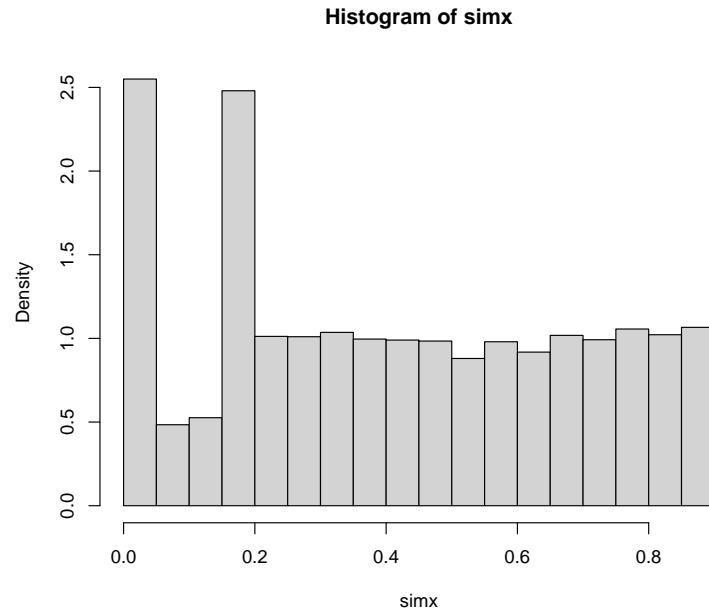
```
# Función cuantil:
fquant <- function(u)
  ifelse(u < 1/10, 0,
         ifelse(u < 2/10, 2*(u - 1/10),
                ifelse(u < 3/10, 1/5, u - 1/10) ) )
# Función para generar nsim valores:
rx <- function(nsim) fquant(runif(nsim))
```

Ejemplo:

```
set.seed(1)
nsim <- 10^4
system.time(simx <- rx(nsim))

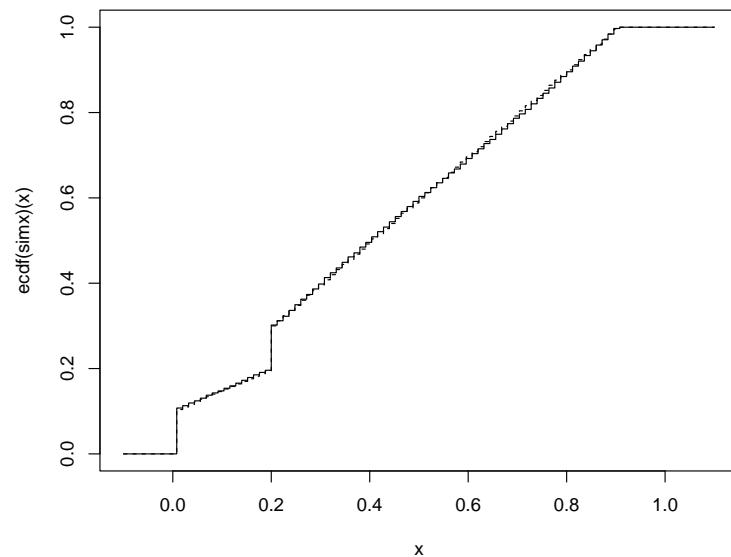
##      user  system elapsed
##        0       0       0

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)
```



Ejercicio 6.6 (propuesto)

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 $k = 20$.

- 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.
- Repetir el apartado anterior ordenando previamente las probabilidades en orden decreciente y también: empleando la función `sample` de R, mediante una tabla guía (con $k - 1$ subintervalos) y usando el método de Alias.

6.7 Métodos específicos para generación de distribuciones notables

Los comentarios al principio de la Sección 5.5 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 5.4).

La distribución binomial negativa, $BN(r, p)$, puede interpretarse como el número de fracasos antes del r -ésimo éxito² y su función de masa de probabilidad es

$$P(X = i) = \binom{i+r-1}{i} p^r (1-p)^i, \text{ para } i = 0, 1, \dots$$

A partir de la propiedad

$$X|_Y \sim Pois(Y), Y \sim \text{Gamma}\left(r, \frac{p}{1-p}\right) \Rightarrow X \sim BN(r, p)$$

se puede deducir el siguiente método específico de simulación.

Algoritmo condicional para simular la binomial negativa

- Simular $L \sim \text{Gamma}\left(r, \frac{p}{1-p}\right)$.
- Simular $X \sim Pois(L)$.
- 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.

²La 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.

Capítulo 7

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 7.7 se tratará brevemente la simulación de vectores aleatorios discretos y de tablas de contingencia, centrándose en el caso bidimensional.

7.1 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 7.1 (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 $\Sigma = \text{diag}(\sigma_1^2, \sigma_2^2, \dots, \sigma_d^2)$, 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 7.1 (de simulación de normales independientes)

1. Simular $Z_1, Z_2, \dots, Z_d \sim \mathcal{N}(0, 1)$ independientes.
2. Para $i = 1, 2, \dots, d$ 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)
```

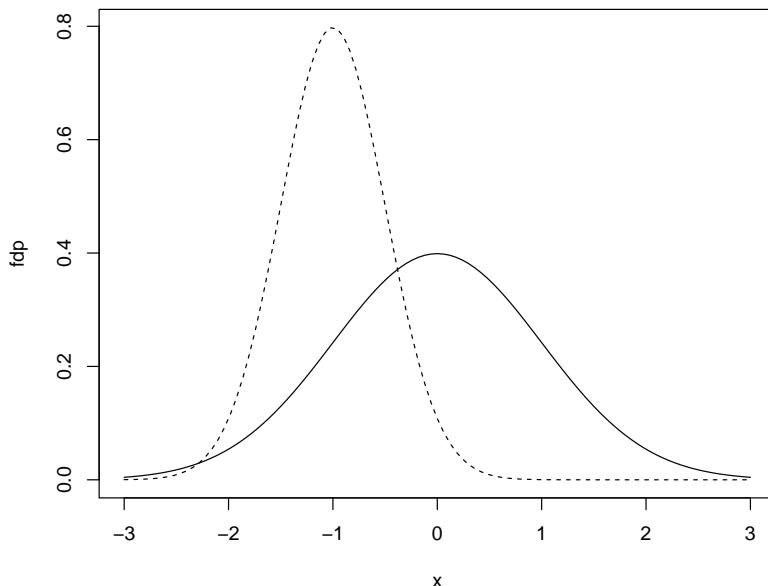


Figura 7.1: Densidades marginales de las componentes del Ejemplo 7.1.

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
```

7.2 El método de aceptación/rechazo

El algoritmo de aceptación-rechazo es el mismo que el del caso univariante descrito en la Sección 5.2, la única diferencia es que las densidades son multidimensionales. Supongamos que la densidad objetivo f y la densidad auxiliar g verifican:

$$f(x_1, x_2, \dots, x_d) \leq c \cdot g(x_1, x_2, \dots, x_d), \quad \forall \mathbf{x} = (x_1, x_2, \dots, x_d) \in \mathbb{R}^d.$$

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 7.2 (distribución bidimensional acotada)

Supongamos que estamos interesados en generar valores de una variable aleatoria bidimensional (X, Y) con función de densidad:

$$f(x, y) = \begin{cases} \frac{3}{16} (2 - (x^2 + y^2)) & \text{si } x \in [-1, 1] \text{ e } y \in [-1, 1] \\ 0 & \text{en otro caso} \end{cases}$$

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: $U \leq 1 - (T_1^2 + T_2^2)/2$.

Ejemplo 7.3 (distribución uniforme en la esfera)

Supongamos que el objetivo es simular puntos uniformemente distribuidos sobre la “esfera” unitaria d -dimensional (ver Figura 2.1):

$$C_d = \{(x_1, x_2, \dots, x_d) \in \mathbb{R}^d : x_1^2 + x_2^2 + \dots + x_d^2 \leq 1\}.$$

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 $\mathcal{U}([-1, 1] \times [-1, 1] \times \dots \times [-1, 1]) = \mathcal{U}([-1, 1]^d)$, dada por:

$$g(x_1, x_2, \dots, x_d) = \begin{cases} \frac{1}{2^d} & \text{si } x_i \in [-1, 1], \text{ para todo } i = 1, 2, \dots, d \\ 0 & \text{en otro caso} \end{cases}$$

La constante c óptima para la utilización del método de aceptación/rechazo es:

$$c = \max_{\{\mathbf{x}: g(\mathbf{x}) > 0\}} \frac{f(\mathbf{x})}{g(\mathbf{x})} = \frac{\frac{1}{V_d(1)}}{\frac{1}{2^d}} = \frac{2^d}{V_d(1)}$$

y la condición de aceptación $cUg(\mathbf{T}) \leq f(\mathbf{T})$ se convierte en:

$$\frac{2^d}{V_d(1)} U \frac{1}{2^d} \mathbf{1}_{[-1,1]^d}(\mathbf{T}) \leq \frac{1}{V_d(1)} \mathbf{1}_{C_d}(\mathbf{T}),$$

o, lo que es lo mismo, $U \mathbf{1}_{[-1,1]^d}(\mathbf{T}) \leq \mathbf{1}_{C_d}(\mathbf{T})$. 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 $\mathbf{1}_{[-1,1]^d}(\mathbf{T}) = \mathbf{1}_{C_d}(\mathbf{T})$, es decir, a que $\mathbf{T} \in C_d$, por tanto, a que se verifique:

$$T_1^2 + T_2^2 + \dots + T_d^2 \leq 1.$$

Por otra parte, la simulación de $T \sim \mathcal{U}([-1, 1]^d)$ puede hacerse trivialmente mediante $T_i \sim \mathcal{U}(-1, 1)$ para cada $i = 1, 2, \dots, d$, ya que las componentes son independientes. Como el valor de U es superfluo en este caso, el algoritmo queda:

1. Simular $V_1, V_2, \dots, V_d \sim \mathcal{U}(0, 1)$ independientes.
2. Para $i = 1, 2, \dots, d$ hacer $T_i = 2V_i - 1$.
3. Si $T_1^2 + T_2^2 + \dots + T_d^2 > 1$ entonces volver al paso 1.
4. Devolver $\mathbf{X} = (T_1, T_2, \dots, T_d)^t$.

Ver el Ejercicio 2.1 para el caso de $d = 2$.

Usando las fórmulas del “volumen” de una “esfera” d -dimensional:

$$V_d(r) = \begin{cases} \frac{\pi^{d/2} r^d}{(d/2)!} & \text{si } d \text{ es par} \\ \frac{2^{\lfloor \frac{d}{2} \rfloor + 1} \pi^{\lfloor \frac{d}{2} \rfloor} r^d}{1 \cdot 3 \cdot 5 \cdots d} & \text{si } d \text{ es impar} \end{cases}$$

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 tremadamente 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 y Costa, 2020, Sección 1.4).

7.3 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 7.1

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 7.2 (de simulación de una normal multivariante)

1. Obtener la factorización de Cholesky $\Sigma = LL^t$.
2. Simular $\mathbf{Z} = (Z_1, Z_2, \dots, Z_d)$ 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 7.4 (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)

```

Alternativamente se podría emplear, por ejemplo, la función `mvrnorm` del paquete MASS que emplea la factorización espectral (`eigen`) (y que tiene en cuenta una tolerancia relativa para corregir 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
##     }
## }

```

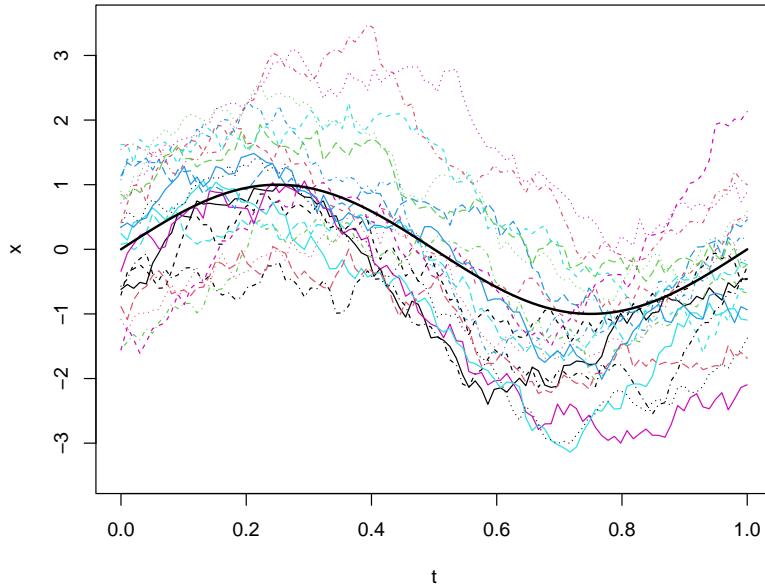


Figura 7.2: Realizaciones del proceso funcional del Ejemplo 7.4, obtenidas a partir de la factorización de Cholesky.

```

##      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: 0x000000002adcf78>
## <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.

7.4 Método de las distribuciones condicionadas

Teniendo en cuenta que:

$$f(x_1, x_2, \dots, x_d) = f_1(x_1) \cdot f_2(x_2|x_1) \cdots f_d(x_d|x_1, x_2, \dots, x_{d-1}),$$

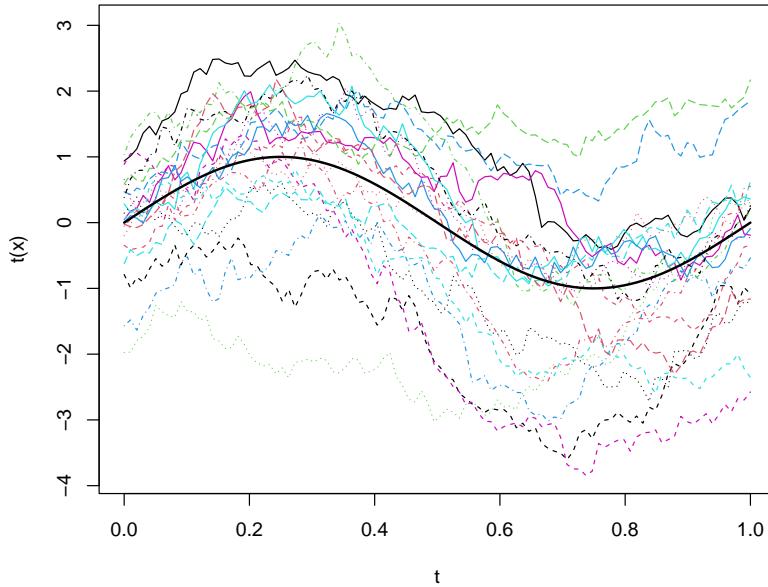


Figura 7.3: Realizaciones del proceso funcional del Ejemplo 7.4, obtenidas empleando la función MASS::mvrnorm.

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 7.3 (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 $\mathbf{X} = (X_1, X_2, \dots, X_d)$.

Nota: En las simulaciones unidimensionales se puede tener en cuenta que $f_i(x_i|x_1, x_2, \dots, x_{i-1}) \propto f_{1,\dots,i}(x_1, x_2, \dots, x_i)$.

Ejemplo 7.5 (distribución uniforme en el círculo unitario)

Se trata de la distribución bidimensional continua con densidad constante en el círculo:

$$C = \{(x_1, x_2) \in \mathbb{R}^2 : x_1^2 + x_2^2 \leq 1\}.$$

Su función de densidad viene dada por:

$$f(x_1, x_2) = \begin{cases} \frac{1}{\pi} & \text{si } (x_1, x_2) \in C \\ 0 & \text{si } (x_1, x_2) \notin C \end{cases}$$

La densidad marginal de la primera variable resulta:

$$f_1(x_1) = \int_{-\sqrt{1-x_1^2}}^{+\sqrt{1-x_1^2}} \frac{1}{\pi} dx_2 = \frac{2\sqrt{1-x_1^2}}{\pi} \text{ si } x_1 \in [-1, 1],$$

es decir:

$$f_1(x_1) = \begin{cases} \frac{2}{\pi}\sqrt{1-x_1^2} & \text{si } x_1 \in [-1, 1] \\ 0 & \text{si } x_1 \notin [-1, 1] \end{cases}$$

Además:

$$f_2(x_2|x_1) = \frac{f(x_1, x_2)}{f_1(x_1)} = \frac{\frac{1}{\pi}}{\frac{2\sqrt{1-x_1^2}}{\pi}} = \frac{1}{2\sqrt{1-x_1^2}}, \text{ si } x_2 \in \left[-\sqrt{1-x_1^2}, \sqrt{1-x_1^2}\right]$$

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 7.6 (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:

$$f(x_1, x_2) = \frac{1}{2\pi\sigma_1\sigma_2\sqrt{1-\rho^2}} \exp\left(-\frac{1}{2(1-\rho^2)}\left(\frac{(x_1-\mu_1)^2}{\sigma_1^2} + \frac{(x_2-\mu_2)^2}{\sigma_2^2} - \frac{2\rho(x_1-\mu_1)(x_2-\mu_2)}{\sigma_1\sigma_2}\right)\right)$$

de donde se deduce (ver e.g. Cao, 2002, p.88; o ecuaciones (7.1) y (7.2) en la Sección 7.5.1) que:

$$\begin{aligned} f_1(x_1) &= \frac{1}{\sigma_1\sqrt{2\pi}} \exp\left(-\frac{(x_1-\mu_1)^2}{2\sigma_1^2}\right) \\ f_2(x_2|x_1) &= \frac{f(x_1, x_2)}{f_1(x_1)} \\ &= \frac{1}{\sigma_2\sqrt{2\pi(1-\rho^2)}} \exp\left(-\frac{(x_2-\mu_2 - \frac{\sigma_2}{\sigma_1}\rho(x_1-\mu_1))^2}{2\sigma_2^2(1-\rho^2)}\right) \end{aligned}$$

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 7.4 (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 7.5.1.

Ejercicio 7.1

Considerando la variable aleatoria bidimensional del Ejemplo 7.2 y teniendo en cuenta que la densidad marginal de la variable X es:

$$f_X(x) = \begin{cases} \frac{1}{8}(5 - 3x^2) & \text{si } x \in [-1, 1] \\ 0 & \text{en otro caso} \end{cases}$$

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).

7.5 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 7.5 (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.

7.5.1 Simulación condicional de una normal multivariante

Si $\mathbf{X} \sim \mathcal{N}_d(\mu, \Sigma)$ es tal que \mathbf{X} , μ y Σ se partitionan 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:

$$E(\mathbf{X}_2|\mathbf{X}_1) = \mu_2 + \Sigma_{21}\Sigma_{11}^{-1}(\mathbf{X}_1 - \mu_1), \quad (7.1)$$

$$Cov(\mathbf{X}_2|\mathbf{X}_1) = \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}. \quad (7.2)$$

Nota: La ecuación (7.1) 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 (7.2) son las correspondientes varianzas kriging).

Ejemplo 7.7 (simulación condicional de datos funcionales o temporales)

Continuando con el Ejemplo 7.4 anterior, consideramos los primeros valores de una simulación incondicional como los datos:

```
idata <- t < 0.5
# idata <- t < 0.2 | t > 0.8
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))
```

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).
```

Las simulaciones condicionales se obtendrán de forma análoga (Figura 7.4):

```
z <- matrix(rnorm(nsim * n2), nrow = n2)
y <- kpred + t(chol(kcov)) %*% z
# Representación gráfica:
plot(t, mu, type = "l", lwd = 2, ylab = "y", ylim = c(-3.5, 3.5)) # media teórica
lines(t[idata], data) # datos
# y <- rep(NA, n)
```

```
# y[idata] <- data
# lines(t, y)
matplot(t[!idata], y, type = "l", add = TRUE) # simulaciones condicionales
lines(t[!idata], kpred, lwd = 2, lty = 2) # media condicional (predicción kriging)
```

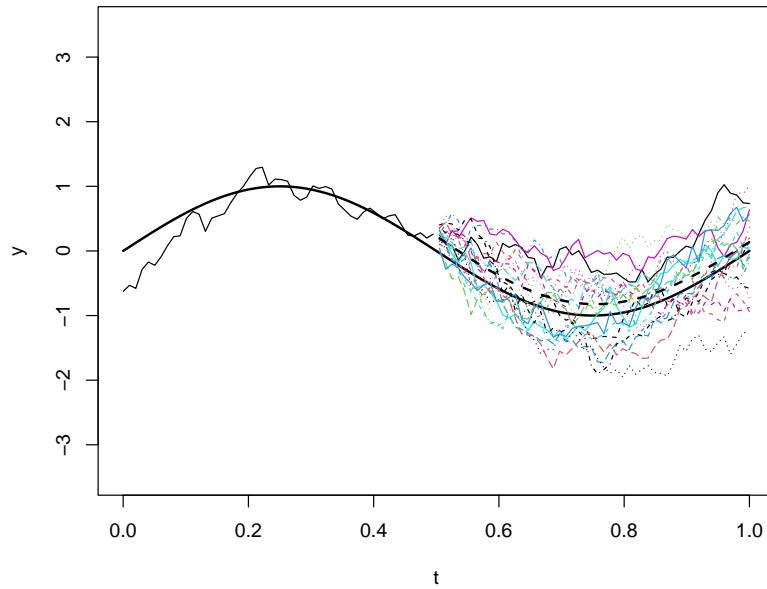


Figura 7.4: Realizaciones condicionales del proceso funcional del Ejemplo 7.7.

Ejemplo 7.8 (simulación condicional de datos espaciales)

Consideramos un proceso espacial bidimensional normal $Z(\mathbf{s}) \equiv Z(x, y)$ de media 0 y covariograma exponencial:

$$\text{Cov}(Z(\mathbf{s}_1), Z(\mathbf{s}_2)) = C(\|\mathbf{s}_1 - \mathbf{s}_2\|) = e^{-\|\mathbf{s}_1 - \mathbf{s}_2\|}.$$

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`:

```
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 función `grf` 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)

```

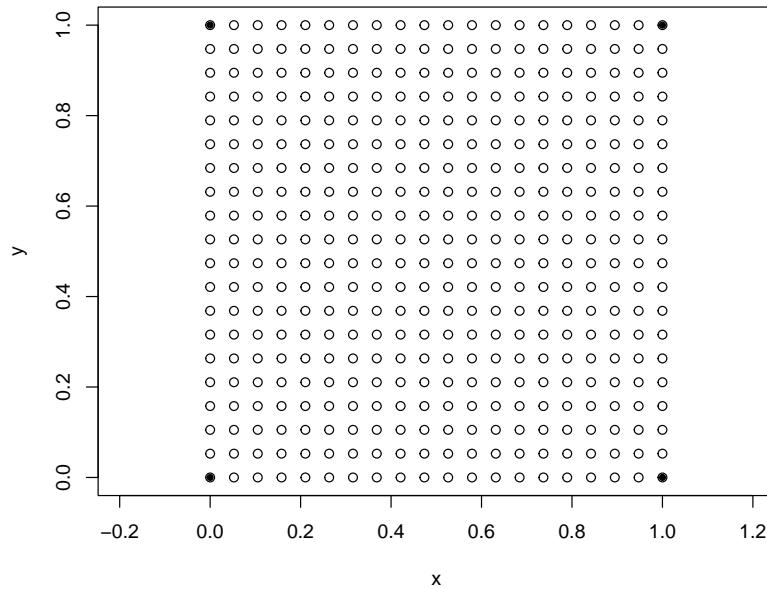


Figura 7.5: Posiciones espaciales de las simulaciones condicionales (y las de los datos).

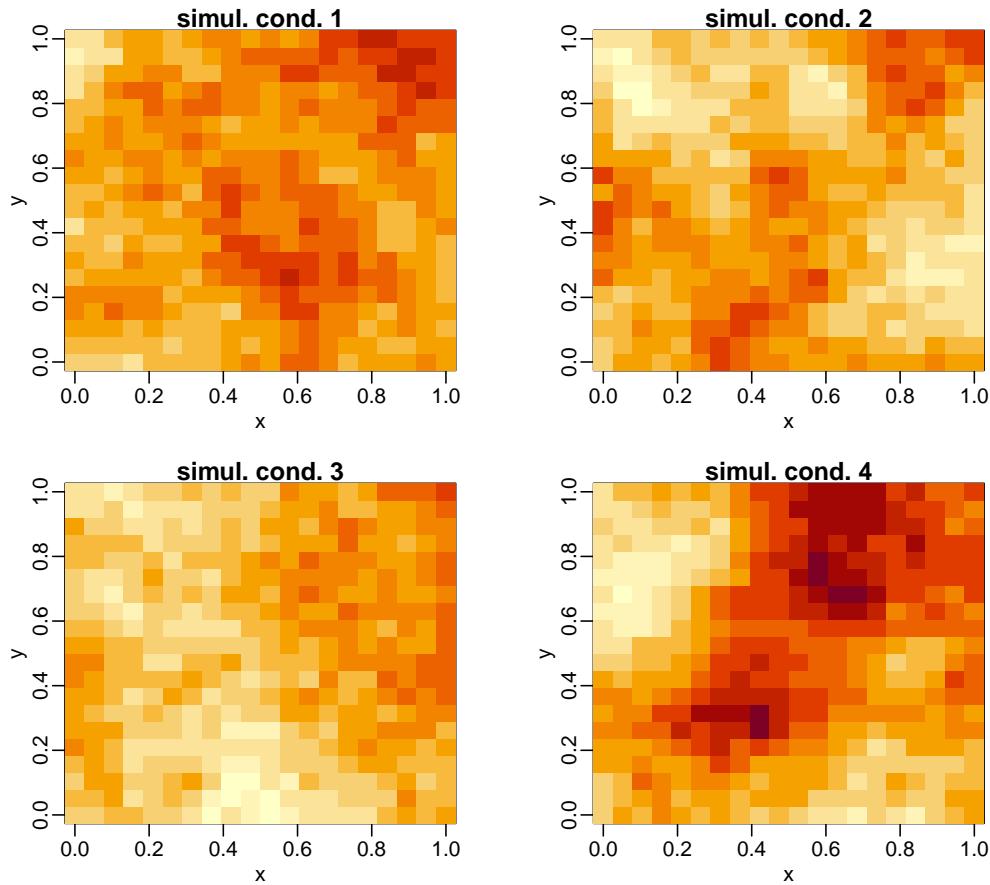
```
# Simulación condicional
set.seed(1)
nsim.cond <- 4
s.out <- output.control(n.predictive = nsim.cond)
kc <- krige.conv(z, loc = new.s, output = s.out,
                  krige = krige.control(type.krige="SK", beta = 0, cov.pars = c(1, 1)))

## 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.

