Simulación Monte Carlo con R

Francisco J. Gálvez

11 de julio de 2014

Índice

1. Prólogo

El objetivo de este trabajo no es el de redactar un manual de R, para eso existen una gran variedad de libros y documentos que pueden encontrase en Internet y en librerías. En este trabajo se pretende dar un idea introductoria de las capacidades generales del entrono R, animando a la instalación y prueba del mismo y a consultar fuentes más especializadas para profundizar en su conocimiento. Al mismo tiempo también se exponen de forma básica los conceptos de los métodos Monte Carlo, y se ve como puede utilizarse el entorno R para la realización de cálculos mediante dichos métodos.

2. Introducción

El método de Monte Carlo es un método no determinista que se utiliza para aproximar expresiones matemáticas complejas y costosas de evaluar con exactitud. Uno de los pilares del método es la generación de números aleatorios, de hecho su nombre viene del famoso casino de Monte Carlo en alusión a la ruleta como elemento generador de números aleatorios.

El método Monte Carlo surgió como solución a un problema en física de partículas en los años cuarenta, para tratar de resolver las ecuaciones integro-diferenciales que gobiernan la dispersión, absorción y difusión de neutrones. Actualmente se aplica a una gran variedad de problemas matemáticos, ya sean de tipo estocástico o determinista, proporcionando soluciones bastante certeras, aunque siempre con un margen de error. El error absoluto que proporciona el Método de Monte Carlo decrece como $\frac{1}{\sqrt{N}}$ en virtud del teorema del límite central.

El desarrollo de los métodos de Monte Carlo ha venido impulsado por el avance en la tecnología informática y la capacidad de computación desarrollada en los últimos años. Estos avances han dado lugar a la simulación computacional como elemento de investigación en muchas áreas.

La esencia del método consiste en que resulta mucho más simple tener una idea del resultado general haciendo pruebas con subconjuntos aleatorios de una muestra y contando sus proporciones que computando todos las posibilidades de combinación existentes formalmente.

3. El Método Monte Carlo

Antes de entrar en el detalle y las técnicas del método Monte Carlo vamos a establecer algunos conceptos que son básicos y de los que se hace uso cuando se trabaja con el método Monte Carlo.

3.1. Teoremas de Convergencia

El error asociado a las técnicas Monte Carlo está en función del número de pruebas que se realizan en la forma $\frac{1}{\sqrt{N}}$. Se puede apreciar que para añadir una cifra decimal al resultado es necesario elevar al cuadrado el número de pruebas a realizar, lo cual supone una convergencia muy lenta a medida que se desea avanzar en la obtención de mayor precisión. La ley de los grandes números y el teorema del límite central son dos recursos utilizados por el método de Monte Carlo para justificar la simulación de las funciones a calcular.

3.1.1. Ley de los grandes números

La ley de los grandes números, también llamada ley del azar, afirma que al repetir un experimento aleatorio un cierto número de veces, la frecuencia relativa de cada suceso elemental tiende a aproximarse a un número fijo, llamado probabilidad del suceso.

Sea $\bar{x}$ el promedio de N variables aleatorias independientes $\{x_i\}$, lo que viene a decir la ley de los grandes números es que cuando N es grande, el promedio $\bar{x}$ de $\{x_i\}$ converge a $E[X] = \mu$

$$P\left(\lim_{N\to\infty} \bar{x}_N = E[X]\right) = 1 \tag{1}$$

Esto viene a decir que cuando se hace un muestreo repetitivo, la esperanza de una variable aleatoria converge al promedio de esa variable.

3.1.2. El Teorema del Límite Central

Muchas de las variables aleatorias de interés en física ...

Sea X_i, con $i = 1, 2, \ldots, n$, una muestra de variables aleatorias independientes que vienen descritas por una función de densidad de probabilidad denotada por $f_i(x)$ con valor medio μ_i y varianza σ^2

Se define ahora la variable aleatoria $z = \sum_i \frac{X_i}{n}$, esto es la media de las muestras X_i, que tiene las siguientes propiedades:

i) El valor esperado viene dado por $E[z] = \frac{\sum_i \mu_i}{n}$

ii) Su varianza viene dada por: $V[z] = \frac{\sum_i \sigma_i^2}{n^2}$

iii) En la medida en la que $n \longrightarrow \infty$ la función de probabilidad de z tiende a una gaussiana con media $E[z]$ y varianza $V[z]$ descritos en i) y ii)

Para que se cumpla el Teorema del Límite Central, las funciones de densidad de probabilidad $f_i(x)$ de las muestras X_i deben tener media y varianza formales, por ejemplo si alguna de las muestras tuviese una distribucion de Cauchy (No tiene media, ni varianza ni momentos definidos), entonces no podría aplicarse el Teorema del Límite Central.

Los puntos i) y ii) son fácilmente demostrables ya que, para el valor medio se tiene:

$$\begin{aligned} E[z] &= \frac{1}{n}\left(E[X_1], E[X_2]+, \ldots, +E[X_n]\right) = \\ &= \frac{1}{n}\left(\mu_1 + \mu_2 + \ldots + \mu_n\right) = \\ &= \frac{\sum_i \mu_i}{n} \end{aligned} \tag{2}$$

y para la varianza:

$$\begin{aligned} V[z] &= V\left[\frac{1}{n}\left(X_1 + X_2 + \ldots + X_n\right)\right] = \\ &= \frac{1}{n^2}\left(V[X_1] + V[X_2] + \ldots + V[X_n]\right) = \\ &= \frac{\sum_i \sigma_i}{n^2} \end{aligned} \tag{3}$$

El punto iii) se demuestra haciendo uso de las propiedades de la función generatriz para z que se denota por $g_z(t)$ y viene data por:

$$g_z(t) = \prod_1^n g_{X_i}\left(\frac{t}{n}\right) \tag{4}$$

La definición de la función generatriz para las variables X_i es:

$$\begin{aligned} g_{X_i}\left(\frac{t}{n}\right) &= 1 + \frac{t}{n}E[X_i] + \frac{1}{2}\frac{t^2}{n^2}E[X_i^2] + \ldots \\ &= 1\mu_i\frac{t}{n} + \frac{1}{2}(\sigma_i^2 + \mu^2)\frac{t^2}{n^2} + \ldots \end{aligned}$$

y en la medida en la que $n \longrightarrow \infty$ se tiene:

$$g_{X_i}\left(\frac{t}{n}\right) \simeq \exp\left(\frac{\mu_i \cdot t}{n} + \frac{1}{2}\sigma_i^2\frac{t^2}{n^2}\right) \tag{5}$$

Por tanto:

$$g_z(t) \simeq \prod \exp\left(\frac{\mu_i \cdot t}{n} + \frac{1}{2}\sigma_i^2\frac{t^2}{n^2}\right) = \exp\left(\frac{\sum_i \mu_i}{n}t + \frac{1}{2}\frac{\sum_i \sigma_i^2}{n^2}t^2\right) \tag{6}$$

Comparando el resultado con la forma que tiene la función generatriz de una distribución gaussiana:

$$g_x(t) = \exp\left(\mu \cdot t + \frac{1}{2}\sigma^2 \cdot t^2\right) \tag{7}$$

se puede apreciar que la función de densidad de probabilidad $g(z)$ de la variable z tiende a una distribución gaussiana con media $sum_i \frac{\mu_i}{n}$ y varianza $\sum_i \frac{\sigma_i^2}{n^2}$ y en concreto, si se considera que z es la media de n medidas independientes de la misma variable aleatoria X (de tal forma que $X_i = X$ para $i = 1, 2, \ldots, n$), entonces a medida que $n \longrightarrow \infty$, z tiene una distribución gaussiana con media μ y varianza $\frac{\sigma^2}{n}$.