```
# 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)
```

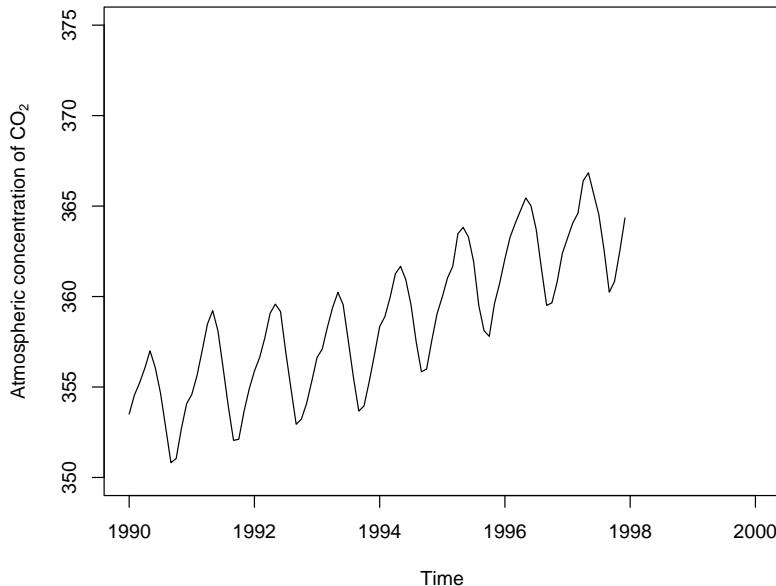
7.5.2 Simulación condicional a partir de un modelo ajustado

En la práctica normalmente se ajusta un modelo a los datos observados y posteriormente se obtienen las simulaciones condicionadas empleando el modelo ajustado.

En R se incluye una función genérica¹ `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 generalizados 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))
```

¹Se 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")
```

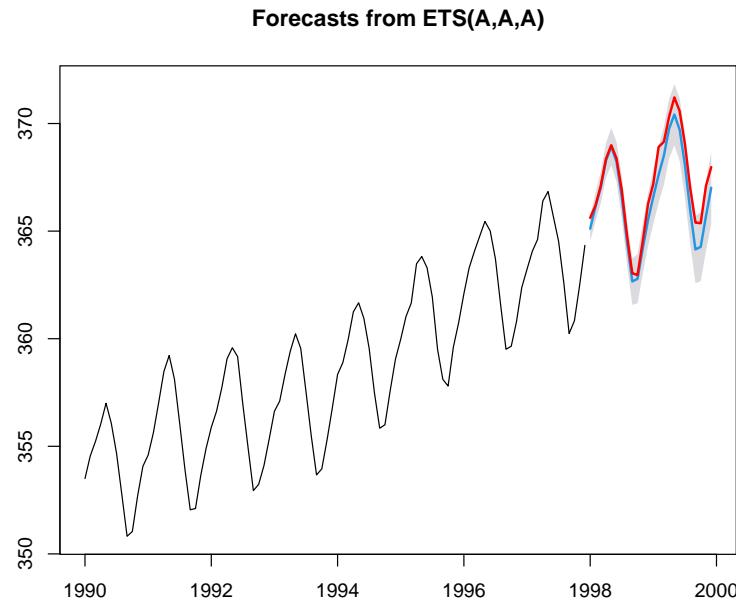


Figura 7.6: Ejemplo de una serie de tiempo (datos observados de co2 en el observatorio Mauna Loa), predicciones futuras (en azul; media distribución condicional) y simulación condicional (en rojo) obtenidas a partir de un modelo ajustado.

Para más detalles ver Hyndman y Athanasopoulos (2018, secciones 4.3 y 11.4).

7.6 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).

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 7.1 (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:

$$F(x, y) = C(F_1(x), F_2(y)), \quad \forall x, y \in \mathbb{R}.$$

Además, si $F_1(\cdot)$ y $F_2(\cdot)$ son continuas entonces $C(\cdot, \cdot)$ es única. Siendo el recíproco también cierto.

7.6.1 Cópulas Arquimediana

Además de las cópulas Gausianas, es una de las familias de cópulas más utilizadas. Son de la forma:

$$C(x_1, x_2, \dots, x_d) = \Psi^{-1} \left(\sum_{i=1}^d \Psi(F_i(x_i)) \right),$$

siendo Ψ su función generadora.

Una condición suficiente para que sea una cópula multidimensional válida es que $\Psi(1) = 0$, $\lim_{x \rightarrow 0} \Psi(x) = \infty$, $\Psi'(x) < 0$ y $\Psi''(x) > 0$.

Ejemplos:

- Cúpula producto o independiente: $\Psi(x) = -\ln(x)$,

$$F(x, y) = F_1(x)F_2(y).$$

- Cúpula de Clayton: $\Psi(x) = \frac{1}{\alpha}(x^{-\alpha} - 1); \alpha > 0$,

$$F(x, y) = (F_1(x)^{-\alpha} + F_2(y)^{-\alpha} - 1)^{-1/\alpha}.$$

- Cúpula de Gumbel: $\Psi(x) = (-\ln(x))^\alpha; \alpha \geq 1$

7.6.2 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 $C_u(v) \equiv C_2(v|u)$ 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 7.4.

Algoritmo 7.6 (de simulación bidimensional mediante cópulas)

1. Generar $U, W \sim \mathcal{U}(0, 1)$
2. Obtener $V = C_U^{-1}(W)$
3. Devolver $(F_1^{-1}(U), F_2^{-1}(V))$

Ejercicio 7.2 (Cúpula bidimensional de Clayton)

Consideramos una variable aleatoria bidimensional con distribuciones marginales uniformes y distribución bidimensional determinada por la cúpula de Clayton.

- a) Teniendo en cuenta que en este caso:

$$C_u^{-1}(w) \equiv \left(w^{-\alpha} (w^{-\frac{\alpha}{\alpha+1}} - 1) + 1 \right)^{-\frac{1}{\alpha}},$$

diseñar una rutina que permita 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)
}
```

- b) Utilizando la rutina anterior generar una muestra de tamaño 10000 y representar gráficamente los valores obtenidos y sus distribuciones marginales.

```
set.seed(54321)
rcunif <- rcclayton(2, 10000)
plot(rcunif, xlab = "u", ylab = "v")
```

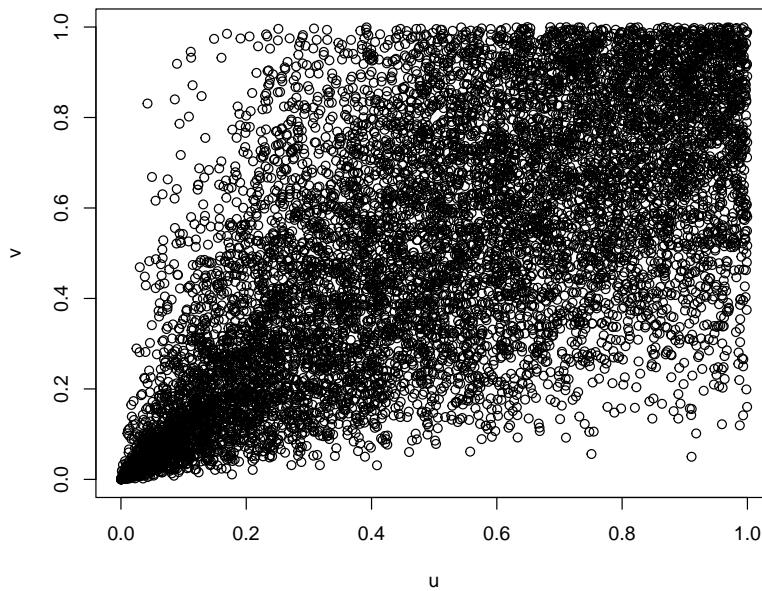


Figura 7.7: Gráfico de dispersión de los valores generados con distribución bidimensional de Clayton.

Representar la densidad conjunta (con `sm::sm.density()`) y las marginales [Figuras: 7.10, 7.11]:

```
# Densidad conjunta
# if(!require(sm)) stop('Required package `sm` not installed.')
sm::sm.density(rcunif, xlab = "u", ylab = "v", zlab = "Density")

# Distribuciones marginales
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: 7.10, 7.11]:

```
if(!require(copula)) stop('Required package `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")
```

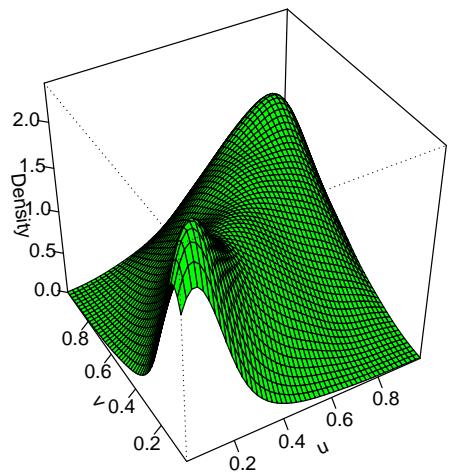


Figura 7.8: Densidad conjunta de los valores generados con distribución bidimensional de Clayton.

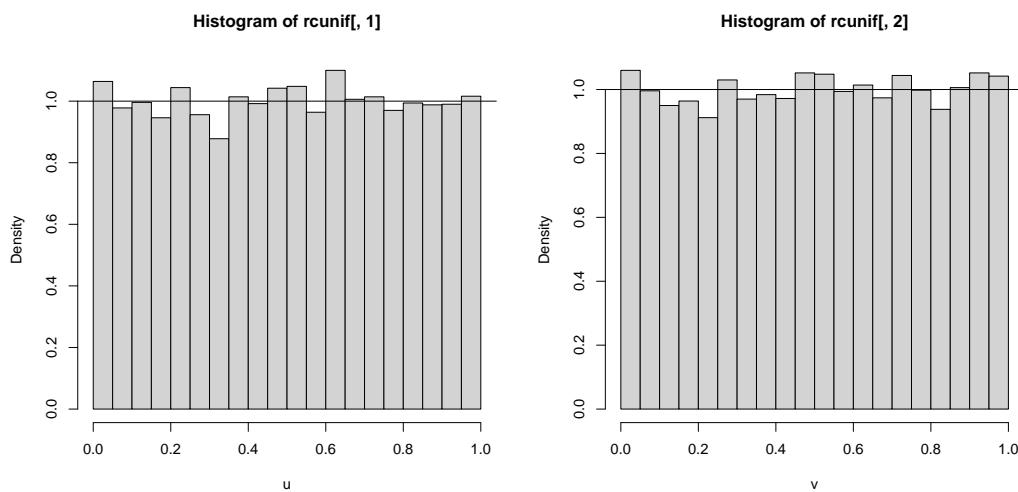


Figura 7.9: Distribuciones marginales de los valores generados con distribución bidimensional de Clayton.

- c) A partir de la muestra anterior generar una muestra de una v.a. bidimensional con distribuciones marginales exponenciales de parámetros 1 y 2 respectivamente (y distribución bidimensional determinada por la cópula de Clayton).

```
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)
```

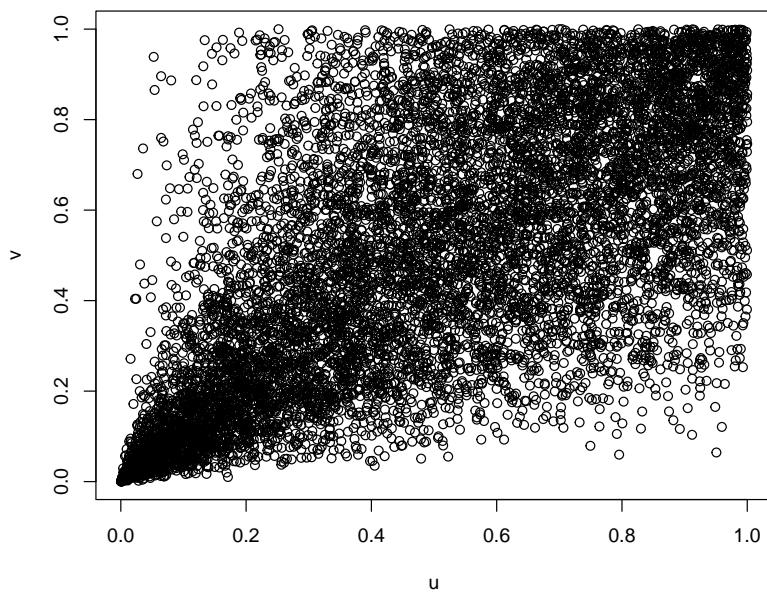


Figura 7.10: Gráfico de dispersión de los valores generados con distribución bidimensional de Clayton empleando el paquete ‘copula’.

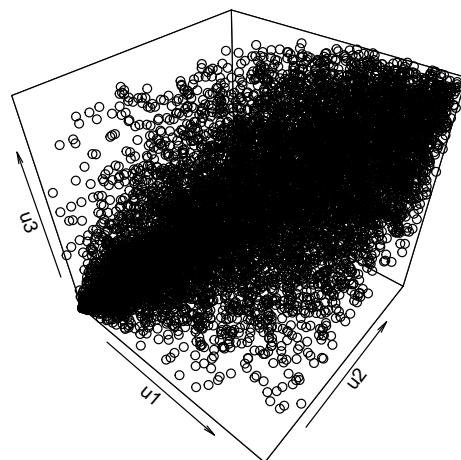


Figura 7.11: Gráfico de dispersión de los valores generados con distribución tridimensional de Clayton empleando el paquete ‘copula’.

```
hist(rcexp[,2], freq = FALSE, xlab = "exp2")
curve(dexp(x, 2), add = TRUE)
```

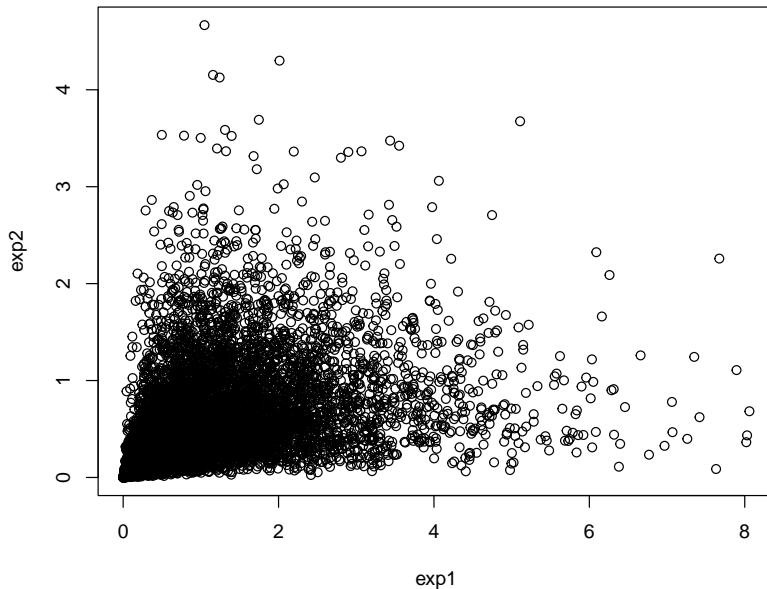


Figura 7.12: Gráfico de dispersión de los valores generados con distribución exponencial y dependencia definida por la cópula de Clayton.

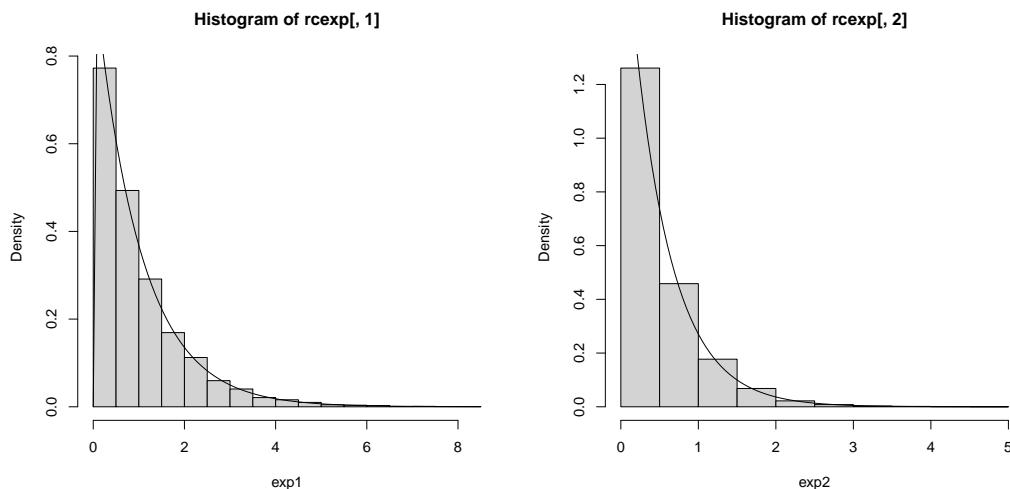


Figura 7.13: Distribuciones marginales exponenciales de los valores generados con dependencia definida por la cópula de Clayton.

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

7.7 Simulación de distribuciones multivariantes discretas

7.7.1 Métodos de codificación o etiquetado para variables 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 índice unidimensional equivalente al índice multidimensional, mediante una función de etiquetado $l(\mathbf{i}) = l(i_1, i_2, \dots, i_d) \in \mathbb{N}$.

Si la variable discreta multidimensional tiene soporte finito, este tipo de recodificación se puede hacer de forma automática en R cambiando simplemente el indexado² (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:

```

z <- 11:18
xy <- matrix(z, ncol = 2)
xy

##      [,1] [,2]
## [1,]    11   15
## [2,]    12   16
## [3,]    13   17
## [4,]    14   18
z <- as.vector(xy)
z

## [1] 11 12 13 14 15 16 17 18
i1d <- seq_along(z)
i1d

## [1] 1 2 3 4 5 6 7 8
# Cálculo del índice bidimensional (inversa de la función de etiquetado: 1d -> 2d)
nx <- nrow(xy)
linv <- function(k) cbind((k - 1) %% nx + 1, floor((k - 1)/nx) + 1)
i2d <- linv(i1d)
# i2d <- arrayInd(i1d, dim(xy))
i2d

##      [,1] [,2]
## [1,]    1    1
## [2,]    2    1
## [3,]    3    1
## [4,]    4    1
## [5,]    1    2
## [6,]    2    2
## [7,]    3    2
## [8,]    4    2
xy[i2d]

## [1] 11 12 13 14 15 16 17 18
# Cálculo del índice unidimensional (función de etiquetado: 2d -> 1d)
l <- function(i, j) nx*(j-1) + i
l(2, 1)

## [1] 2
l(2, 2)

## [1] 6

```

²En R podemos obtener el índice multidimensional empleando la función `arrayInd(ind, .dim, ...)`, siendo `ind` un vector de índices unidimensionales.

```
i1d <- mapply(l, i2d[, 1], i2d[, 2])
i1d

## [1] 1 2 3 4 5 6 7 8
z[i1d]

## [1] 11 12 13 14 15 16 17 18
```

Realmente lo que ocurre es que internamente un objeto `matrix` o `array` está almacenado como un vector y R admite un indexado multidimensional si está presente un atributo `dim`:

```
dim(z) <- c(4, 2)
z

##      [,1] [,2]
## [1,]    11   15
## [2,]    12   16
## [3,]    13   17
## [4,]    14   18

dim(z) <- c(2, 2, 2)
z

## , , 1
##
##      [,1] [,2]
## [1,]    11   13
## [2,]    12   14
##
## , , 2
##
##      [,1] [,2]
## [1,]    15   17
## [2,]    16   18

dim(z) <- NULL
z

## [1] 11 12 13 14 15 16 17 18
```

Si la variable discreta multidimensional no tiene soporte finito (tampoco se podría guardar la función de masa de probabilidad en una tabla), se podrían emplear métodos de codificación más avanzados (ver Cao, 2002, Sección 6.3).

7.7.2 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 Hall, 1994. Analysis of defectivity of semiconductor wafers by contingency table. Proceedings of the Institute of Environmental Sciences 1, 177-183).

```
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 definimos las probabilidades a partir de las frecuencias:

```
df$p <- df$n/sum(df$n)
df

##      n particulas calidad      p
## 1 320      no    buena 0.7111111
## 2 14       si    buena 0.0311111
## 3 80       no    mala  0.1777778
## 4 36       si    mala  0.0800000
```

En formato tabla:

```
pij <- tabla/sum(tabla)
pij

##      particulas
## calidad      no      si
## buena 0.7111111 0.0311111
## mala  0.1777778 0.0800000
```

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] 0.7111111 0.0311111 0.1777778 0.0800000
```

Si las probabilidades estuviesen en una matriz, las convertiríamos a un vector con:

```
as.vector(pij)

## [1] 0.7111111 0.1777778 0.0311111 0.0800000
```

Si simulamos la variable unidimensional:

```
set.seed(1)
nsim <- 20
rz <- sample(z, nsim, replace = TRUE, prob = pz)
```

Podríamos obtener simulaciones bidimensionales, por ejemplo:

```
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
```

7.7.3 Simulación de tablas de contingencia

El código anterior puede ser empleado para simular tablas de contingencia. Aunque en estos casos se suele fijar el total de la tabla (o incluso las frecuencias marginales). En este caso, sólo habría que fijar el número de simulaciones al total de la tabla:

```
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) # Tabla de frecuencias bidimensional
```

```
##      [,1] [,2]
## [1,] 321   78
## [2,] 15    36
```

Aunque puede ser preferible emplear directamente `rmultinom` si se van a generar muchas:

```
ntsim <- 1000
ratablas <- rmultinom(ntsim, sum(n), pz)
ratablas[ , 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
```

Por ejemplo, si se quiere simular bajo independencia, estimando las probabilidades a partir de la tabla:

`tabla`

```
##      particulas
## calidad no si
## buena 320 14
## mala  80 36
```

Consideraríamos como probabilidades:

```
pind <- (rowSums(tabla) %o% colSums(tabla))/(sum(tabla)^2)
matrix(pind, nrow = nrow(tabla))
```

```
##      [,1]      [,2]
## [1,] 0.6597531 0.08246914
## [2,] 0.2291358 0.02864198

ratablas <- rmultinom(ntsim, sum(n), pind)
ratablas[ , 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
```

Para realizar el contraste de independencia:

```
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
```

Ejercicio 7.3 (Distribución del estadístico chi-cuadrado de independencia)

Aproximar por simulación la distribución (exacta) del estadístico chi-cuadrado bajo independencia.

```
sim.stat <- apply(ratablas, 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 8.3 del siguiente capítulo, podríamos aproximar el p -valor del contraste de independencia a partir de esta distribución:

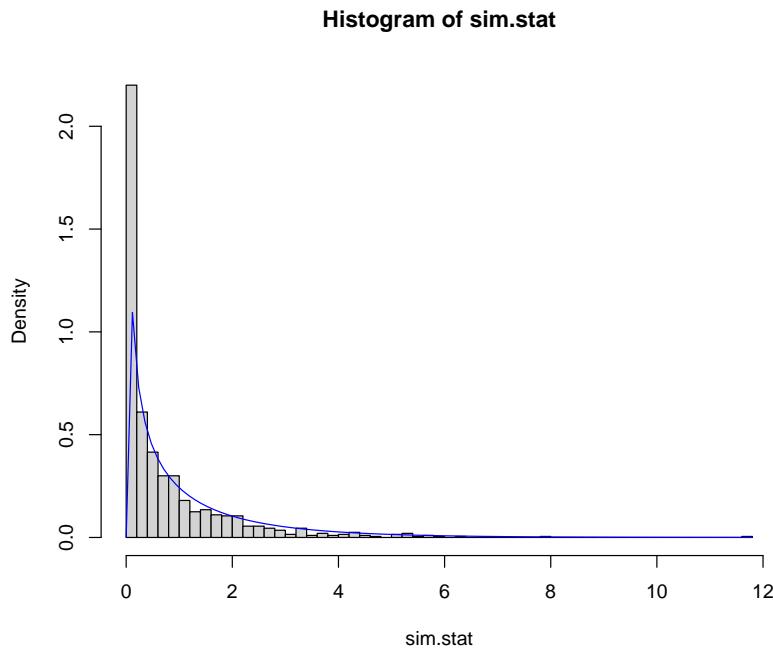


Figura 7.14: Aproximación Monte Carlo de la distribución del estadístico chi-cuadrado bajo independencia.

```
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 8

Aplicaciones en Inferencia Estadística

Como ya se comentó en la introducción muchas de las aplicaciones de la simulación serían de utilidad en Estadística:

- Muestreo, aproximación de distribuciones, remuestreo, ...
- Optimización: Algoritmos genéticos, ...
- Análisis numérico: Evaluación de expresiones, ...
- ...

En este capítulo nos centraremos en algunas de las aplicaciones en inferencia estadística:

- Distribución de estimadores puntuales/estadísticos:
 - Aproximación de la distribución.
 - * Aproximación de características de la distribución.
 - * Validez 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 remuestreo bootstrap.
- Inferencia Bayesiana
- ...

En el siguiente capítulo trararemos la Integración y Optimización Monte Carlo.

Observación: En este capítulo 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 2, 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.

8.1 Distribución en el muestreo

Ejercicio 8.1 (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:

$$\hat{\mu} = \bar{X} = \frac{1}{n} \sum_{i=1}^n X_i$$

es:

$$\bar{X} \sim N\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$$

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")
```

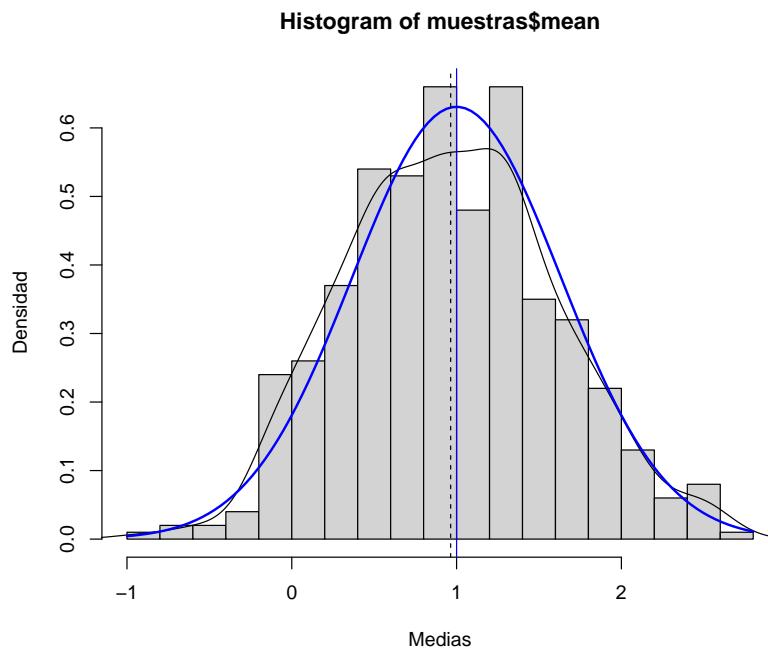


Figura 8.1: Distribución de la media muestral de una distribución normal.