3.2. Generación de Variables Aleatorias

Además de la convergencia de la varianza, un elemento clave en el método Monte Carlo es la Generación de números pseudo-aleatorios. De hecho, generar números aleatorios había sido hasta la aparición de los ordenadores una tarea bastante ardua. Sin embargo el desarrollo de las tecnología de la información ha facilitado enormemente esta labor, aunque hay que decir que los números que genera un computador no son totalmente aleatorios, ya que se generan en base a un algoritmo y por tanto, una vez conocido el algoritmo se pueden predecir los números que genera. La generación de números aleatorios puros solo tiene lugar mediante procesos naturales cuyos sucesos son totalmente impredecibles, sin embargo la computación electrónica nos brinda la oportunidad de generar grandes cantidades de números en intervalos de tiempo muy pequeños y que sin ser totalmente aleatorios, son prácticamente indistinguibles de los números aleatorios reales.

Normalmente todos los algoritmos hacen uso de lo que se denomina semilla. Una semilla es un número aleatorio inicial para comenzar a generar la secuencia de números aleatorios. En muchos casos esta semilla viene establecida por una marca de tiempo. Hay algunas propiedades que un generador de números aleatorios debe cumplir si se desea certificar la aleatoriedad de los números que genera.

Sea un espacio muestral Ω, el cual contiene todos los valores posibles de una variable aleatoria $X = x_1, x_2, \ldots, x_n$. Dicha variable aleatoria tiene un función de probabilidad $f(x)$ que cumple:

$$\int_\Omega f(x)dx = 1 \tag{8}$$

Extraer una muestra consiste en producir una secuencia de variables aleatorias de X tales que para cualquier subconjunto Ω' de Ω se cumple que:

$$P(X \in \Omega') = \int_{\Omega'} f(x)dx \leq 1 \tag{9}$$

Entonces, la generación de variables aleatorias, consiste en producir variables $X_1, X_2, \ldots$ según una distribución de probabilidad cualquiera, siempre que dispongamos de variables aleatorias $\chi_1, \chi_s, \ldots$ que se distribuyen de manera uniforme en Ω.

Las variables aleatorias uniformemente distribuidas se generan mediante lo que denominamos un generador de números aleatorios. Para que los números generados sean de calidad, el generador de números pseudo-aleatorios debe cumplir las siguientes condiciones:

1. Equidistribución. Los número aleatorios deben repetirse por igual, tal y como lo harían en una distribución uniforme verdadera.
2. Periodo largo. Todos los generadores de números aleatorios tienen un periodo a partir del cual comienza a repetirse de nuevo la secuencia de números. Es necesario que este periodo sea lo más largo posible para evitar que se agote una secuencia.
3. Repetitividad. Es necesario que el generador permita repetir la misma secuencia, dado que existen situaciones en las que es necesario repetir varias veces un cálculo con los mismos valores.
4. Largas subsecuencias disjuntas. Es importante que las secuencias generadas no guarden correlación alguna entre ellas.
5. Portabilidad. La rutina debe poder ejcutarse en distintos sistemas informáticos con los mismos resultados.
6. Eficiencia. Cuanto más rápida sea la ejecución de la rutina mejor.

Tipos de generadores pseudo aleatorios

En base al algoritmo utilizado, la mayoría de los generadores de número pseudo-aleatorios pueden clasificarse en alguno de los siguiente tipos:

- Generadores congruentes lineales. Este tipo de generadores producen números aleatorios mediante una fórmula recursiva del tipo:

$$x_{i+1} = (ax_i + c) \texttt{ mod } m \tag{10}$$

  Se parte de un valor inicial x_0 denominado semilla. a es el elemento multiplicador, c es el incremento y m el elemento módulo.