Ejercicio 8.2 (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 $\hat{\mu} = \bar{X}$ se aproxima a la normalidad:

$$\bar{X} \xrightarrow{n \rightarrow \infty} N\left(\mu, \frac{\sigma}{\sqrt{n}}\right)$$

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 8.1 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?

```
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")
```

- 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`.

8.2 Intervalos de confianza

Ejercicio 8.3 (Intervalo de confianza para la media)

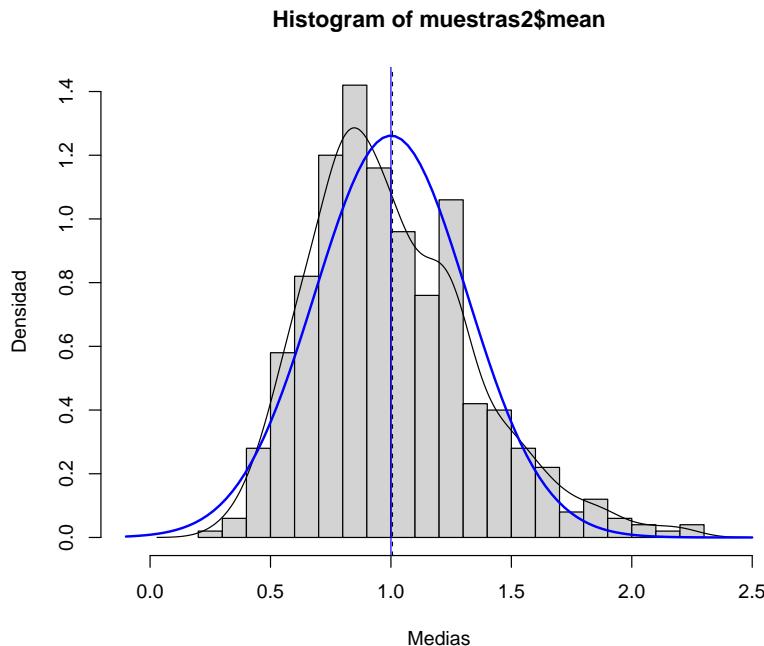


Figura 8.2: Distribución de la media muestral de una distribución exponencial y distribución asintótica.

A partir del enunciado del Ejercicio 8.1, se deduce que el intervalo de confianza (de nivel $1 - \alpha$) para la media μ de una población normal con varianza conocida es:

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

La idea es que el $100(1 - \alpha)\%$ de los intervalos así construidos contendrán 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) # N° 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)
```

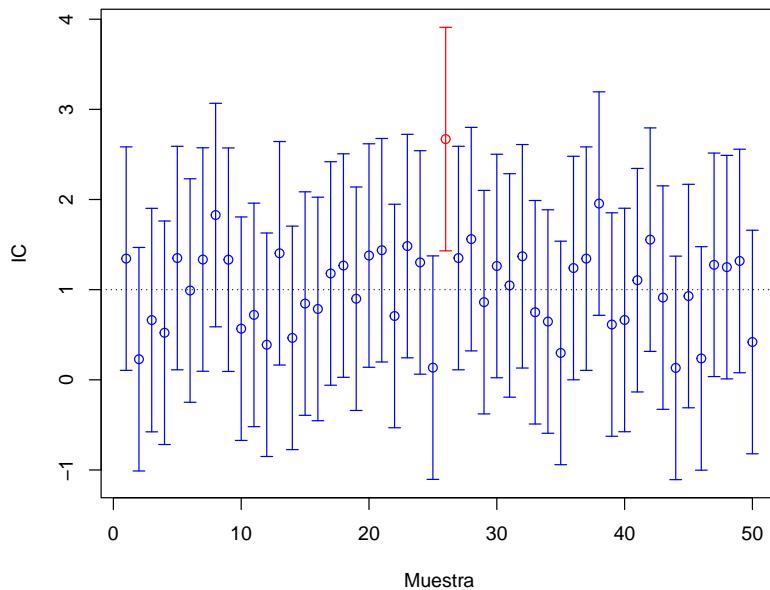


Figura 8.3: Cobertura de las estimaciones por IC.

```
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) # N° 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)
```

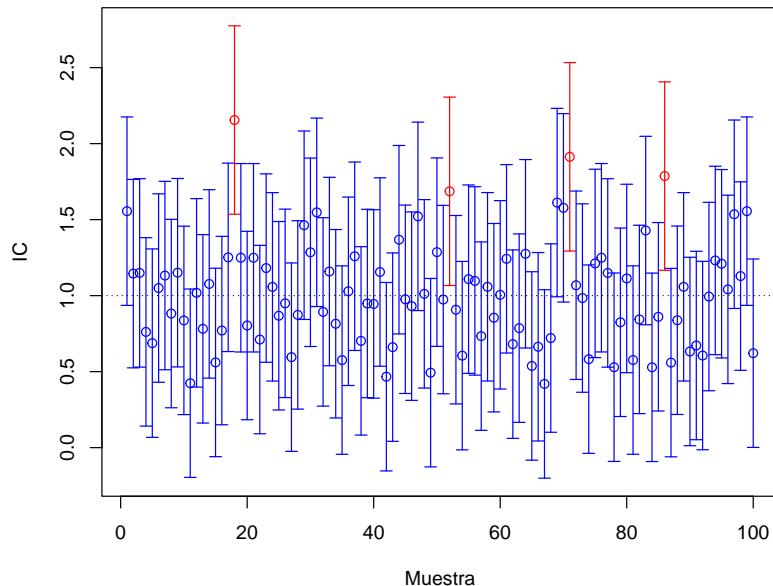


Figura 8.4: Cobertura de las estimaciones por IC (bajo normalidad).

```
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 al final de este capítulo.

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...):

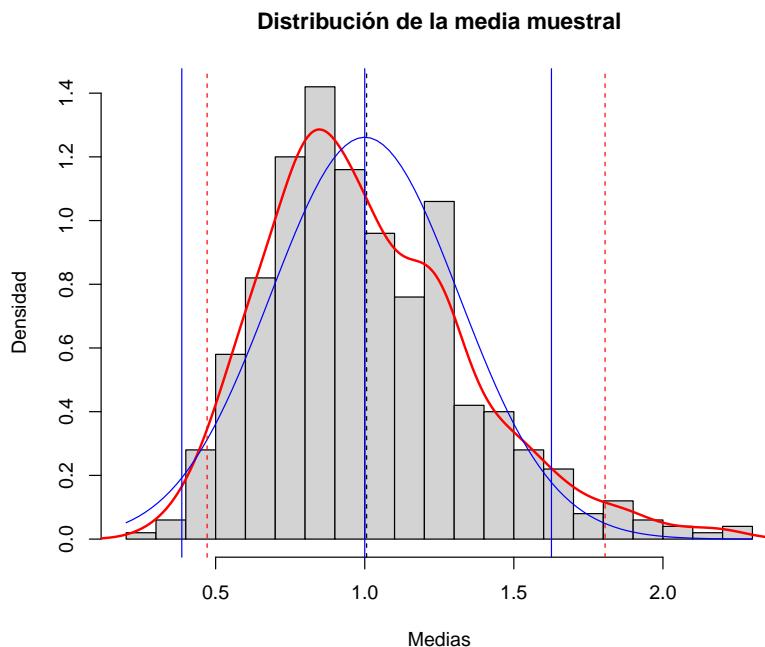
```
# Distribución de la media muestral
hist(muestras2$mean, freq=FALSE, breaks="FD",
      main="Distribución de la media muestral", xlab="Medias", ylab="Densidad")
# Densidad observada (estimación)
```

```

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 8.4 (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 éxitos y a fracasos. Así el intervalo de confianza al $(1 - \alpha) 100\%$

para una proporción mejorada es:

$$IC_{1-\alpha}^a(p) = \left(\tilde{p} - z_{1-\alpha/2} \sqrt{\frac{\tilde{p}(1-\tilde{p})}{\tilde{n}}}, \tilde{p} + z_{1-\alpha/2} \sqrt{\frac{\tilde{p}(1-\tilde{p})}{\tilde{n}}} \right),$$

siendo $\tilde{n} = n + 2a$, $\tilde{p} = \frac{np + a}{\tilde{n}}$.

En el caso de $a = 2$ se denomina IC Agresti-Coull.

- 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:

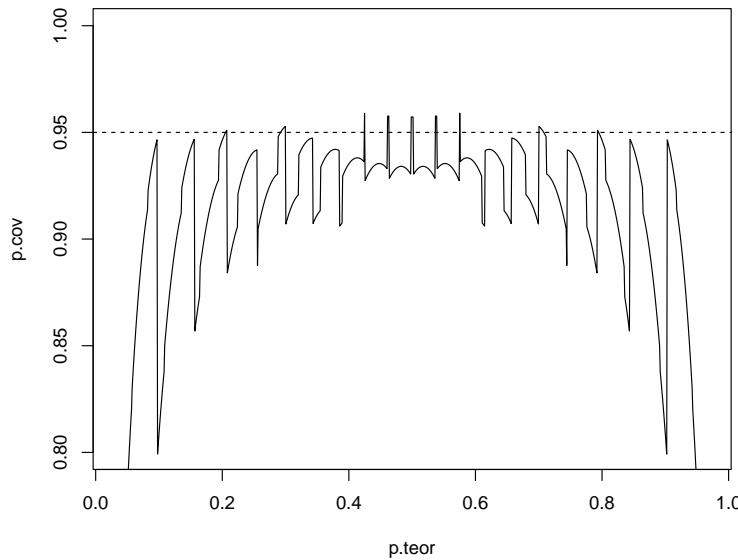
```
x <- 0:n
p.est <- (x + adj)/(n + 2 * adj)
ic.err <- qnorm(1 - alpha/2) * sqrt(p.est * (1 - p.est)/(n + 2 * adj))
lcl <- p.est - ic.err
ucl <- p.est + ic.err
```

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 ($a = 2$).

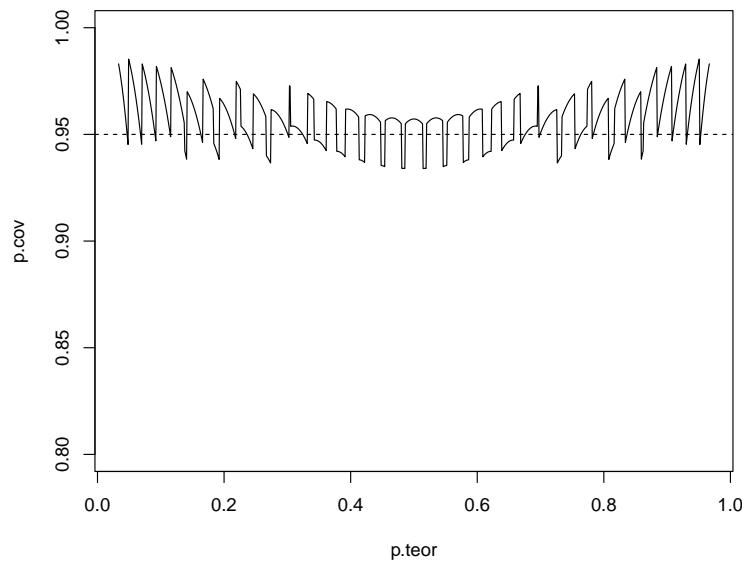
Parámetros:

```

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)

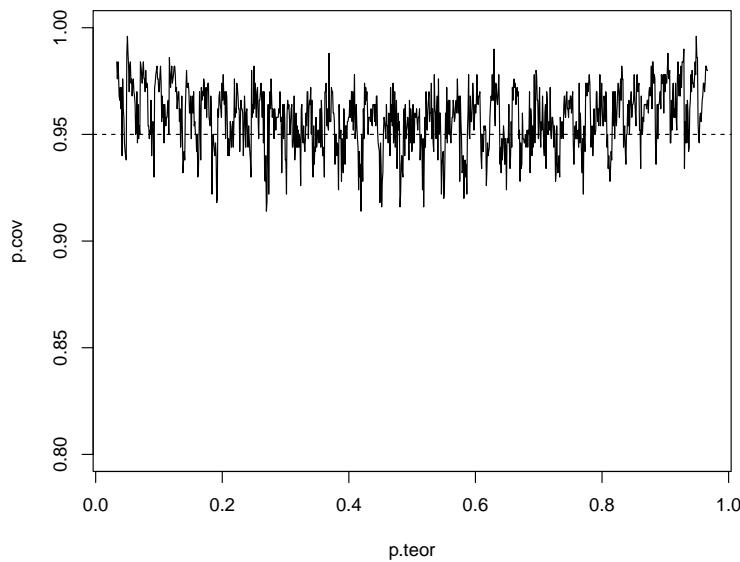
set.seed(54321)
nsim <- 500
# Probabilidades teóricas:
m <- 1000
p.teor <- seq(1/n, 1 - 1/n, length = m)
```

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.

8.3 Contrastes de hipótesis

Ver Capítulo 5 de Cao y Fernández-Casal (2020).

Ejercicio 8.5 (Test de Kolmogorov-Smirnov)

En la Sección 3.3 del Tema 3 se propuso el análisis de la bondad de ajuste de un generador de números pseudo-aleatorios mediante el test de Kolmogorov-Smirnov (ver Sección B.1.5). Sin embargo, si H_0 es compuesta (los parámetros desconocidos se estiman por máxima verosimilitud y se trabaja con \hat{F}_0) los cuantiles de la distribución (asintótica) de D_n pueden ser demasiado conservativos y sería preferible utilizar la distribución exacta.

- 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

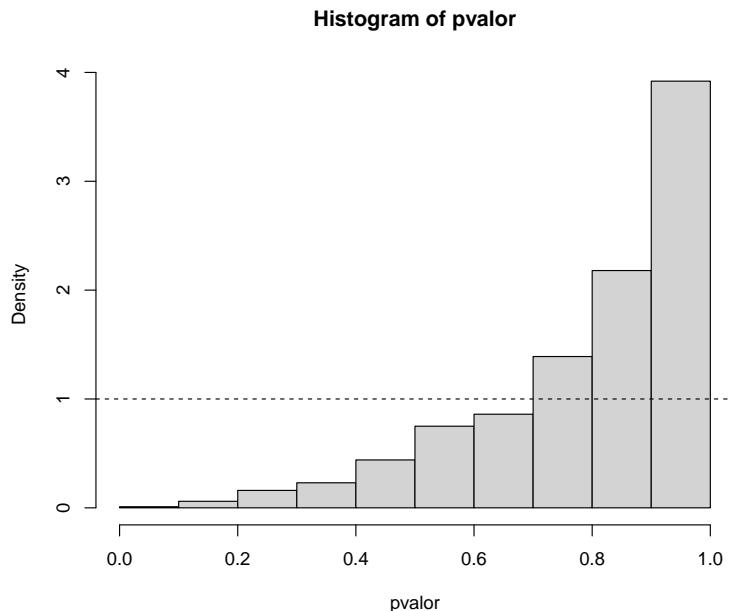
```

Análisis de los p-valores:

```

hist(pvalor, freq=FALSE)
abline(h=1, lty=2)  # curve(dunif(x, 0, 1), add=TRUE)

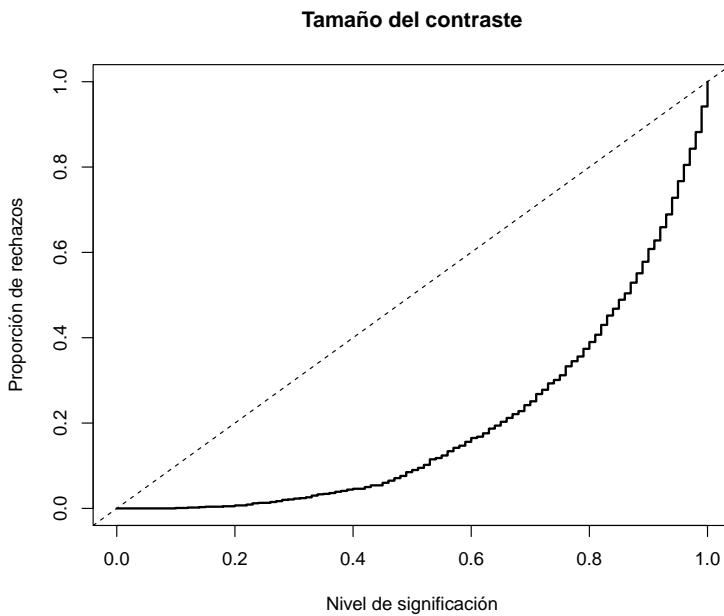
```



```

# 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
}
```

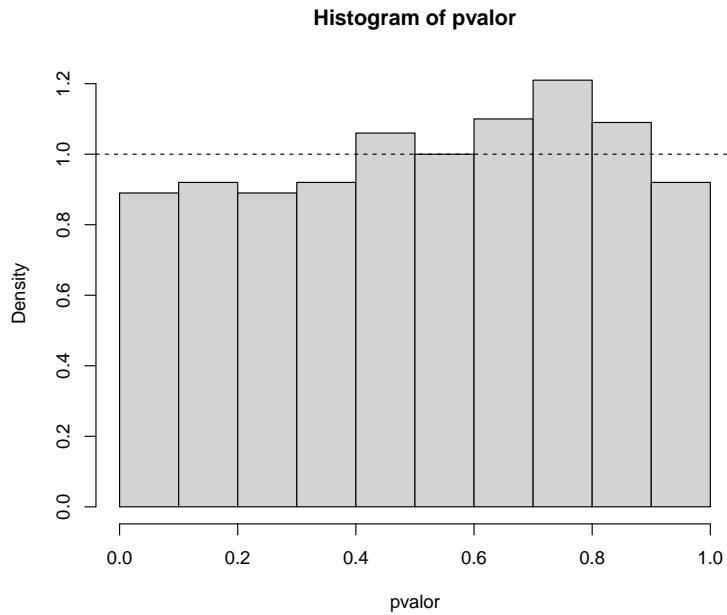
Proporción de rechazos:

```
{
  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")

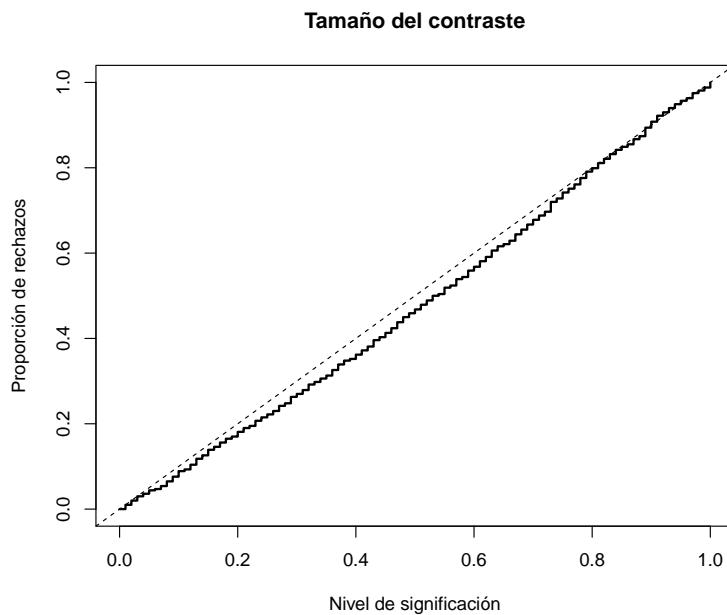
## 
## Proporción de rechazos al 1% = 0.01
## Proporción de rechazos al 5% = 0.044
## Proporción de rechazos al 10% = 0.089
```

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
}
```

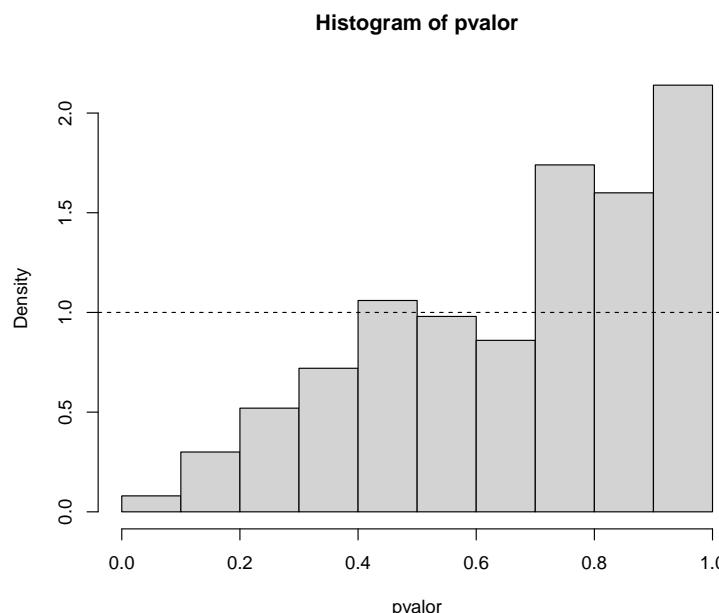
Proporción de rechazos:

```
{
  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")
}

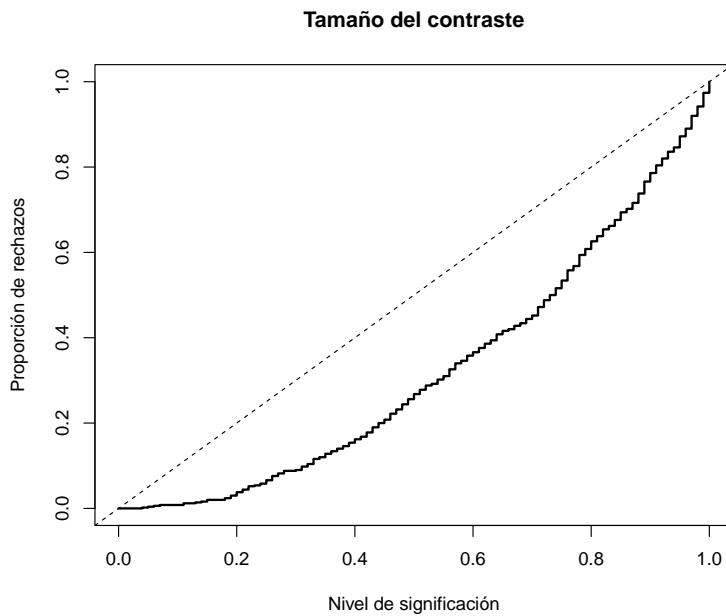
## Proporción de rechazos al 1% = 0
## Proporción de rechazos al 5% = 0.004
## Proporción de rechazos al 10% = 0.008
```

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
}
```

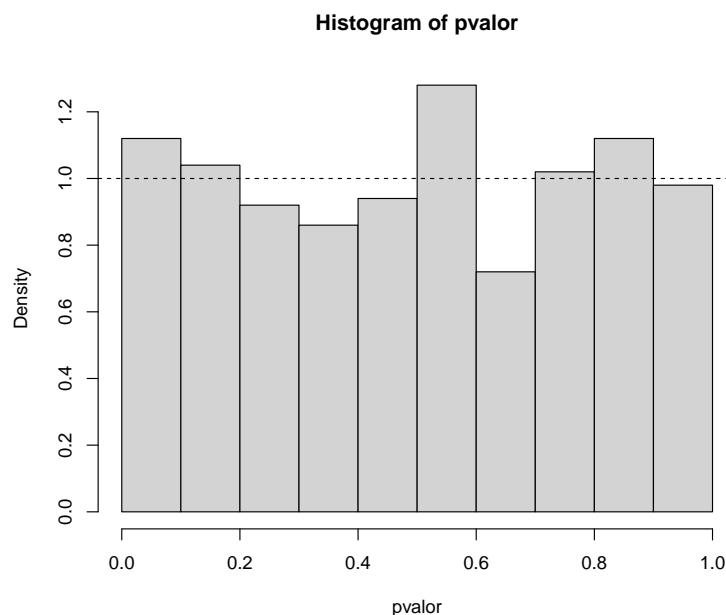
Proporción de rechazos:

```
{
  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")
}

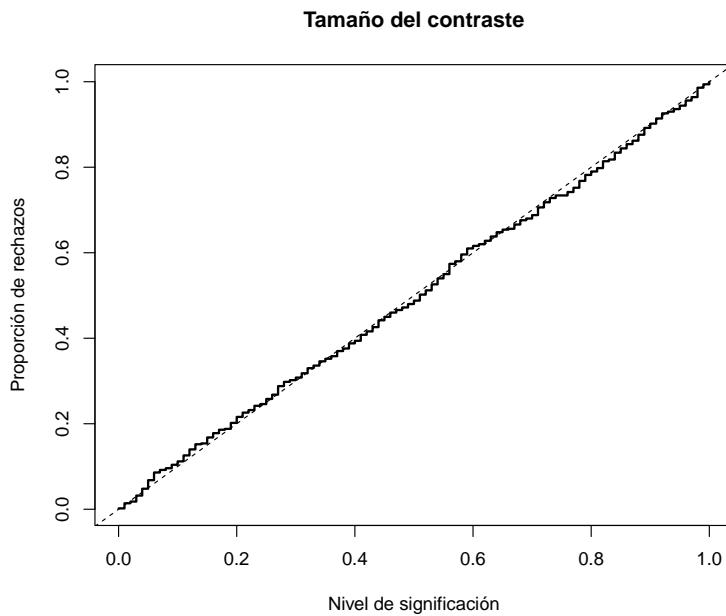
## Proporción de rechazos al 1% = 0.008
## Proporción de rechazos al 5% = 0.058
## Proporción de rechazos al 10% = 0.106
```

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, rate)` == `dweibull(x, 1, 1/rate)`. Estudiamos lo que ocurre al desplazar `dweibull(x, shape, 1/rate)` con $0 < \text{shape} < 2$.

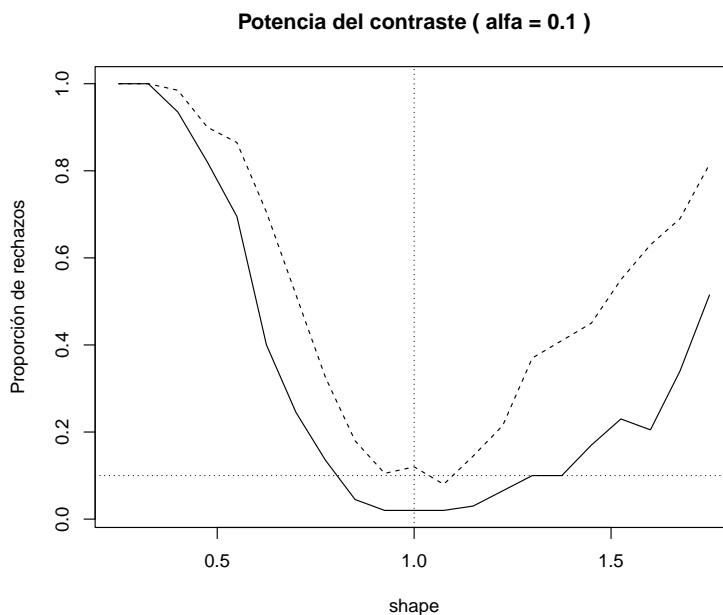
CUIDADO: las simulaciones pueden requerir de mucho tiempo de computación (consideramos valores pequeños de `nx` y `nsim` en datos y en `ks.exp.sim`).

```
set.seed(54321)
nx <- 20
rate <- 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/rate), 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 $D_n = \max(c(D_{\text{Minus}}, D_{\text{Plus}}))$ 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 D_n en la función `ks.exp.sim` por $D_n = \sum(\text{abs}((1:n - 0.5)/n - pexp(sort(x), rate=rate)))$.

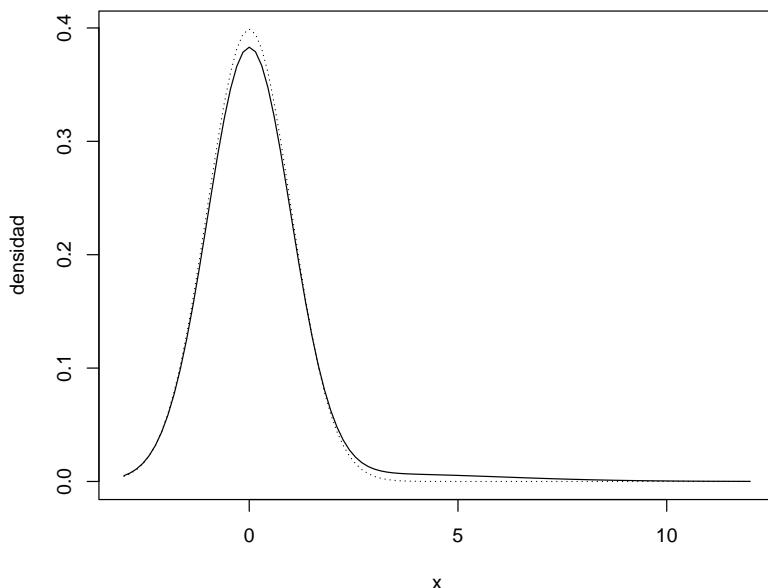
8.4 Comparación de estimadores

Ejercicio 8.6 (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. Consideraremos 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 5.4, 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 objetiva 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 5.4, 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).

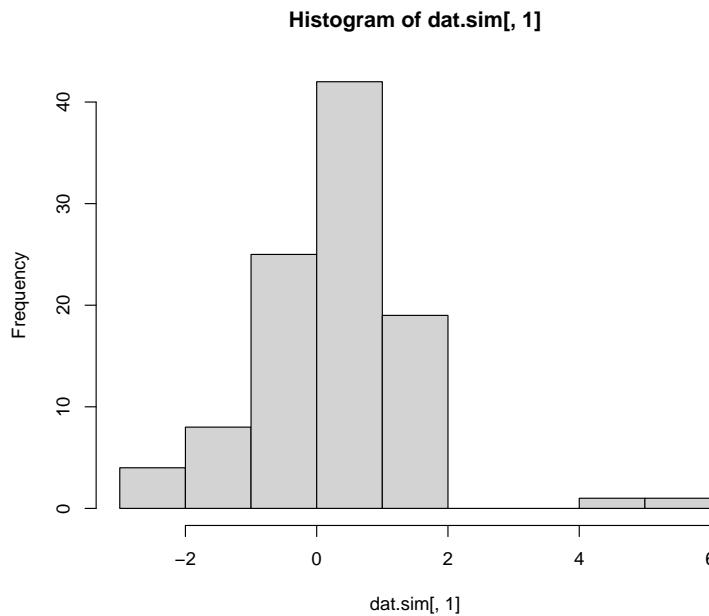
```
# media y mediana
xsd <- 1
xmed <- 0
ndat <- 100
nsim <- 500

# for (isim in 1:nsim) # evitar matrix y apply
set.seed(1)
ntsims <- ndat*nsim
p.sim <- rbinom(ntsims, 1, 0.05)
dat.sim <- rnorm(ntsims, 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)
```

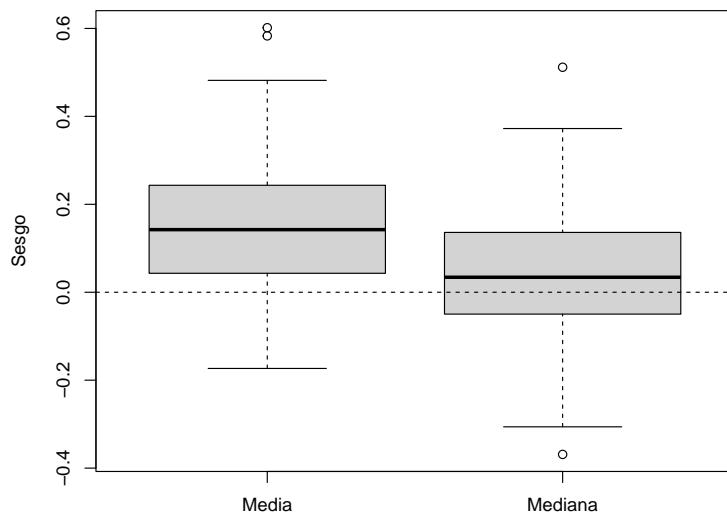
```
## [1] 0.1349537
mean(median.sim) # Coincide con el sesgo (media teórica es 0)
```

```
## [1] 0.04453509
sd(median.sim)
```

```
## [1] 0.1300611
```

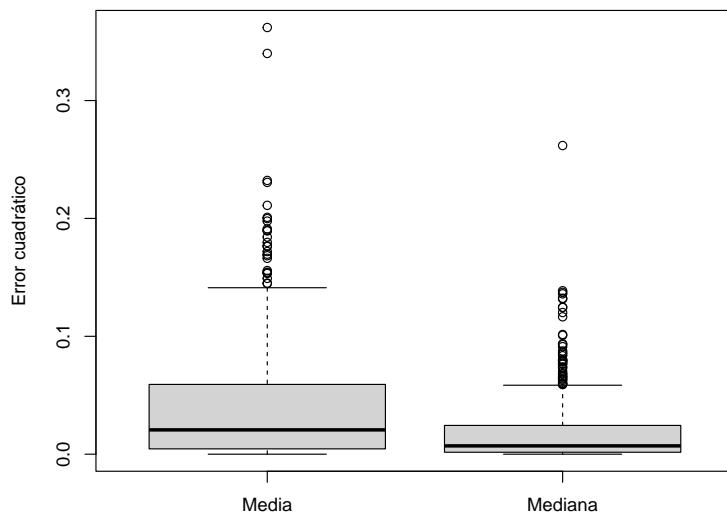
Sesgo:

```
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
```

8.5 Remuestreo Bootstrap

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.

En esta sección se incluye una breve introducción al bootstrap. Para información adicional ver por ejemplo Davison y Hinkley (1997) o Cao y Fernández-Casal (2020).

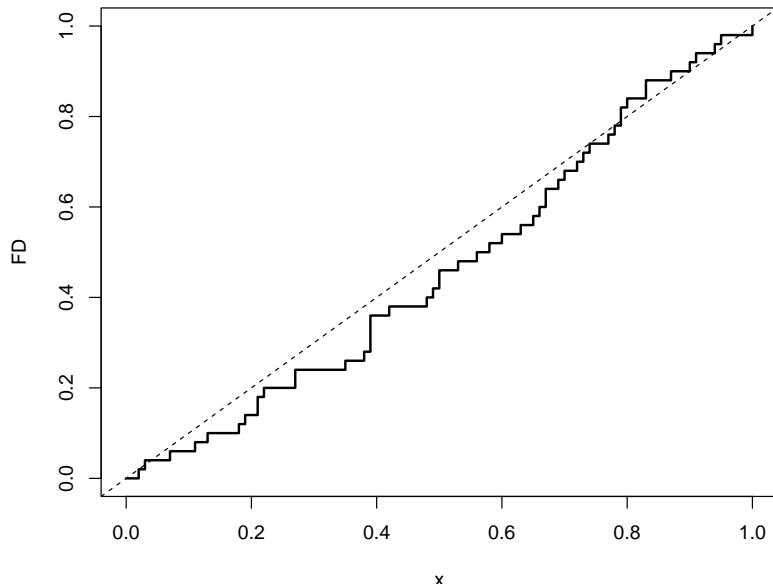
8.5.1 Idea:

Consideramos un conjunto de datos simulado:

```
set.seed(1)
data <- runif(50)
```

La idea es aproximar características poblacionales por las correspondientes de la distribución empírica de los datos observados:

```
# Distribución bootstrap
curve(ecdf(data)(x), ylab = "FD", type = "s", lwd = 2)
# Distribución teórica
abline(a = 0, b = 1, lty = 2)
```



Las características de la distribución empírica se pueden aproximar mediante simulación. 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):

```
# Muestra bootstrap
xboot <- sample(data, replace=TRUE)
```

8.5.2 Métodos de remuestreo Bootstrap

Método de remuestreo (Efron, 1979) utilizado para aproximar características de la distribución en el muestreo de un estadístico:

- Aproximación del sesgo y de la varianza.
- Construcción de intervalos de confianza
- Realizar contrastes de hipótesis.

De utilidad cuando no se dispone la distribución exacta, no es adecuado emplear la distribución asintótica, etc. La idea es aproximar características poblacionales por las correspondientes de la distribución empírica de los datos observados. 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).

Si $\mathbf{x} = (x_1, x_2, \dots, x_n)^t$ es una muestra i.i.d. F_θ y $\hat{\theta} = T(\mathbf{x})$ es un estimador de θ . Para $b = 1, \dots, B$:

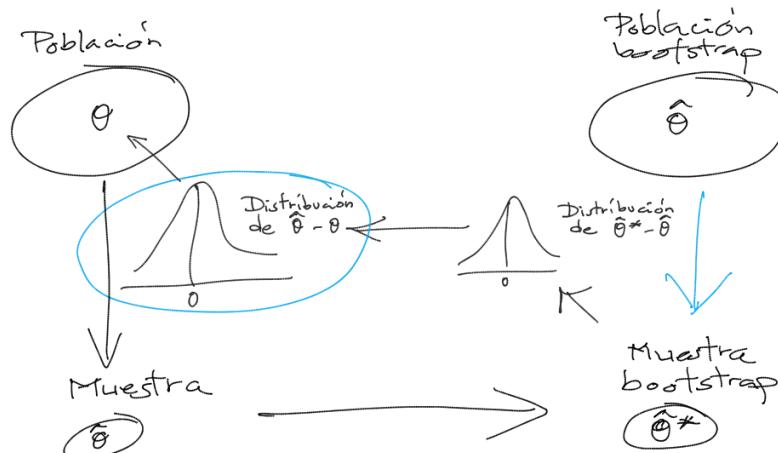
- $\mathbf{x}_b^* = (x_{b1}^*, x_{b2}^*, \dots, x_{bn}^*)^t$ muestra bootstrap obtenida mediante muestreo con reemplazamiento de $\{x_1, x_2, \dots, x_n\}$.
- $\hat{\theta}_b^* = T(\mathbf{x}_b^*)$ valor del estadístico en la muestra bootstrap.

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 $\hat{\theta}_b^* - \hat{\theta}$ aproxima la distribución de $\hat{\theta} - \theta$.

En general podríamos decir que:

- la muestra es a la población lo que la muestra bootstrap es a la muestra.



Ejemplo 8.1 (aproximación del sesgo y del error estándar mediante bootstrap)

Como ejemplo consideraremos una muestra de tamaño $n = 100$ de una normal estándar para tratar de aproximar el sesgo y error estándar de la media y la mediana mediante bootstrap.

```
# dat <- dat.sim[, 1]
set.seed(54321)
ndat <- 100
datmed <- 0
datsd <- 1
dat <- rnorm(ndat, mean=datmed, sd=datsd)
```

Consideramos 1000 réplicas bootstrap:

```
nboot <- 1000
```

Es habitual tomar `nboot + 1` entero múltiplo de 100 e.g. `nboot = 999 ó 1999` (facilita el cálculo de percentiles, orden `(nboot + 1)*p`)

El valor del estadístico en la muestra es:

```
stat.dat <- mean(dat)
```

Generamos las réplicas bootstrap:

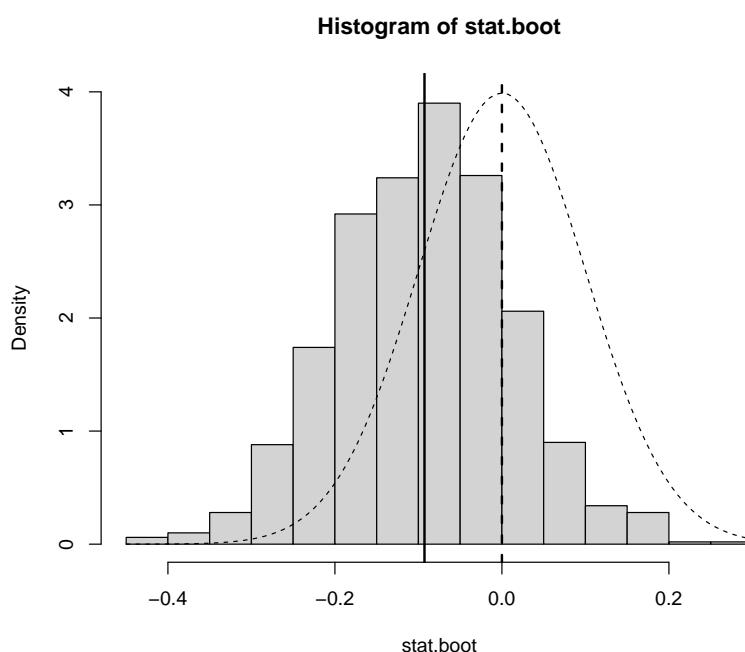
```
set.seed(1)
stat.boot <- numeric(nboot)
for (i in 1:nboot) {
  dat.boot <- sample(dat, replace=TRUE)
  stat.boot[i] <- mean(dat.boot)
}
# Valor esperado bootstrap del estadístico
mean.boot <- mean(stat.boot)
mean.boot
```

```
## [1] -0.09274131
```

Bootstrap percentil:

```
hist(stat.boot, freq=FALSE, ylim = c(0,4))
abline(v=mean.boot, lwd=2)
# abline(v=stat.dat)

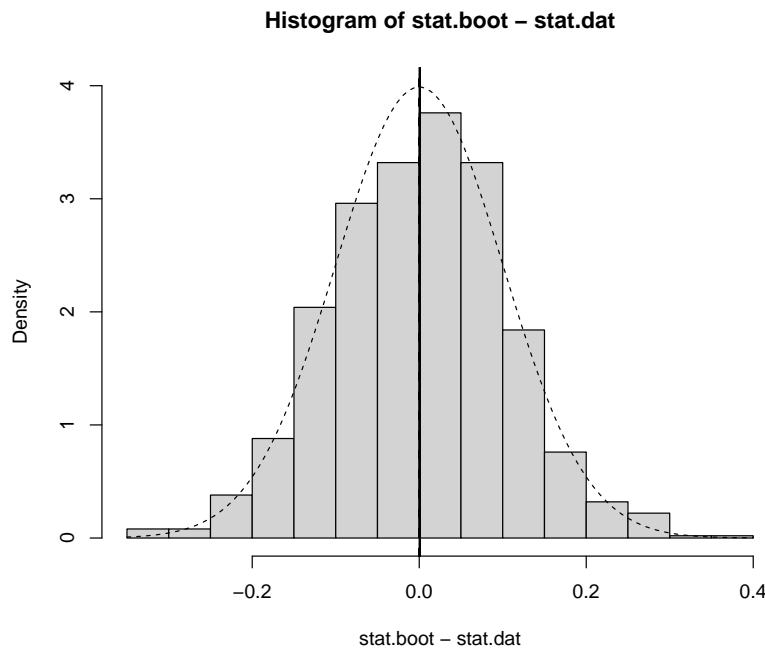
# Distribución poblacional
curve(dnorm(x, datmed, datsd/sqrt(ndat)), lty=2, add=TRUE)
abline(v=datmed, lwd=2, lty=2)
```



Bootstrap natural/básico:

```
hist(stat.boot-stat.dat, freq=FALSE, ylim = c(0,4))
abline(v=mean.boot-stat.dat, lwd=2)
```

```
# Distribución poblacional
# Distribución teórica de stat.dat - stat.teor
curve(dnorm(x, 0, datstd/sqrt(ndat)), lty=2, add=TRUE)
abline(v=0, lwd=2, lty=2)
```



Sesgo y error estándar bootstrap:

```
# sesgo (teor=0)
mean.boot = stat.dat

## [1] 0.0006390906

# error estándar
sd(stat.boot)

## [1] 0.1044501

# error estándar teórico
datstd/sqrt(ndat)

## [1] 0.1
```

Versión “optimizada” para R:

```
boot.strap <- function(dat, nboot=1000, statistic=mean)
{
  ndat <- length(dat)
  dat.boot <- sample(dat, ndat*nboot, replace=T)
  dat.boot <- matrix(dat.boot, ncol=nboot, nrow=ndat)
  stat.boot <- apply(dat.boot, 2, statistic)
}

fstatistic <- function(dat){
  # mean(dat)
  mean(dat, trim=0.2)
  # median(dat)
  # max(dat)
```

```

}

set.seed(1)
stat.dat <- fstatistic(dat)
stat.boot <- boot.bootstrap(dat, nboot, fstatistic)

res.boot <- c(stat.dat, mean(stat.boot)-stat.dat, sd(stat.boot))
names(res.boot) <- c("Estadístico", "Sesgo", "Err.Std")
res.boot

## Estadístico      Sesgo      Err.Std
## -0.144801929  0.005968729  0.119231743

```

8.5.3 Paquetes R: bootstrap, boot

Ver Sección 1.4 de Cao y Fernández-Casal (2019).

```

library(boot)
# ?boot

```

Función estadístico:

```

boot.f <- function(data, indices){
  # data[indices] será la muestra bootstrap
  mean(data[indices])
}

```

Generación de las muestras

```

set.seed(1)
stat.boot <- boot(dat, boot.f, nboot)
stat.boot

##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
##
## Call:
## boot(data = dat, statistic = boot.f, R = nboot)
##
##
## Bootstrap Statistics :
##      original      bias    std. error
## t1* -0.0933804  0.0006390906   0.1049474
## names(stat.boot)

## [1] "t0"        "t"         "R"         "data"      "seed"      "statistic"
## [7] "sim"       "call"      "stype"     "strata"    "weights"

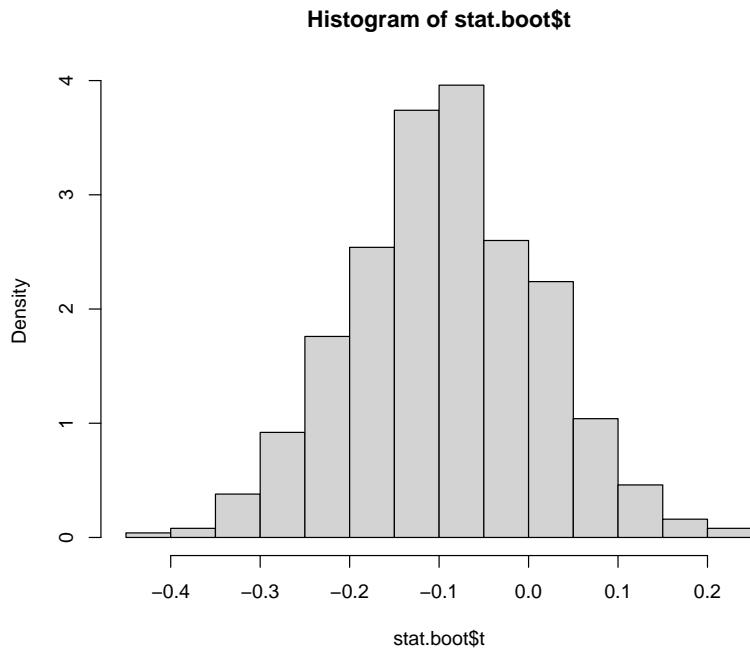
```

8.5.4 Gráficos

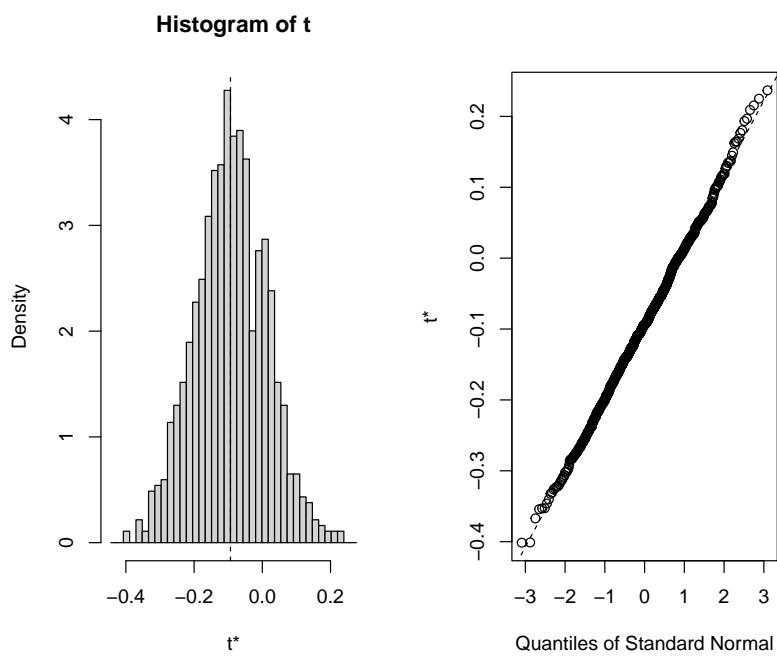
```

hist(stat.boot$t, freq=FALSE)

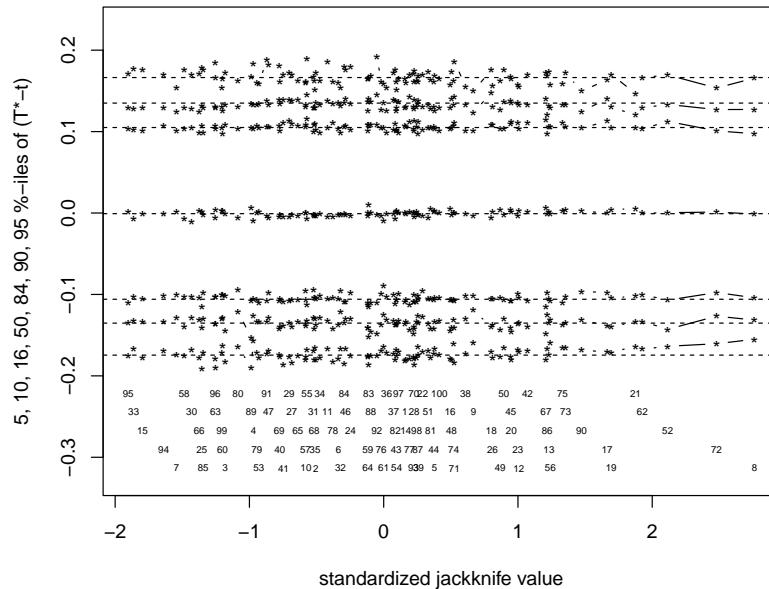
```



```
plot(stat.boot)
```



```
jack.after.boot(stat.boot)
```



8.5.5 Intervalos de confianza bootstrap

```
boot.ci(stat.boot, type=c("norm", "basic", "perc", "bca"))

## BOOTSTRAP CONFIDENCE INTERVAL CALCULATIONS
## Based on 1000 bootstrap replicates
##
## CALL :
## boot.ci(boot.out = stat.boot, type = c("norm", "basic", "perc",
## "bca"))
##
## Intervals :
## Level      Normal          Basic
## 95%   (-0.2997,  0.1117 )  (-0.3028,  0.1147 )
##
## Level      Percentile       BCa
## 95%   (-0.3015,  0.1161 )  (-0.2993,  0.1180 )
## Calculations and Intervals on Original Scale
```

Ejercicio 8.7

Repetir el Ejemplo 8.1 anterior considerando la media recortada al 10% (con parámetros adicionales).