- Generadores congruentes multiplicativos. Estos son un caso concreto de los anteriores en los que no existe incremento, es decir $c = 0$

$$x_{i+1} = (ax_i) \tag{11}$$

- Generadores de Fibonacci Retardados (Lagged Fibonacci Conguential Generators). Este tipo de generadores son una generalización de la secuencia de Fibonacci, en la cual $x_{i+2} = x_{i+1} + x_i$ y que obedecen la fórmula:

$$x_i = (x_{i-p} + x_{i-q}) \tag{12}$$

- Desplazamiento de registros con retroalimentación lineal. (Feedback Shift Register). Con una relación de recurrencia genérica se pueden generar digitos binarios, 0,1 segun:

$$b_i = a_1 b_{i-1} + a_2 b_{i-2} + \ldots + a_n b_{i-n} \quad \texttt{mod } 2 \tag{13}$$

  Para un registro de n bits, se puede conseguir una secuencia máxima de $2^n - 1$. Para conseguir esta secuencia máxima es necesario que el polinomio de realimentación sea un polinomio primitivo, es decir, que sea un polinomio irreducible.

3.3. Técnicas de Integración Monte Carlo

Clásicamente, la integración numérica o mejor dicho el cálculo numérico de integrales se ha venido realizando mediante técnicas de cuadratura numérica. Existen básicamente tres categorías relativas a estos métodos que son:

i Integración de tipo Newton

ii Integración Gaussiana

iii Métodos de Extrapolación

Estos métodos de cuadratura no son muy eficientes cuando se trata de calcular integrales en varias dimensiones, y es aquí donde los métodos de Monte Carlo se aplican con muy buen resultado.

Casi todas las técnicas de integración parten de una densidad muestreadora tal como $P(x) = \frac{P^*(x)}{Z}$ que resulta demasiado compleja para muestrearla directamente, por tanto lo que hace es buscar una densidad más sencilla, sea $Q(x)$ que si se puede muestrear con cierta facilidad.

En función del nivel de similitud entre $Q(x)$ y la densidad real, se utilizará uno u otro método de integración. Cuando la densidad $Q(x)$ sea similar a la densidad real $P(x)$, se utiliza el método de Monte Carlos con muestreo por importancia o con muestreo por rechazo, pero cuando la densidad de distribución de la muestra tan altamente compleja que no se pueden encontrar una densidad $Q(x)$ similar, es necesario hacer uso de otras estrategias, como el método de Metrópolis-Hastings. El muestreo de Gibbs es otra técnica para muestras del al menos dos dimensiones.

3.3.1. Integración Monte Carlo Directa

El método Monte Carlo permite especialmente realizar la integración de funciones multidimensionales de forma más rápida y precisa que mediante los métodos habituales.

$$G = \int_\Omega dx g(x) f(x) \tag{14}$$

N variables aleatorias $X_1, \dots X_N$

$$G_N = \frac{1}{N} \sum_i g(X_i) \tag{15}$$

G_N es un estimador de G

$$E[G_N] = G \tag{16}$$

Aplicando la ley de los grandes números expuesta anteriormente en (1)

$$P\left(\lim_{N \to \infty} G_N = G \right) = 1$$

Sin embargo aunque el método Monte Carlo es el más adecuado para la resolución de integrales en varias dimensiones, el error que introduce depende del número de tests que se realicen en la forma $1/\sqrt{N}$, lo cual supone una convergencia muy lenta en la resolución de algunos problemas. Por esa razón, se aplican algunas estrategias para disminuir este error y que podriamos clasificar, a grades rasgos de la siguiente manera:

- Técnicas de reducción de la varianza
  - Estratificación de Muestras
  - Muestreo por Importancia
- Métodos Adaptativos

3.3.2. Muestras Estratificadas

Este es un método eficiente para la reducción de la varianza. La técnica consiste en dividir todo el espacio de integración en subespacios para realizar una integración Monte Carlo en cada uno de los subespacios e ir sumando los resultados parciales.