```
boot.f <- function(data, indices, trim = 0.1){
  mean(data[indices], trim)
}

set.seed(1)
boot(dat, boot.f, nboot, trim = 0.2)

##
## ORDINARY NONPARAMETRIC BOOTSTRAP
##
```

```

## Call:
## boot(data = dat, statistic = boot.f, R = nboot, trim = 0.2)
##
## Bootstrap Statistics :
##      original     bias   std. error
## t1* -0.1448019 0.006095294 0.119021

```

Ejercicio 8.8 (propuesto)

En el tema 2 se propuso el análisis de la aleatoriedad de un generador de números pseudo-aleatorios mediante el test de rachas, que se podría implementar repetidamente. Sin embargo, la aproximación asintótica empleada por la rutina `runs.test` de la librería `tseries` no es adecuada para tamaños muéstrales pequeños ($n < 40$) y sería preferible utilizar la distribución exacta (o por lo menos utilizar una corrección por continuidad).

- Analizar el comportamiento del contraste empleando repetidamente el test de rachas, considerando 500 pruebas con muestras de tamaño 10 de una $Bernoulli(0.5)$. ¿Se observa algo extraño?
- Realiza un programa que permita aproximar por simulación la función de masa de probabilidad del estadístico número de rachas (a partir de valores de una $Bernoulli(0.5)$). Representarla gráficamente y compararla con la densidad normal. Obtener los puntos críticos para contrastar la hipótesis nula de aleatoriedad para $\alpha = 0.01, 0.05$ y 0.1 . ¿Es esta distribución adecuada para el contraste de aleatoriedad de variables continuas? ¿Cuál debería ser la probabilidad de obtener una única racha al aplicar el test a una variable continua?
- Diseñar una rutina que permita realizar el contraste de aleatoriedad de una variable continua aproximando el p -valor por simulación. Asumir que la distribución del estadístico puede ser asimétrica, en cuyo caso el p -valor $p = 2 \min \{P(R \leq \hat{R} | H_0), P(R \geq \hat{R} | H_0)\}$.
- Diseñar una rutina que permita realizar el contraste de aleatoriedad de una variable continua aproximando el p -valor mediante bootstrap.

Capítulo 9

Integración y Optimización Monte Carlo

Uno de los objetivos habituales de los estudios de simulación es la aproximación de una esperanza, es decir, se trataría de evaluar una integral, que en ocasiones puede ser compleja y de alta dimensión. Esto puede ser de interés en otros campos, aunque la integral no esté relacionada con procesos estocásticos. Adicionalmente, en muchos campos, incluido la Estadística, hay que resolver problemas de optimización. Para evitar problemas de mínimos locales se puede recurrir a herramientas que emplean búsquedas aleatorias de los valores óptimos.

9.1 Integración Monte Carlo (clásica)

La integración Monte Carlo se emplea principalmente para aproximar integrales multidimensionales:

$$I = \int \cdots \int h(x_1, \dots, x_d) dx_1 \cdots dx_d$$

donde puede presentar ventajas respecto a los métodos tradicionales de integración numérica (ver Apéndice C), ya que la velocidad de convergencia no depende del número de dimensiones.

Supongamos que nos interesa aproximar:

$$I = \int_0^1 h(x) dx$$

Si x_1, x_2, \dots, x_n i.i.d. $\mathcal{U}(0, 1)$ entonces:

$$I = E(h(\mathcal{U}(0, 1))) \approx \frac{1}{n} \sum_{i=1}^n h(x_i)$$

Si el intervalo de integración es genérico:

$$I = \int_a^b h(x) dx = (b-a) \int_a^b h(x) \frac{1}{(b-a)} dx = (b-a) E(h(\mathcal{U}(a, b))).$$

Si x_1, x_2, \dots, x_n i.i.d. $\mathcal{U}(a, b)$:

$$I \approx \frac{1}{n} \sum_{i=1}^n h(x_i) (b-a)$$

Ejercicio 9.1 (integración Monte Carlo clásica)

Crear una función que implemente la integración Monte Carlo clásica para aproximar integrales del tipo:

$$I = \int_a^b h(x)dx.$$

Emplearla para aproximar:

$$\int_0^1 4x^4 dx = \frac{4}{5},$$

y representar gráficamente la aproximación en función de n .

Como primera aproximación podríamos considerar:

```
mc.integral0 <- function(fun, a, b, n) {
  # Integración Monte Carlo de fun entre a y b utilizando una muestra de tamaño n
  # fun es una función de una sola variable (y que no es vectorial)
  # Se asume 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))
}
```

Función a integrar:

```
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)
```

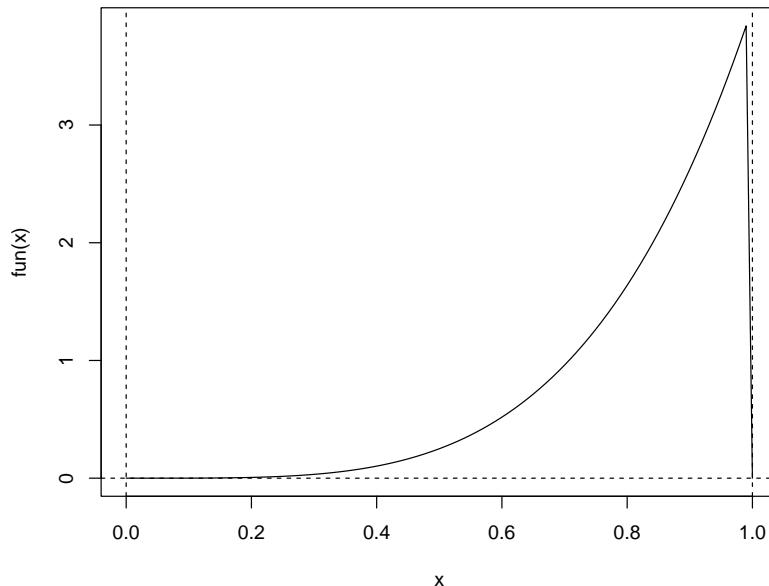


Figura 9.1: Ejemplo de integral en dominio acotado.

```
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:

```
mc.integral <- function(fun, a, b, n, plot = TRUE) {
  fx <- sapply(runif(n, a, b), fun) * (b - a)
  if (plot) {
    cumn <- 1:n
    estint <- cumsum(fx)/cumn
    esterr <- sqrt((cumsum(fx^2)/cumn - estint^2)/(cumn-1)) # Errores estándar
    plot(estint, ylab = "Media y rango de error", type = "l", lwd = 2, ylim = mean(fx) +
      c(-1, 1) * max(esterr[-1]), xlab = "Iteraciones")
    lines(estint + 2 * esterr, col = "darkgray", lty = 3)
    lines(estint - 2 * esterr, col = "darkgray", lty = 3)
    valor <- estint[n]
    abline(h = valor, lty = 2)
    return(list(valor = valor, error = 2 * esterr[n]))
  } else return(list(valor = mean(fx), error = 2 * sd(fx)/sqrt(n)))
}

set.seed(1)
mc.integral(fun, 0, 1, 5000)

## $valor
## [1] 0.8142206
##
## $error
## [1] 0.0309005
abline(h = 4/5)
```

Si sólo interesa la aproximación:

```
set.seed(1)
mc.integral(fun, 0, 1, 5000, plot = FALSE)

## $valor
## [1] 0.8142206
##
## $error
## [1] 0.0309005
```

Nota: Es importante tener en cuenta que la función `mc.integral` solo es válida para dominio finito.

9.1.1 Caso general

A partir a ahora consideraremos que interesa aproximar una integral de la forma:

$$\theta = E(h(X)) = \int h(x) f(x) dx$$

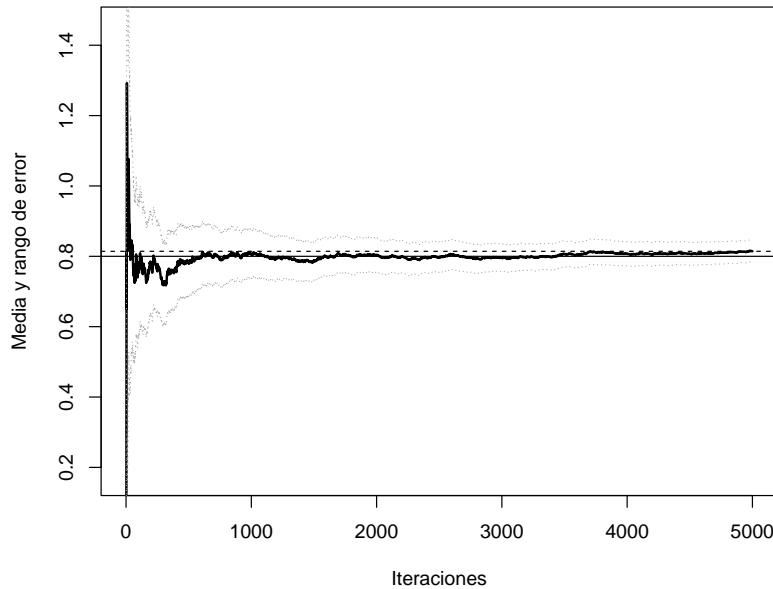


Figura 9.2: Convergencia de la aproximación de la integral mediante simulación.

siendo $X \sim f$, entonces, si x_1, x_2, \dots, 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 9.1).

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
```

Ejercicio 9.2

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 solvete este problema.

9.2 Muestreo por importancia

Para aproximar la integral:

$$\theta = E(h(X)) = \int h(x) f(x) dx,$$

puede ser preferible generar observaciones de una densidad g que tenga una forma similar al producto hf .

Si $Y \sim g$:

$$\theta = \int h(x) f(x) dx = \int \frac{h(x) f(x)}{g(x)} g(x) dx = E(q(Y)).$$

siendo $q(x) = \frac{h(x)f(x)}{g(x)}$.

Si y_1, y_2, \dots, y_n i.i.d. $Y \sim g$:

$$\theta \approx \frac{1}{n} \sum_{i=1}^n q(y_i) = \frac{1}{n} \sum_{i=1}^n w(y_i) h(y_i) = \hat{\theta}_g$$

con $w(y) = \frac{f(y)}{g(y)}$.

En este caso $Var(\hat{\theta}_g) = Var(q(Y))/n$, 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 aplicar el TCL, la varianza del estimador $\hat{\theta}_g$ es finita si:

$$E(q^2(Y)) = \int \frac{h^2(x) f^2(x)}{g(x)} dx = E\left(h^2(X) \frac{f(X)}{g(X)}\right) < \infty.$$

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 $E(h^2(X))$; ver Sección 4.1).

La distribución de los pesos $w(y_i)$ debería ser homogénea para evitar datos influyentes (inestabilidad).

Ejercicio 9.3

Aproximar la integral del Ejercicio 9.2 anterior empleando muestreo por importancia considerando como densidad auxiliar una exponencial de parámetro $\lambda = 1$ truncada en t :

$$g(x) = \lambda e^{-\lambda(x-t)}, x > t,$$

(emplear `rexp(n)+t` y `dexp(y-t)`). Comparar $h(x)f(x)$ con $g(x)f(4.5)$ y representar gráficamente la aproximación en función de n .

```
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)
```

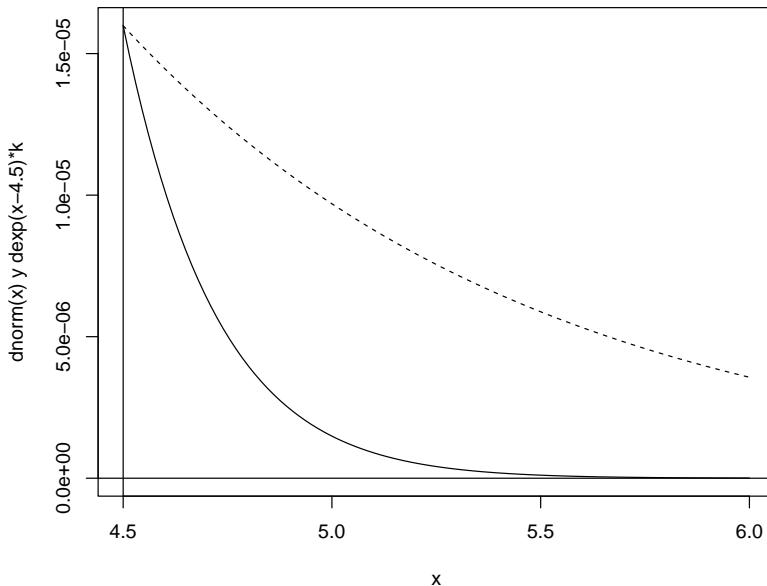


Figura 9.3: Objetivo a integrar (densidad objetivo truncada) y densidad auxiliar reescalada.

Se generan los valores de la densidad auxiliar y se calculan los pesos:

```
set.seed(1)
nsim <- 10^3
y <- rexp(nsim) + 4.5 # Y ~ g
w <- dnorm(y)/dexp(y - 4.5)
```

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
plot(cumsum(w)/1:nsim, type = "l", ylab = "Aproximación", xlab = "Iteraciones")
abline(h = pnorm(-4.5), lty = 2)
```

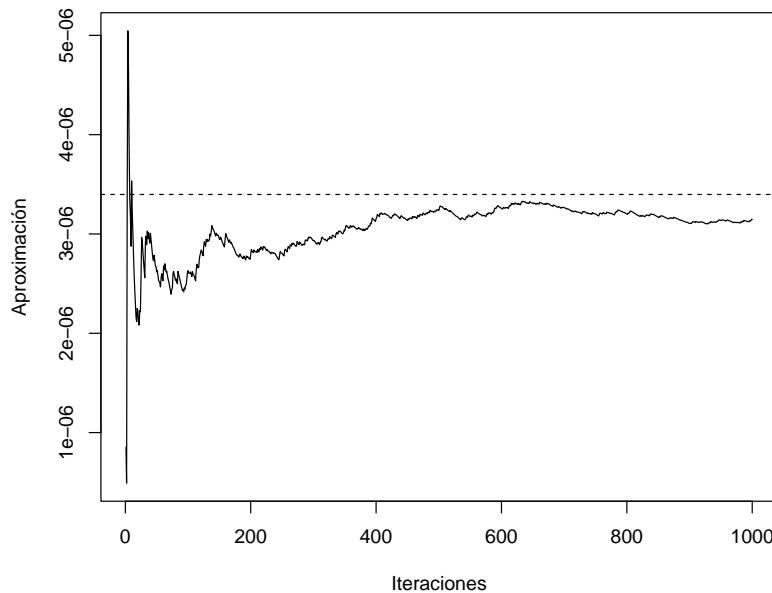


Figura 9.4: Convergencia de la aproximación de la integral mediante muestreo por importancia.

El error estandar de la aproximación sería `sqrt(var(w * h(y))/nsim)`:

```
sqrt(var(w)/nsim) # sd(w*h(y))/sqrt(nsim)
```

```
## [1] 1.371154e-07
```

Empleando la aproximación tradicional:

```
est <- mean(rnorm(nsim) > 4.5)
est
```

```
## [1] 0
sqrt(est * (1 - est)/nsim)
```

```
## [1] 0
```

Ejemplo 9.1 (muestreo por importancia con mala densidad auxiliar)

Supongamos que se pretende aproximar $P(2 < X < 6)$ siendo $X \sim \text{Cauchy}(0, 1)$ empleando muestreo por importancia y considerando como densidad auxiliar la normal estandar $Y \sim N(0, 1)$. Representaremos gráficamente la aproximación y estudiaremos los pesos $w(y_i)$.

Nota: En este caso van a aparecer problemas (la densidad auxiliar debería tener colas más pesadas que la densidad objetivo; sería adecuado si intercambiaramos las distribuciones objetivo y auxiliar, como en el ejercicio 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)
```

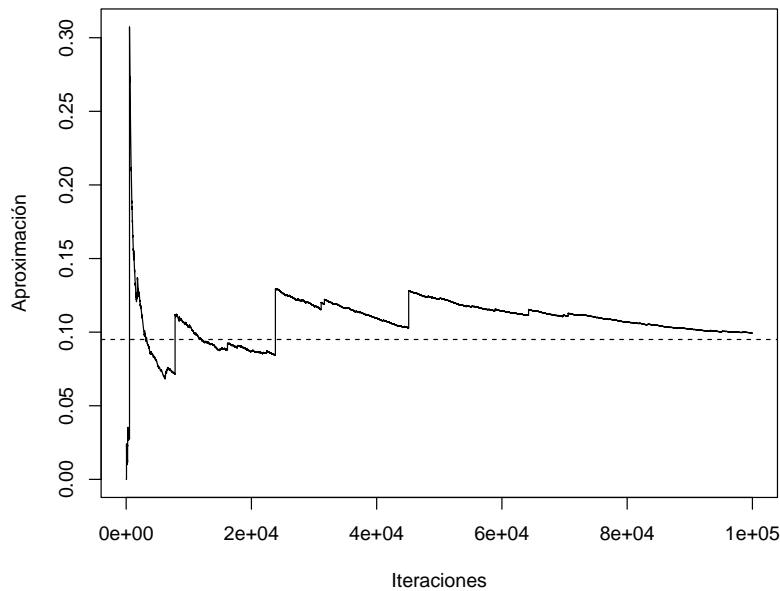


Figura 9.5: Gráfico de convergencia de la aproximación mediante muestreo por importancia con mala densidad auxiliar.

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:

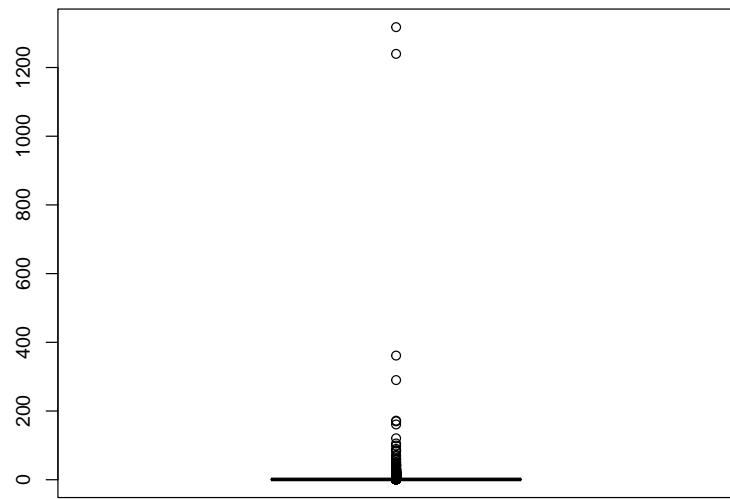


Figura 9.6: Gráfico de cajas de los pesos del muestreo por importancia reescalados (de forma que su media es 1).

```
boxplot(nsim * w/sum(w))
```

9.2.1 Remuestreo (del muestreo) por importancia

Cuando f y/o g son quasi-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. = $w(y_i) / \sum_{i=1}^n w(y_i)$) se obtiene una simulación aproximada de f (*Sample importance resampling*, Rubin, 1987).

Ejercicio 9.4

Generar 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).

Se trata de simular una normal a partir de una Cauchy (Sampling Importance Resampling). NOTA: En este caso $f(y) = dnorm(y)$ y $g(y) = dcauchy(y)$, al revés del ejercicio 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)

```

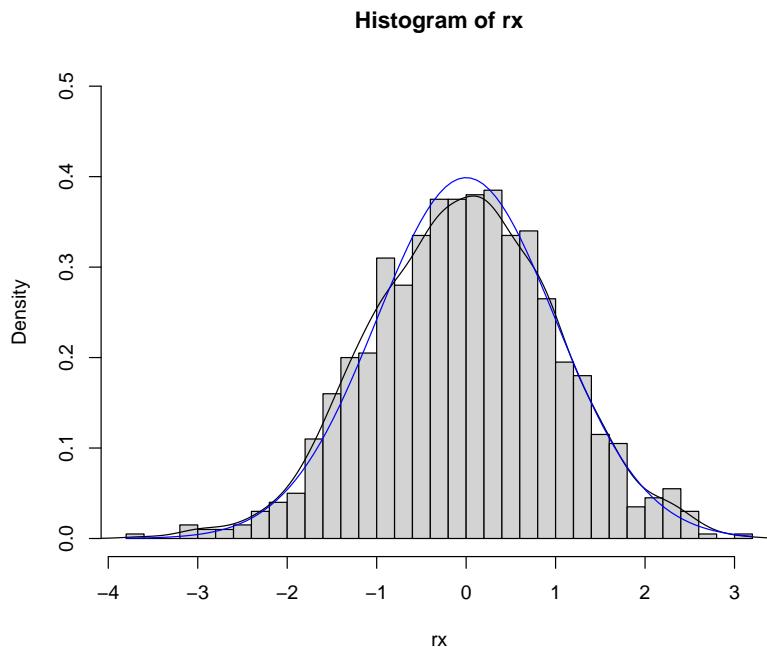


Figura 9.7: Distribución de los valores generados mediante remuestreo por importancia y densidad objetivo.

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 $\text{sum}(w * h(y))/\text{sum}(w)$ (media ponderada) y en el análisis de convergencia se emplearía $\text{cumsum}(w * h(y))/\text{cumsum}(w)$.

Ejercicio 9.5 (propuesto)

Consideramos una variable aleatoria con densidad:

$$f(x) \propto e^{-x} \cos^2(x), \text{ si } x > 0.$$

- 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.

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 comprobar 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 $c = \frac{3}{5}$).

9.3 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 inicial¹. Un ejemplo donde es habitual que aparezcan este tipo de problemas es en la estimación por máxima verosimilitud (la función objetivo puede ser multimodal).

Ejercicio 9.6 (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. Generar una muestra de 200 valores de esta distribución con $\mu_1 = 0$ y $\mu_2 = 2.5$, construir la correspondiente función de verosimilitud y representarla gráficamente. Obtener la estimación por máxima verosimilitud de los parámetros empleando la rutina `nlm`.

Obtención de la muestra (simulación mixtura dos normales):

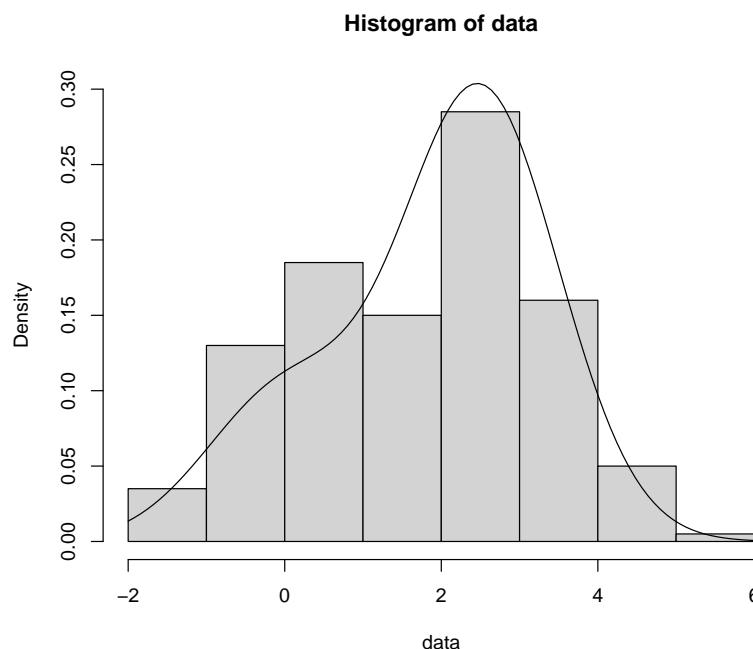
```
nsim <- 200
mu1 <- 0
mu2 <- 2.5
sd1 <- sd2 <- 1

set.seed(12345)
p.sim <- rbinom(nsim, 1, 0.25)
```

¹Este tipo de algoritmos se denominan *codiciosos* o *voraces*, porque buscan la mejor opción a “corto plazo”.

```
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)
```



Logaritmo (negativo) de la función de verosimilitud (para la estimación de las medias)

```
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 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`:

```

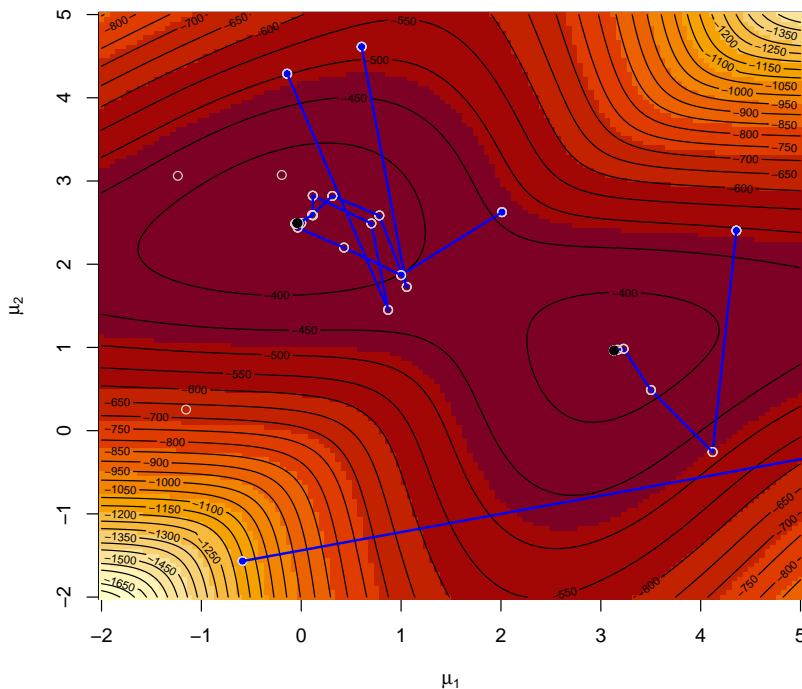
mmu1 <- mmu2 <- seq(-2, 5, length = 128)
lli <- outer(mmu1, mmu2, function(x,y) apply(cbind(x,y), 1, like))

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
```

9.3.1 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 (transformación Boltzman-Gibbs):

$$g(x) \propto \exp(-f(x)/T),$$

donde $T > 0$ es un parámetro (denominado temperatura) seleccionado de forma que se garantize la integrabilidad.

Entre los métodos de optimización Monte Carlo podríamos destacar:

- Métodos con gradiente aleatorio.
- Temple simulado.
- Algoritmos genéticos.
- Montecarlo EM.
- ...

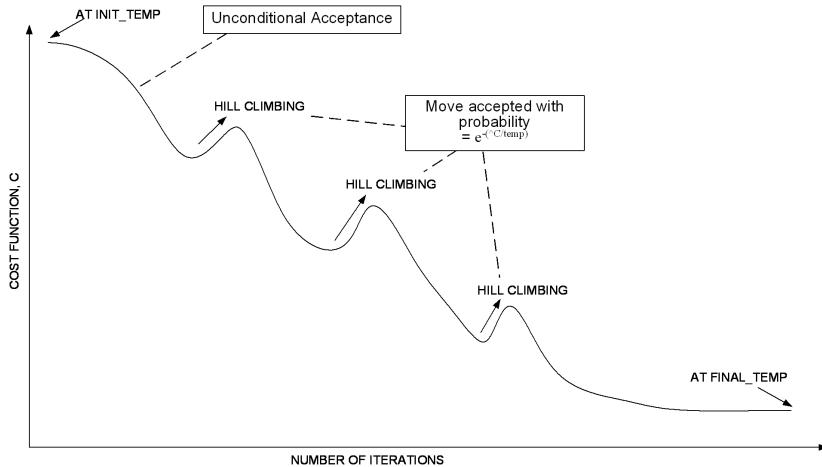
9.4 Temple simulado

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.



9.4.1 Algoritmo:

```

temp <- TEMP.INIT
place <- INIT.PLACEMENT()
cost.place <- COST(place)
while(temp < TEMP.FINAL) {
  while(LOOP.CRITERION()) {
    place.new <- PERTURB(place, temp)
    cost.new <- COST(place.new)
    cost.inc <- cost.new - cost.place
    temp <- SCHEDULE(temp)
    if ((cost.inc < 0) || (runif(1) > exp(-(cost.inc/temp)))) break
  }
  place <- place.new
  cost.place <- cost.new
  # temp <- SCHEDULE(temp)
}
COST <- function(place, ...) {...}
SCHEDULE <- function(temp, ...) {...}
INIT.PLACEMENT <- function(...) {...}
LOOP.CRITERION <- function(...) {...}

```

Adaptado de Premchand Akella (ppt).

Este algoritmo se puede ver como una adaptación del método de Metropolis-Hastings que veremos más adelante (Tema 11 Introducción a los métodos de cadenas de Markov Monte Carlo).

Ejercicio 9.7 (Estimación máximo-verosímil empleando temple simulado)

Repetir el Ejercicio 9.6 anterior empleando el algoritmo del temple simulado.

Minimización “SANN” con optim:

```

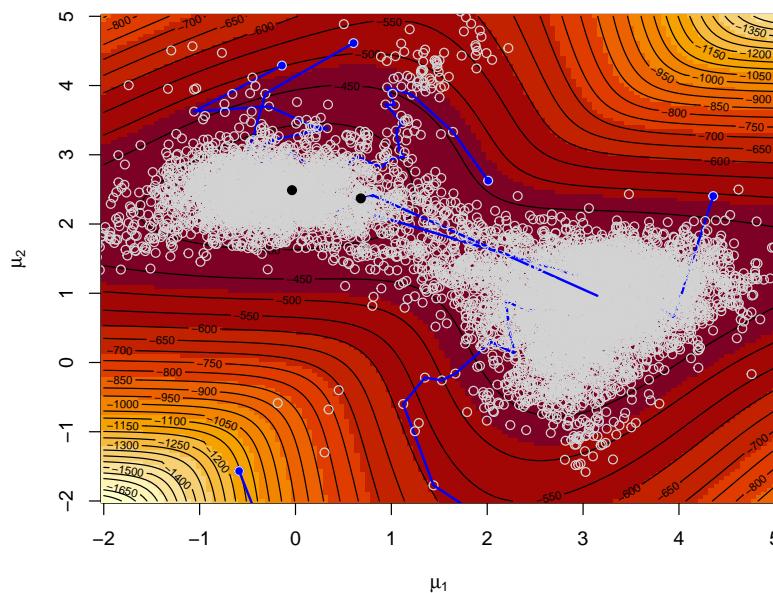
# Representar la superficie del logaritmo de la verosimilitud
image(mmu1, mmu2, -lli, xlab = expression(mu[1]), ylab = expression(mu[2]))
contour(mmu1, mmu2, -lli, nlevels = 50, add = TRUE)
points(starts, col = "blue", pch = 19)

```

```

set.seed(1)
for (j in 1:nstarts){
  # Normalmente llamariamos 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 = 200))
  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

```

Alternativa: función basada en el algoritmo empleado en el ejemplo 5.9 del libro: Robert y Casella, Introducing Monte Carlo Methods with R, Springer, 2010.

```

SA <- function(fun, pini, lower = -Inf, upper = Inf, tolerance = 10^(-4), factor = 1) {
  temp <- scale <- iter <- dif <- 1
  npar <- length(pini)
  the <- matrix(pini, ncol = npar)
  curfun <- hval <- fun(pini)
  while (dif > tolerance) {
    prop <- the[iter, ] + rnorm(npar) * scale[iter]
    # Se decide si se acepta la propuesta
    if (any(prop < lower) || any(prop > upper) ||
        (temp[iter] * log(runif(1)) > -fun(prop) + curfun)) prop <- the[iter, ]
    curfun <- fun(prop)
    hval <- c(hval, curfun)
    the <- rbind(the, prop)
    iter <- iter + 1
  }
}
```

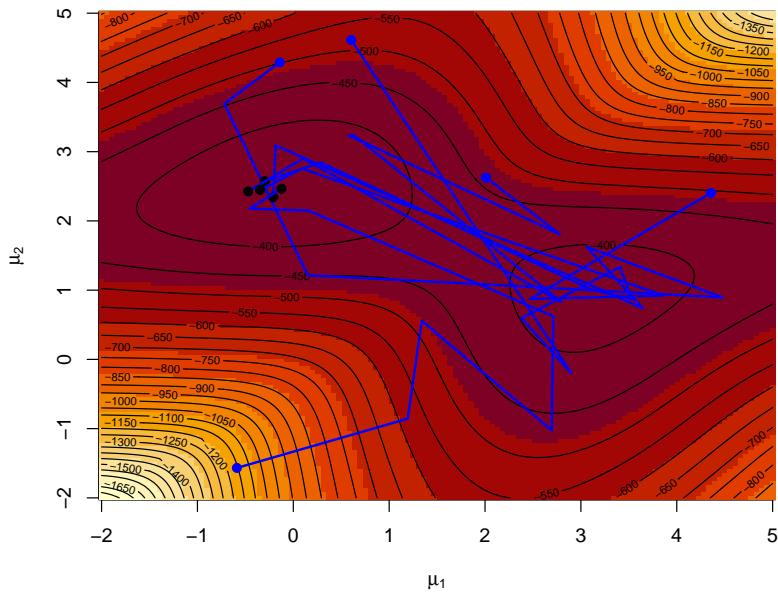
```

temp <- c(temp, 1/log(iter + 1)) # Actualizar la temperatura
# Se controla el nº de perturbaciones aceptadas
ace <- length(unique(the[(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 perturbación
dif <- (iter < 100) + (ace < 2) + (max(hval) - max(hval[1:(iter/2)]))
}
list(theta = the, val = hval, ite = 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, ])
  lines(sar$the[, 1], sar$the[, 2], lwd = 2, col = "blue")
  points(sar$the[sar$it, 1], sar$the[sar$it, 2], pch = 19)
}

```



9.5 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.

Repetir el ejercicio anterior empleando la función `DEoptim`.

Ejercicio 9.8 (Estimación máximo-verosímil empleando un algoritmo genético)

Optimización con algoritmo genético implementado en `DEoptim`:

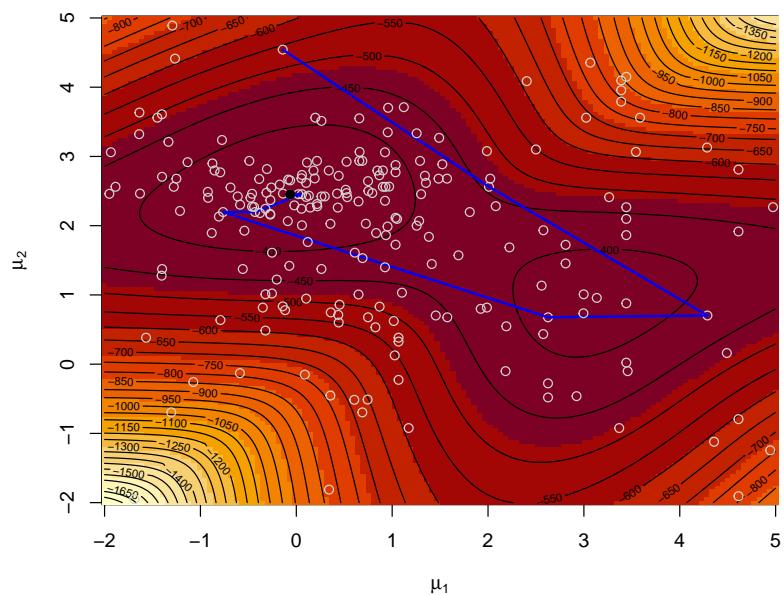
```
require(DEoptim)

# 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)
```



Capítulo 10

Técnicas de reducción de la varianza

Si el coste computacional de generar un número suficiente de simulaciones es demasiado grande (por ejemplo, cuando evaluar el estadístico de interés requiere mucho tiempo de computación), nos puede interesar emplear técnicas de reducción de la varianza que nos permitan obtener una buena aproximación con un número menor de generaciones. Éstas técnicas son aplicadas normalmente cuando se pretende ofrecer respuestas lo más precisas posibles y principalmente sobre cantidades medias.

En este capítulo asumiremos que **estamos interesados en aproximar la media** de un estadístico mediante simulación y **no nos interesa aproximar su varianza**. Existe un sinfín de técnicas encaminadas a reducir la varianza en un estudio de este tipo (respecto a una aproximación estándar). Algunas de ellas son:

- Muestreo por importancia (Sección 9.2).
- Variables antitéticas.
- Muestreo estratificado.
- Variables de control.
- Números aleatorios comunes.
- Condicionamiento.

No obstante, en general, si uno de los objetivos de la simulación es precisamente estimar la variabilidad no convendría emplear estas técnicas (para ello hay otros métodos disponibles, como el Jackniffe o el Bootstrap; ver p.e. Cao y Fernández-Casal, 2021, Capítulo 2).

10.1 Variables antitéticas

Supongamos que pretendemos aproximar

$$\theta = E(Z)$$

con $Var(Z) = \sigma^2$. Si generamos n pares $(X_1, Y_1), \dots, (X_n, Y_n)$ de $X \sim Y \sim Z$ con $Cov(X, Y) < 0$, el estimador combinado tiene menor varianza:

$$\begin{aligned} Var\left(\frac{\bar{X} + \bar{Y}}{2}\right) &= \frac{1}{4} (Var(\bar{X}) + Var(\bar{Y}) + 2Cov(\bar{X}, \bar{Y})) \\ &= \frac{\sigma^2}{2n} + \frac{1}{2n} Cov(X, Y) \\ &= \frac{\sigma^2}{2n} (1 + \rho(X, Y)), \end{aligned}$$

que es equivalente a una muestra unidimensional independiente con el mismo número de observaciones $2n$ (con una reducción del $-100\rho(X, Y)\%$).

10.1.1 Ejemplo: Integración Monte Carlo

Para aproximar:

$$I = \int_0^1 h(x) dx,$$

a partir de x_1, x_2, \dots, x_n i.i.d. $\mathcal{U}(0, 1)$. Podemos emplear:

$$\begin{aligned} I &= E\left(\frac{h(U) + h(1-U)}{2}\right) \\ &\approx \frac{1}{2n} \sum_{i=1}^n (h(x_i) + h(1-x_i)). \end{aligned}$$

10.1.2 Generación de variables antitéticas

Cuando se utiliza el método de inversión resulta sencillo obtener pares de variables con correlación negativa:

- $U \sim \mathcal{U}(0, 1)$ para simular X .
- $1 - U$ para simular la variable antitética Y .

En el caso general, si $X = h(U_1, \dots, U_d)$ y h es monótona puede verse (e.g. Ross, 1997) que $Y = h(1 - U_1, \dots, 1 - U_d)$ está negativamente correlada con X .

Si $X \sim \mathcal{N}(\mu, \sigma)$ puede tomarse como variable antitética

$$Y = 2\mu - X$$

En general esto es válido para cualquier variable simétrica respecto a un parámetro μ . (e.g. $X \sim \mathcal{U}(a, b)$ e $Y = a + b - X$).

Ejercicio 10.1

Variables antitéticas en implementación Monte Carlo Generar variables antitéticas para aproximar integrales del tipo:

$$I = \int_a^b h(x) dx.$$

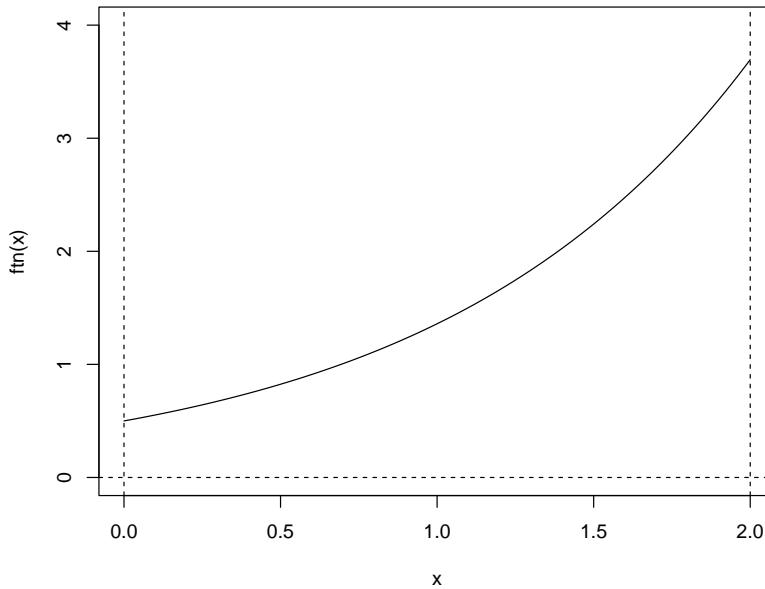
Emplearla para aproximar:

$$E(e^{\mathcal{U}(0,2)}) = \int_0^2 \frac{1}{2} e^x dx \approx 3.194,$$

y representar gráficamente la aproximación en función de n .

Representamos la función objetivo:

```
a <- 0; b <- 2
ftn <- function(x) return(exp(x)/(b-a))
curve(ftn, a, b, ylim=c(0,4))
abline(h=0, lty=2)
abline(v=c(a,b), lty=2)
```



Se trata de calcular la media de $e^{\mathcal{U}(0,2)}$:

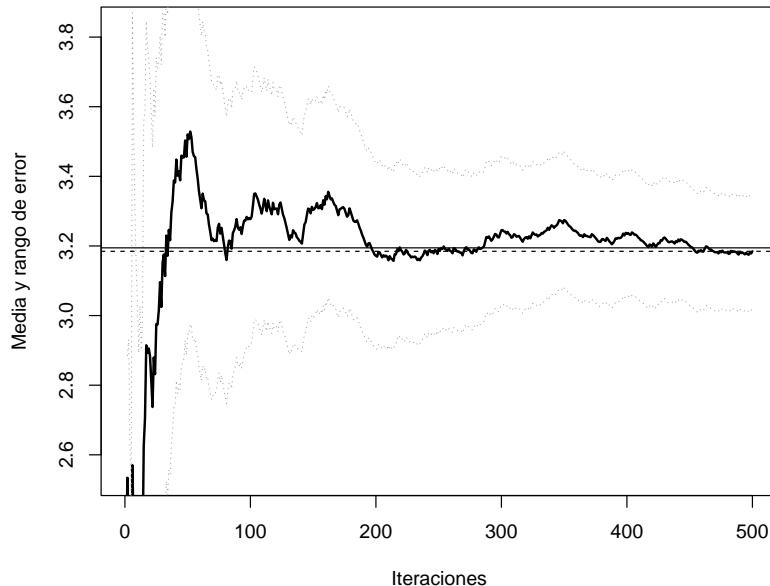
```
teor <- (exp(b)-exp(a))/(b-a)
teor
```

```
## [1] 3.194528
```

Para la aproximación por integración Monte Carlo podemos emplear la función del capítulo anterior:

```
mc.integral <- function(fun, a, b, n, plot = TRUE) {
  fx <- sapply(runif(n, a, b), fun) * (b - a)
  if (plot) {
    cumn <- 1:n
    estint <- cumsum(fx)/cumn
    esterr <- sqrt((cumsum(fx^2)/cumn - estint^2)/(cumn-1)) # Errores estándar
    plot(estint, ylab = "Media y rango de error", type = "l", lwd = 2, ylim = mean(fx) +
         c(-1, 1) * max(esterr[-1])), xlab = "Iteraciones")
    lines(estint + 2 * esterr, col = "darkgray", lty = 3)
    lines(estint - 2 * esterr, col = "darkgray", lty = 3)
    valor <- estint[n]
    abline(h = valor, lty = 2)
    return(list(valor = valor, error = 2 * esterr[n]))
  } else return(list(valor = mean(fx), error = 2 * sd(fx)/sqrt(n)))
}

set.seed(54321)
res <- mc.integral(ftn, a, b, 500)
abline(h = teor)
```



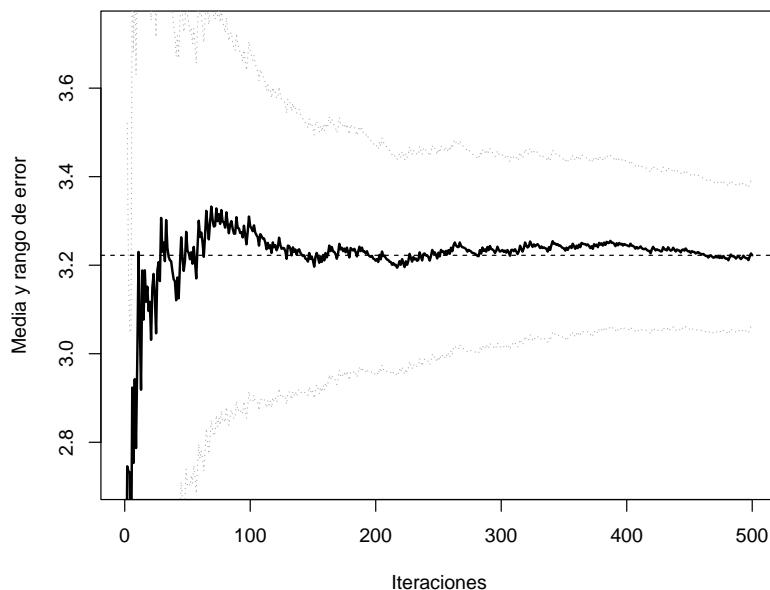
res

```
## $valor
## [1] 3.184612
##
## $error
## [1] 0.1629942
```

Para la integración Monte Carlo con variables antitéticas podríamos considerar:

```
mc.integrala <- function(ftn, a, b, n, plot = TRUE) {
  # n es el nº de evaluaciones de la función objetivo (para facilitar comparaciones, solo se genera
  x <- runif(n%/%2, a, b)
  # La siguiente línea solo para representar alternando
  x <- as.numeric(matrix(c(x,a+b-x),nrow=2,byrow=TRUE))
  # bastaría con emplear p.e. c(x,a+b-x)
  fx <- sapply(x, ftn)*(b-a)
  if (plot) {
    cumn <- 1:n
    estint <- cumsum(fx)/cumn
    esterr <- sqrt((cumsum(fx^2)/cumn - estint^2)/(cumn-1)) # Errores estándar
    plot(estint, ylab = "Media y rango de error", type = "l", lwd = 2, ylim = mean(fx) +
      c(-1, 1) * max(esterr[-1]), xlab = "Iteraciones")
    lines(estint + 2 * esterr, col = "darkgray", lty = 3)
    lines(estint - 2 * esterr, col = "darkgray", lty = 3)
    valor <- estint[n]
    abline(h = valor, lty = 2)
    return(list(valor=estint[n],error=2*esterr[n]))
  } else return(list(valor=mean(fx),error=2*sd(fx)/sqrt(n)))
}

set.seed(54321)
res <- mc.integrala(ftn, a, b, 500)
```



```
res
## $valor
## [1] 3.222366
##
## $error
## [1] 0.165086
```

Pero aunque aparentemente converge antes, parece no haber una mejora en la precisión de la aproximación. Si calculamos el porcentaje (estimado) de reducción del error:

```
100*(0.1619886-0.1641059)/0.1619886
```

```
## [1] -1.307067
```

El problema es que en este caso se está estimando mal la varianza (asumiendo independencia). Hay que tener cuidado con las técnicas de reducción de la varianza si uno de los objetivos de la simulación es precisamente estimar la variabilidad. En este caso, una versión de la función anterior para integración Monte Carlo con variables antitéticas, con aproximación del error bajo dependencia podría ser:

```
mc.integrala2 <- function(ftn, a, b, n, plot = TRUE, ...) {
  # n es el nº de evaluaciones de la función objetivo (para facilitar comparaciones, solo se generan
  x <- runif(n%/%2, a, b)
  # La siguiente línea solo para representar alternando
  x <- matrix(c(x,a+b-x),nrow=2,byrow=TRUE)
  # bastaría con emplear p.e. c(x,a+b-x)
  fx <- apply(x, 1, ftn)*(b-a)
  corr <- cor(fx[,1], fx[,2])
  fx <- as.numeric(fx)
  return(list(valor=mean(fx), error=2*sd(fx)/sqrt(n)*sqrt(1+corr)))
}

set.seed(54321)
res <- mc.integrala2(ftn, a, b, 500)
res
## $valor
```

```
## [1] 3.222366
##
## $error
## [1] 0.05700069
```

Porcentaje estimado de reducción del error:

```
100*(0.1619886-0.05700069)/0.1619886
```

```
## [1] 64.81191
```

En este caso puede verse que la reducción teórica de la varianza es del 96.7%

10.2 Estratificación

Si se divide la población en estratos y se genera en cada uno un número de observaciones proporcional a su tamaño (a la probabilidad de cada uno) nos aseguramos de que se cubre el dominio de interés y se puede acelerar la convergencia.

- Por ejemplo, para generar una muestra de tamaño n de una $\mathcal{U}(0, 1)$, se pueden generar $l = \frac{n}{k}$ observaciones ($1 \leq k \leq n$) de la forma:

$$U_{j_1}, \dots, U_{j_l} \sim \mathcal{U}\left(\frac{(j-1)}{k}, \frac{j}{k}\right) \text{ para } j = 1, \dots, k.$$

Si en el número de obsevaciones se tiene en cuenta la variabilidad en el estrato se puede obtener una reducción significativa de la varianza.

Ejemplo 10.1 (muestreo estratificado de una exponencial)

Supóngase el siguiente problema (absolutamente artificial pero ilustrativo para comprender esta técnica). Dada una muestra de tamaño 10 de una población con distribución:

$$X \sim \exp(1),$$

se desea aproximar la media poblacional (es sobradamente conocido que es 1) a partir de 10 simulaciones. Supongamos que para evitar que, por puro azar, exista alguna zona en la que la exponencial toma valores, no representada en la muestra simulada de 10 datos, se consideran tres estratos. Por ejemplo, el del 40% de valores menores, el siguiente 50% de valores (intermedios) y el 10% de valores más grandes para esta distribución.

El algoritmo de inversión (optimizado) para simular una $\exp(1)$ es:

1. Generar $U \sim \mathcal{U}(0, 1)$.
2. Hacer $X = -\ln U$.

Dado que, en principio, simulando diez valores $U_1, U_2, \dots, U_{10} \sim \mathcal{U}(0, 1)$, no hay nada que nos garantice que las proporciones de los estratos son las deseadas (aunque sí lo serán en media). Una forma de garantizar el que obtengamos **4, 5 y 1** valores, repectivamente, en cada uno de los tres estratos, consiste en simular:

- 4 valores de $\mathcal{U}[0.6, 1)$ para el primer estrato,
- 5 valores de $\mathcal{U}[0.1, 0.6)$ para el segundo y
- uno de $\mathcal{U}[0, 0.1)$ para el tercero.

Otra forma de proceder consistiría en rechazar valores de U que caigan en uno de esos tres intervalos cuando el cupo de ese estrato esté ya lleno (lo cual no sería computacionalmente eficiente).

El algoritmo con la estratificación propuesta sería como sigue:

1. Para $i = 1, 2, \dots, 10$:
2. Generar U_i :
 - 2a. Generar $U \sim \mathcal{U}(0, 1)$.
 - 2b. Si $i \leq 4$ hacer $U_i = 0.4 \cdot U + 0.6$.
 - 2c. Si $4 < i \leq 9$ hacer $U_i = 0.5 \cdot U + 0.1$.
 - 2d. Si $i = 10$ hacer $U_i = 0.1 \cdot U$.
3. Devolver $X_i = -\ln U_i$.

No es difícil probar que:

- $\text{Var}(X_i) = 0.0214644$ si $i = 1, 2, 3, 4$,
- $\text{Var}(X_i) = 0.229504$ si $i = 5, 6, 7, 8, 9$ y
- $\text{Var}(X_{10}) = 1$.

Como consecuencia:

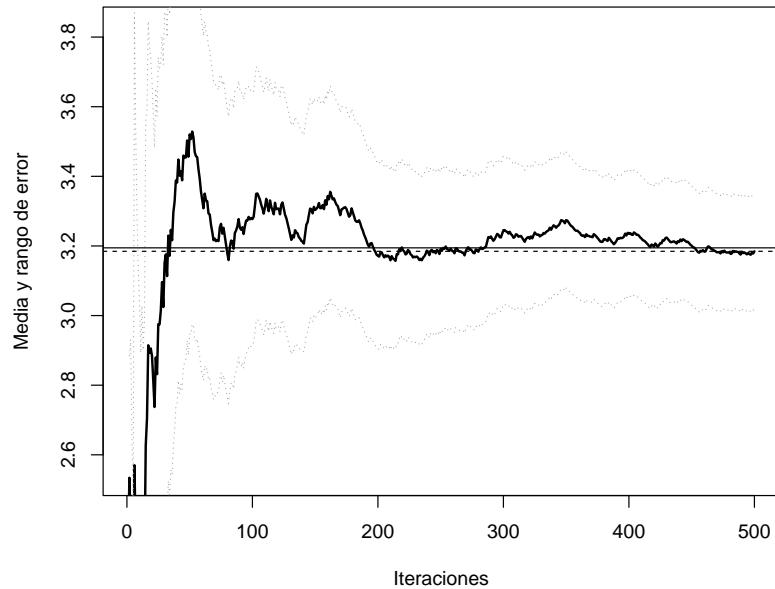
$$\text{Var}(\bar{X}) = \frac{1}{10^2} \sum_{i=1}^{10} \text{Var}(X_i) = 0.022338$$

que es bastante menor que 1 (la varianza en el caso de muestreo aleatorio simple no estratificado).

Ejercicio 10.2 (Integración Monte Carlo con estratificación), considerando k subintervalos regularmente espaciados en el intervalo $[0, 2]$. ¿Cómo varía la reducción en la varianza dependiendo del valor de k ?