Matemáticamente este método esta basado en la propiedad fundamental de la integral de Riemann:

$$\int_0^1 f(x)dx = \int_0^a f(x)dx + \int_a^1 f(x)dx, \qquad 0 < a < 1 \tag{17}$$

Al dividir el espacio total M en k subespacios M_j, con $j = 1, \dots, k$ se toman ahora N_j puntos en cada subspacios M_j, y de esta forma se puede expresar la varianza total, en función de las varianzas generadas por las muestras en cada subespacio:

$$\frac{\sigma^2}{N} = \sum_j \frac{\sigma_j^2}{N_j} \tag{18}$$

Una elección adecuada del particionamiento en subespacios y del número de puntos de muestra en cada uno de ellos puede conducir a una reducción de la varianza σ^2, sin embargo, también puede

ocurrir que una elección no correcta pueda generar una varianza todavía mayor.

Supongamos pues, que se divide el dominio de integración en k regiones, con varianzas $\sigma_1^2, \sigma_2^2, \ldots, \sigma_k^2$, y en estas regiones se toman $N_1, N_2, \ldots, N_k$ muestras respectivamente. Tendremos que:

$$\frac{\sigma^2}{N} = \sum_{j=1}^{j=k} \frac{\sigma_j}{N_j} \tag{19}$$

$$N = \sum_{j=1}^{j=k} N_j \tag{20}$$

Podemos inferir que:

$$\frac{N_j}{N} = \frac{\sigma_j}{\sum_{j=1}^{k} \sigma_j} \tag{21}$$

De forma general se puede decir que la varianza se minimiza cuando el número de tests que se realizan en cada subespacio M_j es proporcional a σ_j.

3.3.3. Muestreo por Importancia

Este método es también un método que permite reducir la varianza en el caso de integrandos singulares. Matemáticamente consiste en un cambio de las variables de integración.

$$\int f(x)dx = \int \frac{f(x)}{p(x)} p(x)dx = \int \frac{f(x)}{p(x)} dP(x) \tag{22}$$

siendo $p(x) = \frac{\partial^d}{\partial x_1, \ldots \partial x_d} P(x)$

Si se considera que $p(x)$ solo pueden ser funciones positivas y normalizadas a la unidad, entonces se puede interpretar $p(x)$ como una función de densidad de probabilidad, y si se disponen de variables aleatorias para la distribución $P(x)$, entonces se podrá calcular la integral a partir de una muestras de números aleatorios $x_1, \ldots, x_N$ distribuidos según $P(x)$.

$$\int p(x)dx = 1 \tag{23}$$

$$E = \frac{1}{N} \sum_{n=1}^{N} \frac{f(x_n)}{p(x_n)} \tag{24}$$

El error de está integración vendrá dado por:

$$\frac{\sigma(f/p)}{\sqrt{N}} \tag{25}$$

y el estimador que se puede calcular para la varianza $\sigma^2(\frac{f}{p}$ viene dado por:

$$S^2\left(\frac{f}{p}\right) = \frac{1}{N}\sum_{n=1}^{N}\left(\frac{f(x_n)}{p(x_n)}\right)^2 - E^2 \tag{26}$$

(NOTA: Incluir recuadro a lapiz de la pag. 14)

3.3.4. Métodos Adaptativos

Los métodos adaptativos se aplican cuando no se conoce de antemano la función a integrar y, utilizando estos métodos se diseña un algoritmo que aprende en con cada iteración hasta dar con el integrando.

Uno de los métodos adaptativos más utilizado en física de altas energias es VEGAS. Este método combina iterativamente el muestreo por importancia y el muestreo estratificado, de esta forma, se acumulan más puntos en aquellas regiones que más contribuyen a la integral.

El procedimiento de trabajo consiste en dividir el dominio de integración en varios subdominios, formando asi un grid sobre el que se realizan las integraciones de forma individual sobre cada retículo. Una vez hecha la primera tanda de integraciones sobre todos los retículos que componen el grid, este se reajusta para dar más peso aquellos retículos que tienen una contribución más notable.

En cada una de las iteraciones se utiliza la técnica de muestreo por importancia, ya que se parte de una función de densidad de probabilidad del tipo:

$$P(x) = \frac{|f(x)|}{\int |f(x)|dx} \tag{27}$$

que se va ajustando poco a poco hasta que se obtiene el grid más óptimo.

4. El Lenguaje R

R es un lenguaje para la realización de cálculos estadísticos. Es software libre, está disponible para varias plataformas y existe abundante documentación en internet. En el apartado de bibliografia se indican algunos links y referencias de utilidad para aprender todo lo necesario sobre el lenguaje R.

La página oficial del lenguaje R es el sitio http:
www.r-project.org , donde se encuentra el repositorio de R, denominado CRAN (Comprehensive R Archive Network)

4.1. Instalación

Para Descargar e instalar el software de R, accedemos a la página del proyecto y entramos en CRAN. En muchas distribuiciones Linux, R forma ya parte de la propia distribución por lo que no es necesario realizar instalación alguna.

4.1.1. Instalación en Linux Debian

El lenguaje R se encuentra en los repositorios de Debian desde 1997 y dispone de una gran número de paquetes descargables e instalables mediante la herramienta de instalación de Debin apt-get. Para ver todos los paquetes de R que se encuentran disponibles en el repositorio al que tiene acceso nuestra distribución de Debian, ejecutar el siguiente comando:

```
#> apt-cache search ^r-.*
```

Para comenzar, realizamos una instalación base, para lo cual solo hay que instalar el paquete `r-base`. Desde una cuenta de usuario autorizado para instalar (root), ejecutamos el comando:

```
#> apt-get install r-base
```

Este paquete tiene bastantes dependencias, por tanto, el sistema nos solicitará autorización para la instalación de las mismas. Decimos que si.

```
Leyendo lista de paquetes... Hecho
Creando árbol de dependencias
Leyendo la información de estado... Hecho
Se instalarán los siguientes paquetes extras:
  build-essential cdbs dbus dpkg-dev fakeroot fontconfig g++ g++-4.7
   ...
Paquetes sugeridos:
  devscripts dbus-x11 debian-keyring g++-multilib g++-4.7-multilib
  ...
Se instalarán los siguientes paquetes NUEVOS:
  build-essential cdbs dbus dpkg-dev fakeroot
  ...

0 actualizados, 144 se instalarán, 0 para eliminar y 0 no actualizados.
Necesito descargar 104 MB de archivos.
Se utilizarán 269 MB de espacio de disco adicional después de esta operación.
¿Desea continuar [S/n]?
```

Una vez finalizado el proceso ya podemos acceder al entorno de trabajo de R. Tecleando R en la línea de comandos del sistema operativo entramos a la línea de comandos de R que debe presentar el siguiente aspecto:

```
R version 2.15.1 (2012-06-22) -- "Roasted Marshmallows"
Copyright (C) 2012 The R Foundation for Statistical Computing
ISBN 3-900051-07-0
Platform: x86_64-pc-linux-gnu (64-bit)

R es un software libre y viene sin GARANTIA ALGUNA.
Usted puede redistribuirlo bajo ciertas circunstancias.
Escriba 'license()' o 'licence()' para detalles de distribucion.

R es un proyecto colaborativo con muchos contribuyentes.
Escriba 'contributors()' para obtener más información y
'citation()' para saber cómo citar R o paquetes de R en publicaciones.

Escriba 'demo()' para demostraciones, 'help()' para el sistema on-line de ayuda,
o 'help.start()' para abrir el sistema de ayuda HTML con su navegador.
Escriba 'q()' para salir de R.
>
```

4.1.2. Instalación en Windows

En el caso de que trabajemos en un entorno windows, sera necesario descargar el software y realizar de la pagína oficial del proyecto R o de alguno de sus mirrors. Para ello efectuamos los siguientes pasos

1. Accedemos al sitio web del proyecto R: wwww.r-project.org y entramos en CRAN.
2. Elegimos un servidor desde donde descargar el software.
3. Seleccionamos `Download R for Windows` como Sistema Operativo,
4. Seleccionarmos el programa base.