```
mc.integrale <- function(ftn, a, b, n, k) {
  # Integración Monte Carlo con estratificación
  l <- n%%k
  int <- seq(a, b, len=k+1)
  x <- runif(l*k, rep(int[-(k+1)], each=1), rep(int[-1], each=1))
  # l uniformes en cada uno de los intervalos  $[(j-1)/k, j/k]$ 
  fx <- sapply(x, ftn)*(b-a)
  return(list(valor=mean(fx), error=2*sd(fx)/sqrt(n)))  # error mal calculado
}

set.seed(54321)
res <- mc.integral(ftn, a, b, 500)
abline(h = theor)
```



```
res
```

```
## $valor
## [1] 3.184612
##
## $error
## [1] 0.1629942
set.seed(54321)
mc.integrale(ftn, a, b, 500, 50)
```

```
## $valor
## [1] 3.193338
##
## $error
## [1] 0.1597952
set.seed(54321)
mc.integrale(ftn, a, b, 500, 100)
```

```
## $valor
## [1] 3.193927
##
## $error
## [1] 0.1599089
```

De esta forma no se tiene en cuenta la variabilidad en el estrato. El tamaño de las submuestras debería incrementarse hacia el extremo superior.

Ejercicio 10.3

Repetir el ejemplo anterior considerando intervalos regularmente espaciados en escala exponencial.

10.3 Variables de control

En este caso se trata de sacar partido tanto a una covarianza positiva como negativa. La idea básica es emplear una variable Y , con media conocida μ_Y , para controlar la variable X (con media desconocida), de forma que ambas variables estén “suficientemente” correlacionadas. La versión “controlada” de X será:

$$X^* = X + \alpha(Y - \mu_Y)$$

con $E(X^*) = E(X) = \theta$. Puede verse que $Var(X^*) = Var(X) + \alpha^2 Var(Y) + 2\alpha Cov(X, Y)$ es mínima para:

$$\alpha^* = -\frac{Cov(X, Y)}{Var(Y)},$$

con $Var(X^*) = Var(X)(1 - \rho^2(X, Y))$ (lo que supone una reducción del $100\rho^2(X, Y)\%$).

En la práctica normalmente α^* no es conocida. Para estimarlo se puede realizar ajuste lineal de X sobre Y (a partir de los datos simulados X_i e Y_i , $1 \leq i \leq n$):

- Si $\hat{x} = \hat{\beta}_0 + \hat{\beta}_1 y$ es la recta ajustada, con $\hat{\beta}_1 = \frac{S_{XY}}{S_Y^2}$ y $\hat{\beta}_0 = \bar{X} - \hat{\beta}_1 \bar{Y}$, la estimación sería:

$$\hat{\alpha}^* = -\hat{\beta}_1$$

- Adicionalmente, para aproximar θ :

$$\begin{aligned}\hat{\theta} &= \bar{X}^* = \bar{X} - \hat{\beta}_1(\bar{Y} - \mu_Y) \\ &= \hat{\beta}_0 + \hat{\beta}_1 \mu_Y\end{aligned}$$

- Si $\mu_Y = 0 \Rightarrow \hat{\theta} = \bar{X}^* = \hat{\beta}_0$.

Ejercicio 10.4 (Integración Monte Carlo con variables de control)

Aproximar la integral anterior empleando la variable $U \sim \mathcal{U}(0, 2)$ para controlar la variable e^U .

Se trata de calcular la media de $\exp(\mathcal{U}(a, b))$:

```
a <- 0; b <- 2
teor <- (exp(b)-exp(a))/(b-a)
teor
```

```
## [1] 3.194528
```

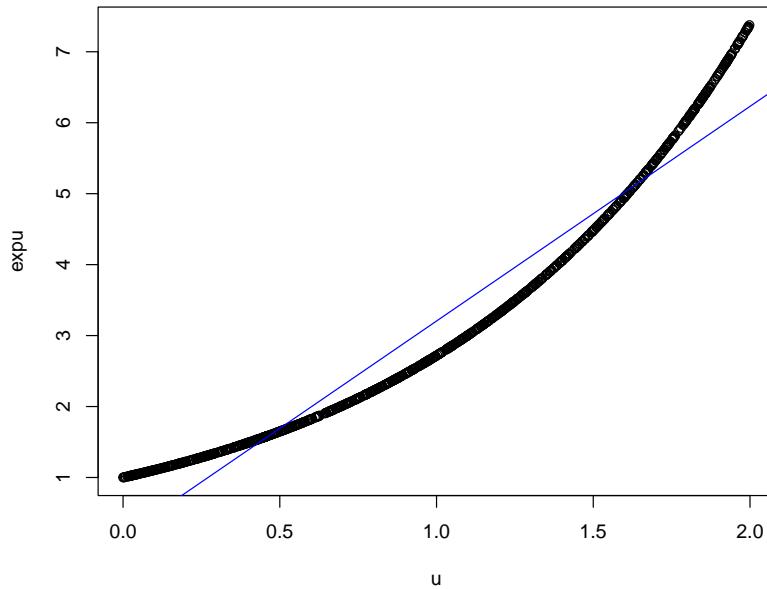
Aproximación clásica por simulación:

```
set.seed(54321)
nsim <- 1000
u <- runif(nsim, a, b)
expu <- exp(u)
mean(expu)
```

```
## [1] 3.182118
```

Con variable control:

```
plot(u, expu)
reg <- lm(expu ~ u)$coef
abline(reg, col='blue')
```



```
# summary(lm(expu ~ u)) # R-squared: 0.9392
reg[1]+reg[2] # Coincidirá con la solución mean(expuc)
```

```
## (Intercept)
##      3.204933
```

Lo siguiente ya no sería necesario:

```
expuc <- expu - reg[2]*(u-1)
mean(expuc)
```

```
## [1] 3.204933
```

Estimación del porcentaje de reducción en la varianza:

```
100*(var(expu)-var(expuc))/var(expu)
```

```
## [1] 93.91555
```

10.4 Números aleatorios comunes

Se trataría de una técnica básica del diseño de experimentos: realizar comparaciones homogéneas (bloquear). Por ejemplo cuando se diseña un experimento para la comparación de la media de dos variables, se pueden emplear las denominadas muestras apareadas, en lugar de muestras independientes.

Supóngamos que estamos interesados en las diferencias entre dos estrategias (e.g. dos estimadores):

$$E(X) - E(Y) = E(X - Y).$$

Para ello se generan dos secuencias X_1, X_2, \dots, X_n , e Y_1, Y_2, \dots, Y_n y se calcula:

$$\bar{X} - \bar{Y} = \frac{1}{n} \sum_{i=1}^n (X_i - Y_i)$$

- Si las secuencias se generan de modo independiente:

$$Var(\bar{X} - \bar{Y}) = \frac{1}{n} (Var(X) + Var(Y))$$

- Si se generar las secuencias empleando **la misma semilla**, los datos son dependientes:

$$\text{Cov}(X_i, Y_i) > 0$$

y tendríamos que:

$$\begin{aligned} \text{Var}(\bar{X} - \bar{Y}) &= \frac{1}{n^2} \sum_{i=1}^N \text{Var}(X_i - Y_i) = \frac{1}{n} \text{Var}(X_i - Y_i) \\ &= \frac{1}{n} (\text{Var}(X_i) + \text{Var}(Y_i) - 2\text{Cov}(X_i, Y_i)) \\ &\leq \frac{1}{n} (\text{Var}(X_i) + \text{Var}(Y_i)) \end{aligned}$$

En el capítulo de aplicaciones de la simulación se empleó esta técnica para comparar distribuciones de estimadores...

10.5 Ejercicios

Ejercicio 10.5 (propuesto)

Aproximar mediante integración Monte Carlo (clásica) la media de una distribución exponencial de parámetro 1/2:

$$I = \int_0^\infty \frac{x}{2} e^{-\frac{x}{2}} dx$$

y representar gráficamente la aproximación en función de n . Comparar los resultados con los obtenidos empleando variables antitéticas, ¿se produce una reducción en la varianza?

Nota: Puede ser recomendable emplear el método de inversión para generar las muestras (antitéticas) de la exponencial.

MC clásico:

```
nsim <- 1000
lambda <- 0.5
set.seed(1)
x <- -log(runif(nsim)) / lambda
# Aprox por MC da media
mean(x) # valor teor 1/lambda = 2
```

```
## [1] 1.97439
```

```
# Aprox da precisión
var(x)
```

```
## [1] 3.669456
```

MC con variables antitéticas:

```
# xa <-
# mean(xa) # Aprox por MC da media (valor teor 1/lambda = 2)
# var(xa) # Aprox da precisión suponiendo independencia
# corr <- cor(x1, x2)
# var(xa)*(1 + corr) # Estimación varianza suponiendo dependencia
```

Estimación del porcentaje de reducción en la varianza

```
# 100*(var(x) - var(xa))/var(x)
```


Referencias

Bibliografía básica

- Cao, R. (2002). *Introducción a la simulación y a la teoría de colas*. NetBiblo.
- 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

Libros

- Azarang, M. R. y García Dunna, E. (1996). *Simulación y análisis de modelos estocásticos*. McGraw-Hill.
- Bratley, P., Fox, B.L. y Schrage L.E. (1987). *A guide to simulation*. Springer-Verlag.
- Cao R. y Fernández-Casal R. (2020). *Técnicas de Remuestreo* https://rubenfcasal.github.io/book_remuestreo.
- 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 deterministic methods*. Oxford University Press.
- Gentle, J.E. (1998). *Random number generation and Monte Carlo methods*. Springer-Verlag.
- Hyndman, R.J. y Athanasopoulos, G. (2018). *Forecasting: principles and practice*. OTexts. Disponible online: 2nd edition (forecast), 3rd edition (fable).
- Hofert, M. (2018). *Elements of Copula Modeling with R*, Springer.
- Hörmann, W. et al. (2004). *Automatic Nonuniform Random Variate Generation*. Springer.
- 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.
- 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.
- Moeschlin, O., Grycko, E., Pohl, C. y Steinert, F. (1998). *Experimental stochastics*. Springer-Verlag.
- Nelsen, R.B. (2006). *An introduction to copulas*, 2^a ed., Springer.
- Nelson, R. (1995). *Probability, stochastic processes, and queueing theory: the mathematics of computer performance modelling*. 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.

Artículos

- 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.
- 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.
- 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.
- 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.
- 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.

Apéndice A

Enlaces

Recursos para el aprendizaje de R

A continuación se muestran algunos recursos que pueden ser útiles para el aprendizaje de R y la obtención de ayuda (basados en el post <https://rubenfcasal.github.io/post/ayuda-y-recursos-para-el-aprendizaje-de-r>, que puede estar más actualizado).

Ayuda online:

- Ayuda en línea sobre funciones o paquetes: RDocumentation
- Buscador RSeek
- StackOverflow, en castellano
- Cross Validated

Cursos (gratuitos):

- Coursera:
 - Introducción a Data Science: Programación Estadística con R
 - Mastering Software Development in R
- DataCamp:
 - Introducción a R
- Stanford online:
 - Statistical Learning
- Curso UCA: Introducción a R, R-commander y shiny
- Curso R CODER
- Udacity: Data Analysis with R
- Swirl Courses: se pueden hacer cursos desde el propio R con el paquete swirl.

Para información sobre cursos en castellano se puede recurrir a la web de R-Hispano en el apartado formación. Algunos de los cursos que aparecen en entradas antiguas son gratuitos. Ver: Cursos MOOC relacionados con R.

Libros

- *Iniciación:*
 - 2011 - The Art of R Programming. A Tour of Statistical Software Design, (No Starch Press)
 - R for Data Science (online, online-castellano, O'Reilly)

- Hands-On Programming with R: Write Your Own Functions and Simulations, by Garrett Grolemund (O'Reilly)
 - **Avanzados:**
 - 2008 - Software for Data Analysis: Programming with R - Chambers (Springer)
 - Advanced R by Hadley Wickham (online: 1^a ed, 2^a ed, Chapman & Hall)
 - R packages by Hadley Wickham (online, O'Reilly)
 - **Bookdown:** el paquete `bookdown` de R permite escribir libros empleando R Markdown y compartirlos. En <https://bookdown.org> está disponible una selección de libros escritos con este paquete (un listado más completo está disponible aquí). Algunos libros en este formato en castellano son:
 - Fernández-Casal R., Roca-Pardiñas J. y Costa J. (2019). *Introducción al Análisis de Datos con R*. <https://rubenfcasal.github.io/intror> (github).
 - Cao R. y Fernández-Casal R. (2020). *Técnicas de Remuestreo* https://rubenfcasal.github.io/book_remuestreo (github).
 - Fernández-Casal R. y Costa J. (2020). *Aprendizaje Estadístico*. https://rubenfcasal.github.io/aprendizaje_estadistico (github).
 - López-Taboada G. y Fernández-Casal R. (2020). *Prácticas de Tecnologías de Gestión y Manipulación de Datos*. <https://gltaboada.github.io/tgdbook> (github).
 - Fernández-Casal R. y Cotos-Yáñez T.R. (2018). *Escritura de libros con bookdown*. https://rubenfcasal.github.io/bookdown_intro (github). Incluye un apéndice con una introducción a Rmarkdown.
 - Gil Bellotta C.J. (2018). R para profesionales de los datos: una introducción.
 - Quintela del Rio A. (2019). Estadística Básica Edulcorada.
- Material online:** en la web se puede encontrar mucho material adicional, por ejemplo:
- CRAN: Other R documentation
 - RStudio:
 - Online learning, Webinars
 - tidyverse: dplyr, ggplot2, tibble, tidyr, stringr, readr, purrr.
 - RMarkdown, shiny, sparklyr
 - CheatSheets github: RMarkdown, Shiny, dplyr, tidyr, stringr.
 - Blogs en inglés:
 - <https://www.r-bloggers.com/>
 - <https://www.littlemissdata.com/blog/rstudioconf2019>
 - RStudio: <https://blog.rstudio.com>
 - Microsoft Revolutions: <https://blog.revolutionanalytics.com>
 - Blogs en castellano:
 - <https://www.datanalytics.com>
 - <http://oscarperpinan.github.io/R>
 - <http://rubenfcasal.github.io>
 - Listas de correo:
 - Listas de distribución de r-project.org: <https://stat.ethz.ch/mailman/listinfo>
 - Búsqueda en R-help: <http://r.789695.n4.nabble.com/R-help-f789696.html>

- Búsqueda en R-help-es: <https://r-help-es.r-project.narkive.com>
<https://grokbase.com/g/r/r-help-es>
- Archivos de R-help-es: <https://stat.ethz.ch/pipermail/r-help-es>

Apéndice B

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 anteriores¹. 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 3.3.2. No obstante, en la última sección se describirán, únicamente con fines ilustrativos, algunos de los primeros métodos diseñados específicamente para generadores aleatorios.

B.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 interesaría 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
$\begin{cases} H_0 : F = \mathcal{N}(0, 1) \\ H_1 : F \neq \mathcal{N}(0, 1) \end{cases}$	$\begin{cases} H_0 : F = \mathcal{N}(\mu, \sigma^2) \\ H_1 : F \neq \mathcal{N}(\mu, \sigma^2) \end{cases}$

Entre los métodos gráficos habituales estarían: histograma, gráfico de la densidad suavizada, gráfico

¹El 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.

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.

B.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 $n_k = \sum_{i=1}^n \mathbf{1}(X_i \in [L_{k-1}, L_k])$, 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):

$$\hat{f}_n(x) = \frac{n_i}{n(L_k - L_{k-1})}$$

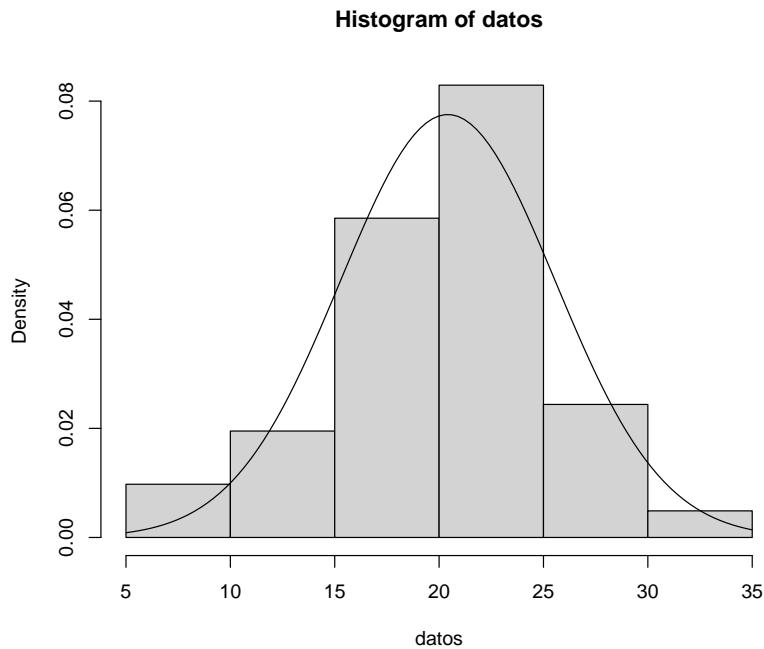
Como ya se ha visto anteriormente, en R podemos generar este gráfico con la función `hist()` del paquete base. Algunos de los principales parámetros (con los valores por defecto) son los siguientes:

```
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 B.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()`.

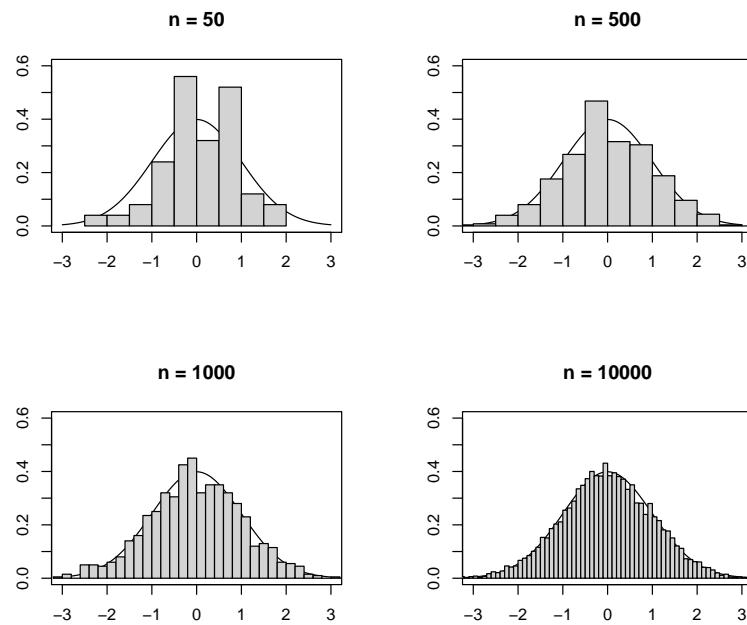


Figura B.1: Convergencia del histograma a la densidad teórica.

B.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 $X_{(1)} \leq X_{(2)} \leq \dots \leq X_{(n)}$ y:

$$F_n(x) = \begin{cases} 0 & \text{si } x < X_{(1)} \\ \frac{i}{n} & \text{si } X_{(i)} \leq x < X_{(i+1)} \\ 1 & \text{si } X_{(n)} \leq x \end{cases}$$

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)
```

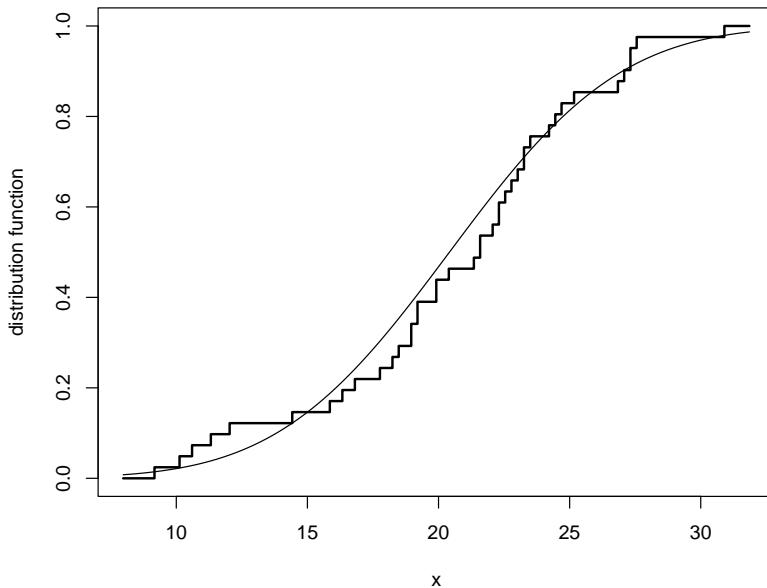


Figura B.2: Comparación de la distribución empírica de los datos de ejemplo con la función de distribución de la aproximación normal.

B.1.3 Gráficos P-P y Q-Q

El gráfico de probabilidad (o de probabilidad-probabilidad) es el gráfico de dispersión de:

$$\{(F_0(x_i), F_n(x_i)) : i = 1, \dots, n\}$$

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).

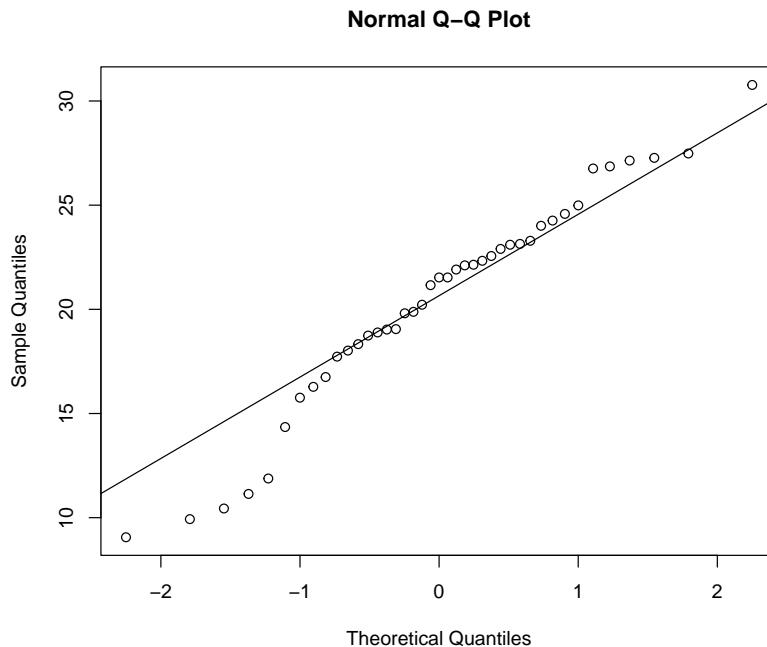
El gráfico Q-Q (cuantil-cuantil) es equivalente al anterior pero en la escala de la variable:

$$\{(q_i, x_{(i)}) : i = 1, \dots, n\}$$

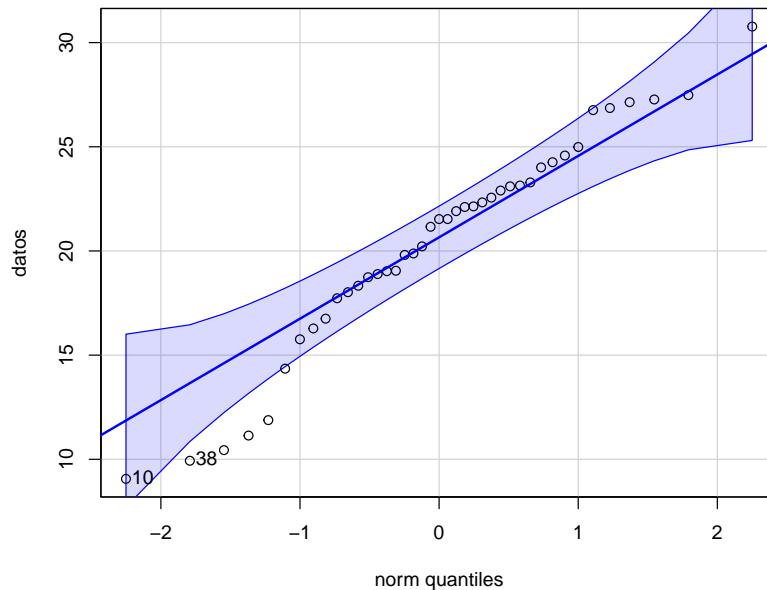
siendo $x_{(i)}$ los cuantiles observados y $q_i = F_0^{-1}(p_i)$ los esperados² bajo H_0 .

Ejemplo:

```
qqnorm(datos)
qqline(datos)
```



```
require(car)
qqPlot(datos, "norm")
```



```
## [1] 10 38
```

²Típicamente $\{p_i = \frac{(i-0.5)}{n} : i = 1, \dots, n\}$.

B.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, \dots, 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 corregir 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):

$$\chi^2 = \sum_{i=1}^k \frac{(n_i - e_i)^2}{e_i} \underset{\text{aprox.}}{\sim} \chi^2_{k-r-1}, \text{ si } H_0 \text{ cierta}$$

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 Ejercicio 7.3 de la Sección 7.7.3 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 siguiente función (que imita a las incluídas en R):

```
#-----
# chisq.test.cont(x, distribution, nclasses, output, nestpar,...)
#-----
# Realiza el test chi-cuadrado de bondad de ajuste para una distribución continua
# discretizando en intervalos equiprobables.
# Parámetros:
#   distribution = "norm", "unif", etc
#   nclasses = floor(length(x)/5)
#   output = TRUE
#   nestpar = 0= nº de parámetros estimados
#   ... = parámetros distribución
# Ejemplo:
#   chisq.test.cont(x, distribution="norm", nestpar=2, mean=mean(x), sd=sqrt((nx-1)/nx)*sd(x))
#-----
chisq.test.cont <- function(x, distribution = "norm", nclasses = floor(length(x)/5),
  output = TRUE, nestpar = 0, ...) {
  # Funciones distribución
  q.distrib <- eval(parse(text = paste("q", distribution, sep = "")))
```

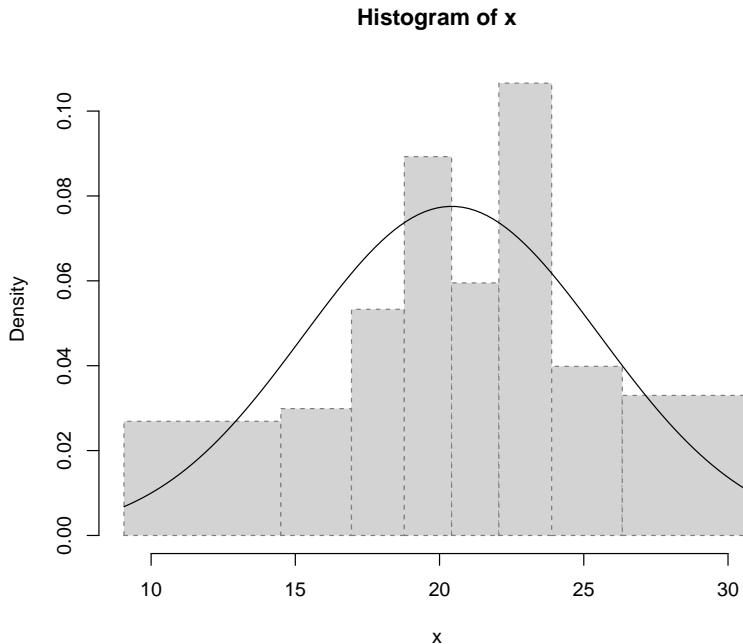
```

d.distrib <- eval(parse(text = paste("d", distribution, sep = "")))
# Puntos de corte
q <- q.distrib((1:(nclases - 1))/nclases, ...)
tol <- sqrt(.Machine$double.eps)
xbreaks <- c(min(x) - tol, q, max(x) + tol)
# Gráficos y frecuencias
if (output) {
  xhist <- hist(x, breaks = xbreaks, freq = FALSE, lty = 2, border = "grey50")
  curve(d.distrib(x, ...), add = TRUE)
} else {
  xhist <- hist(x, breaks = xbreaks, plot = FALSE)
}
# Cálculo estadístico y p-valor
O <- xhist$counts # Equivalente a table(cut(x, xbreaks)) pero más eficiente
E <- length(x)/nclases
DNAME <- deparse(substitute(x))
METHOD <- "Pearson's Chi-squared test"
STATISTIC <- sum((O - E)^2/E)
names(STATISTIC) <- "X-squared"
PARAMETER <- nclases - nestpar - 1
names(PARAMETER) <- "df"
PVAL <- pchisq(STATISTIC, PARAMETER, lower.tail = FALSE)
# Preparar resultados
classes <- format(xbreaks)
classes <- paste("(", classes[-(nclases + 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")
}

```

Continuando con el ejemplo anterior, podríamos contrastar normalidad mediante:

```
chisq.test.cont(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

```

B.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 $F_n(X_{(i)}) = \frac{i}{n}$:

$$\begin{aligned}
 D_n &= \max_{1 \leq i \leq n} \left\{ \frac{i}{n} - F_0(X_{(i)}), F_0(X_{(i)}) - \frac{i-1}{n} \right\} \\
 &= \max_{1 \leq i \leq n} \{D_{n,i}^+, D_{n,i}^-\}
 \end{aligned}$$

Si H_0 es simple y F_0 es continua, la distribución del estadístico D_n bajo H_0 no depende de 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:

$$p = P(D_n \geq d) \leq \alpha.$$

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.

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 8.5 de la Sección 8.3).

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
```

B.2 Diagnosis de la independencia

Los métodos “clásicos” de inferencia estadística se basan en suponer que las observaciones X_1, \dots, 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: datos simulados

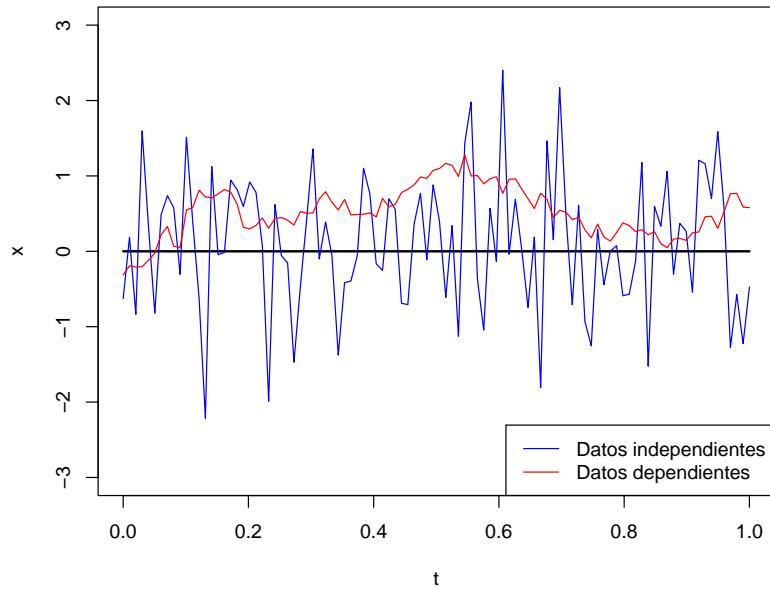
Consideramos un proceso temporal estacionario con dependencia exponencial (la dependencia entre las observaciones depende del “salto” entre ellas; ver Ejemplo 7.4 en la Sección 7.3).

```
n <- 100          # N° 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', 'red'))
```



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

B.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
- Contrastos:
 - Tests basados en rachas
 - Test de Ljung-Box

B.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))
```

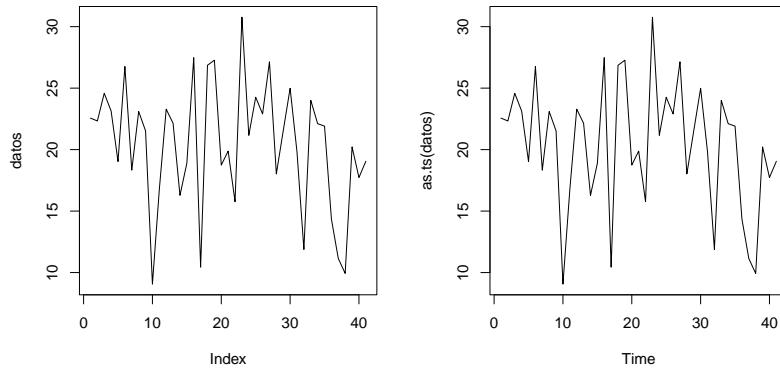


Figura B.3: Ejemplos de gráficos secuenciales.

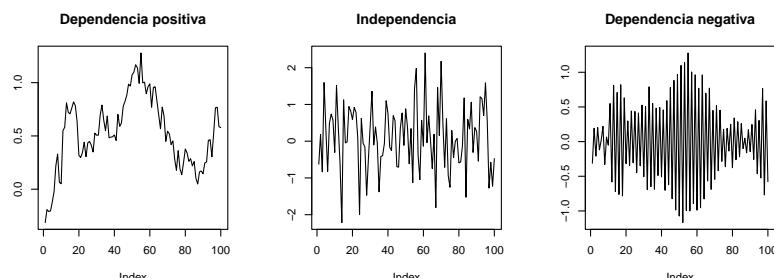
```
par(old.par)
```

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)
```

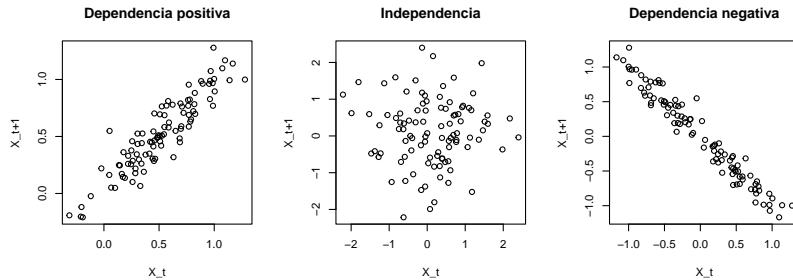
pero suele ser preferible emplear un gráfico de dispersión retardado.

B.2.3 Gráfico de dispersión 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')
```

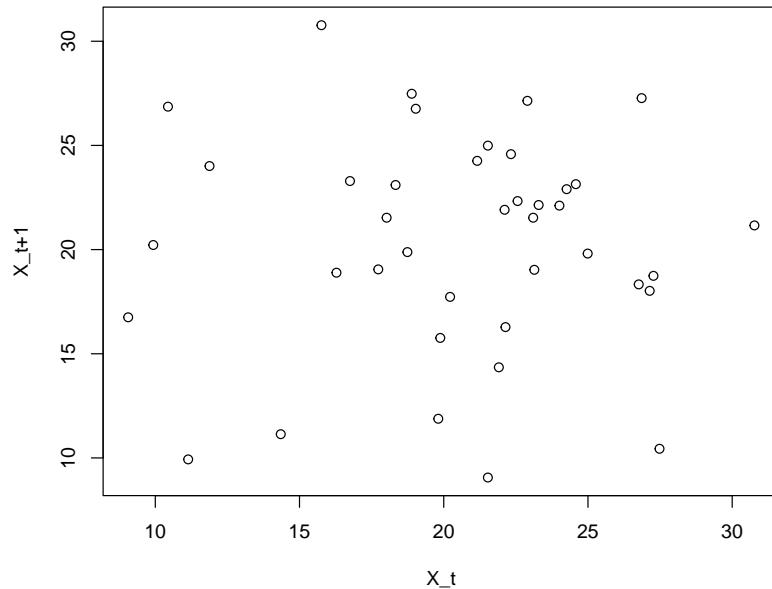


```
par(old.par)
```

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 dispersión 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: Calidad de un generador aleatorio

En el caso de una secuencia muy grande de número pseudoaleatorios (supuestamente independientes), sería muy difícil 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")
```

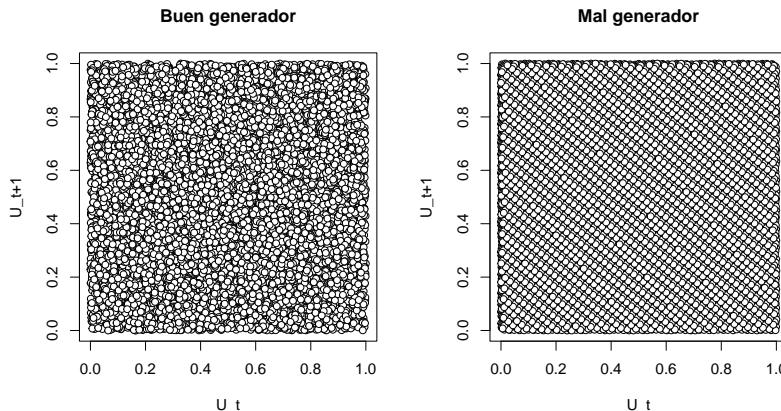


Figura B.4: Ejemplos de gráficos de dispersión retardados de dos secuencias de longitud 10000.

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 3.1 de la Sección 3.1).

B.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:

$$\rho(X_i, X_{i+k}) = \frac{Cov(X_i, X_{i+k})}{\sigma(X_i)\sigma(X_{i+k})}$$

- En el caso de datos homogéneos (estacionarios) la correlación sería función únicamente del salto:

$$\rho(X_i, X_{i+k}) \equiv \rho(k)$$

denominada función de autocorrelación simple (fas) o correlograma.

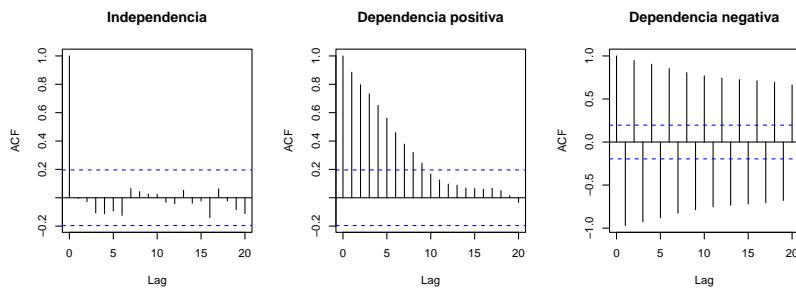
- Su estimador es el correlograma muestral:

$$r(k) = \frac{\sum_{i=1}^{n-k} (X_i - \bar{X})(X_{i+k} - \bar{X})}{\sum_{i=1}^n (X_i - \bar{X})^2}$$

- 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:

$$r(k) \underset{\text{aprox.}}{\sim} N\left(\rho(k), \frac{1}{n}\right)$$

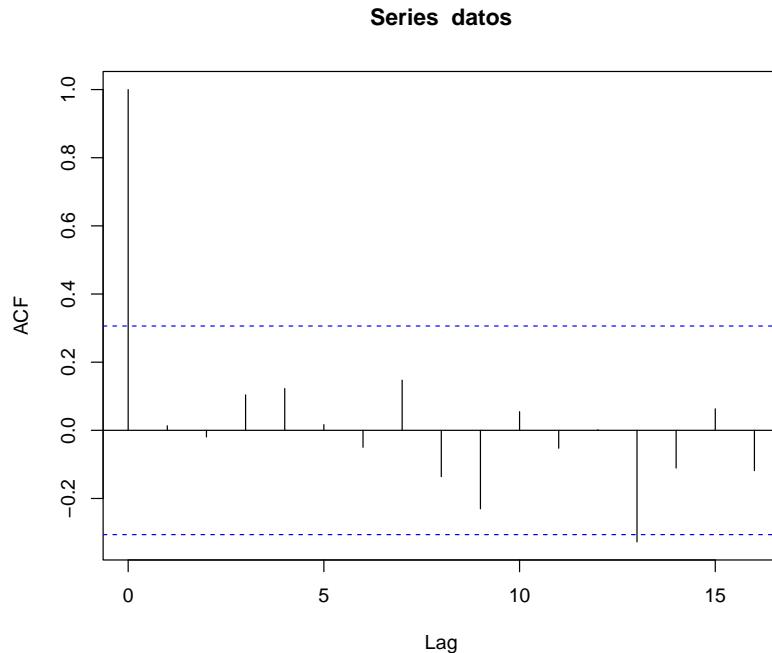
- Si el tamaño muestral es grande, podríamos aceptar $H_0 : \rho(k) = 0$ si:

$$|r(k)| < \frac{2}{\sqrt{n}}$$

- 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

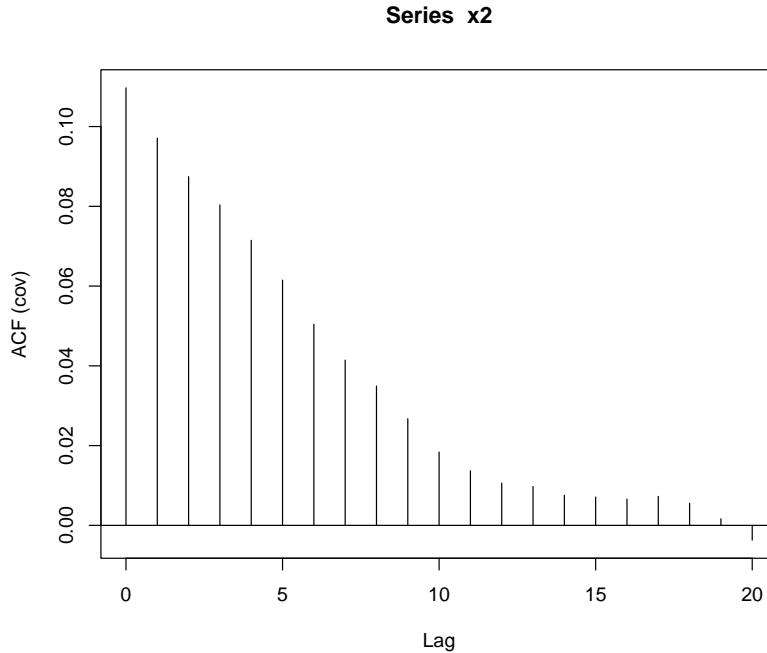
```
acf(datos) # correlaciones
```



La función `acf` también permite estimar el covariograma³.

³En algunos campos, como en estadística espacial, en lugar del covariograma se suele emplear el semivariograma $\gamma(k) = C(0) - C(k)$.

```
covar <- acf(x2, type = "covariance")
```



B.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:

+ + + + - - - + + + - - + + + + + + + + - - -

y nos interesa contrastar:

$$\begin{cases} H_0 : \text{La muestra es aleatoria} \\ H_1 : \text{La muestra no es aleatoria} \end{cases}$$

Una *racha* es una secuencia de observaciones iguales (o similares):

$\overbrace{+++}^1 \overbrace{---}^2 \overbrace{++}^3 \overbrace{- -}^4 \overbrace{++++}^5 \overbrace{---}^6$

- 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:

$$R \underset{\text{aprox.}}{\sim} N \left(1 + \frac{2n_1 n_2}{n}, \frac{2n_1 n_2 (2n_1 n_2 - n)}{n^2(n-1)} \right)$$

siendo n_1 y n_2 el número de signos + y - en la muestra, respectivamente ($n_1 + n_2 = n$). Para tamaños muestrales 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.

B.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:

$$\begin{cases} H_0 : \rho_1 = \rho_2 = \dots = \rho_m = 0 \\ H_1 : \rho_i \neq 0 \text{ para algún } i \end{cases}$$

- 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:

$$Q = n(n+2) \sum_{k=1}^m \frac{r(k)^2}{n-k} \underset{aprox.}{\sim} \chi_m^2, \text{ si } H_0 \text{ es cierta.}$$

- Se rechaza H_0 si el valor observado es grande ($Q \geq \chi_{m,1-\alpha}^2$):

$$p = P(\chi_m^2 \geq Q)$$

- 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 `dwttest()` del paquete `lmtest`.

B.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, \dots, 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 $\{1, \dots, K\}$.

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 3.3.2.

B.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 B.1.4 e implementado en la función `chisq.test.cont()`. 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:

```

set.seed(1)
u <- runif(1000)

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
# Equivalente a
# source("Test Chi-cuadrado continua.R")
# chisq.test.cont(u, distribution = "unif", nclasses = k, output = FALSE, min = 0, max = 1)
```

B.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.

B.3.3 El contraste del poker

En el contrato 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 sola 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:

$$\begin{aligned} P(\text{AAAAA}) &= 0.0001, P(\text{AAAAB}) = 0.0045, P(\text{AAABB}) = 0.0090, \\ P(\text{AAABC}) &= 0.0720, P(\text{AABBC}) = 0.1080, P(\text{AABCD}) = 0.5040, \\ P(\text{ABCDE}) &= 0.3024. \end{aligned}$$

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 $d = K$.

B.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

$$P(Q = K), P(Q = K + 1), \dots, P(Q = M - 1), P(Q \geq M),$$

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 $Q = 5, Q = 6, \dots, Q = 19, Q \geq 20$, 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 C

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:

$$MISE(\hat{f}) = \int E \left[(\hat{f}(x) - f(x))^2 \right] dx$$

Cuando el numero de dimensiones es pequeño, nos puede interesar emplear un método numérico para aproximar este tipo de integrales.

C.1 Integración numérica unidimensional

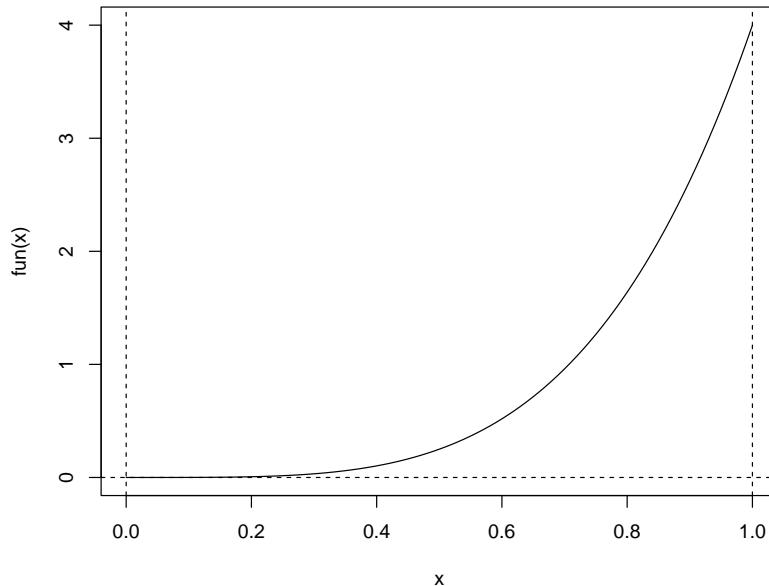
Supongamos que nos interesa aproximar una integral de la forma:

$$I = \int_a^b h(x) dx.$$

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)
```



C.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:

$$\int_a^b f(x) dx \approx \frac{h}{2} [f(a) + 2f(a+h) + 2f(a+2h) + \dots + f(b)]$$

```
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 $O(n^{-\frac{2}{d}})$.

C.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 $i = 0, 1, \dots, n$. 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 approxima 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)))
  # Una cota del error en valor absoluto es:
  # h^4*(b-a)*max(c(f.vec1, fvec.2))^4/180.
}
```

```
simpson(fun, 0, 1, 20)
```

```
## [1] 0.8000033
```

El máximo error (en el caso unidimensional) viene dado por la expresión:

$$\frac{(b-a)^5}{180n^4} \max_{a \leq \xi \leq b} |f^{(4)}(\xi)|.$$

En el caso general multidimensional sería $O(n^{-\frac{4}{d}})$.

C.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

C.1.4 Comandos de R

```

integrate(fun, 0, 1)    # Permite límites infinitos

## 0.8 with absolute error < 8.9e-15
## Cuidado: fun debe ser vectorial...

require(MASS)
area(fun, 0, 1)

## [1] 0.8000001

```

C.2 Integración numérica bidimensional

Supongamos que nos interesa aproximar una integral de la forma:

$$I = \int_{a_x}^{b_x} \int_{a_y}^{b_y} f(x, y) dy dx$$

Consideraremos como ejemplo:

$$\int_{-1}^1 \int_{-1}^1 (x^2 - y^2) dx dy = 0$$

```
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]
```

C.2.1 Representación gráfica

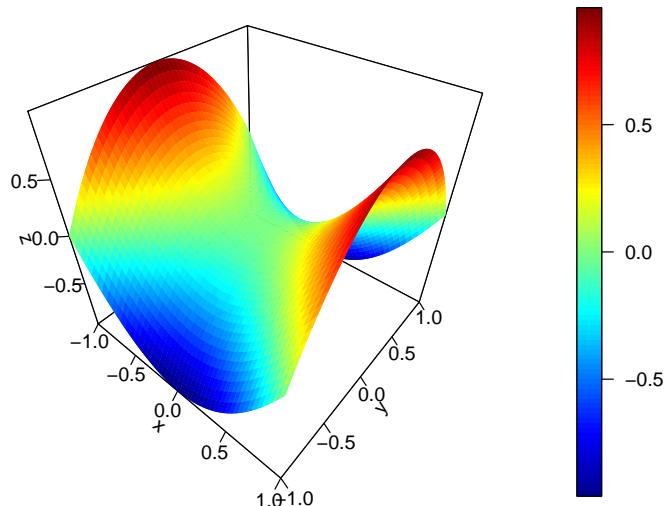
Puede ser de utilidad las herramientas de los paquetes `plot3D` y `plot3Drgl` (también se pueden utilizar las funciones `spersp`, `simage`, `spoints` y `splot` del paquete `npsp`).

```
if(!require(plot3D)) stop('Required package `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")
```



C.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
```

C.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.](http://tolstoy.newcastle.edu.au/)

Alternativamente se podría emplear la función `adaptIntegrate()` del paquete `cubature`.