5. Realizamos la descarga.

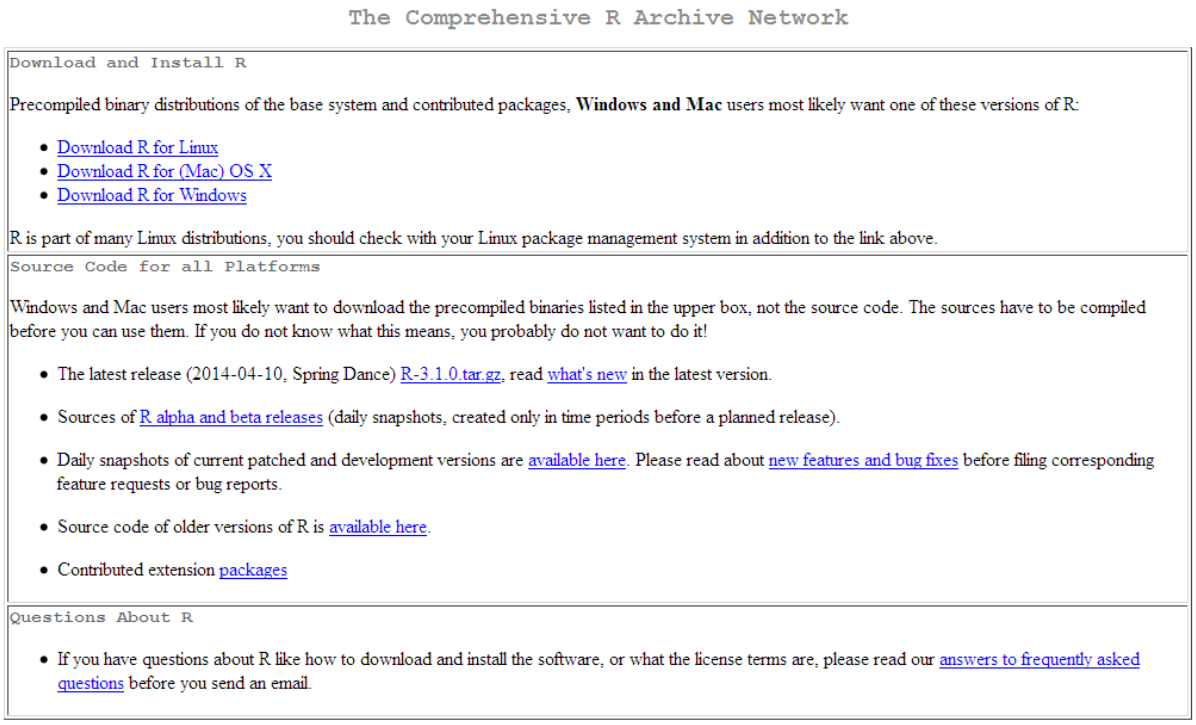

Figura 4.1 *Sitio Web del proyecto R.*

El fichero de descarga es un programa ejecutable que realiza la instalación de R. El ejecutable tiene el formato `R-x.x.x-win.exe` . Para una instalación rápida podemos aceptar las opciones por defecto, pero si hacemos esto, el programa instalará un versión de R de 32 bits y otra versión de 64 bit. Si no sabemos la arquitectura del sistema podemos dejar que instale ambas, pero es conveniente instalar aquella que coincida con nuestra arquitectura de 32 o 64 bit (actualmente casi todas las estaciones de trabajo son de 64bits).

Una vez finalizada la isntalación accedemos al menu principal de Windows, buscamos el producto instalado y hacemos click sobre él. De esta forma accedemos a la consola de R en Windows, que tiene el siguiente aspecto:

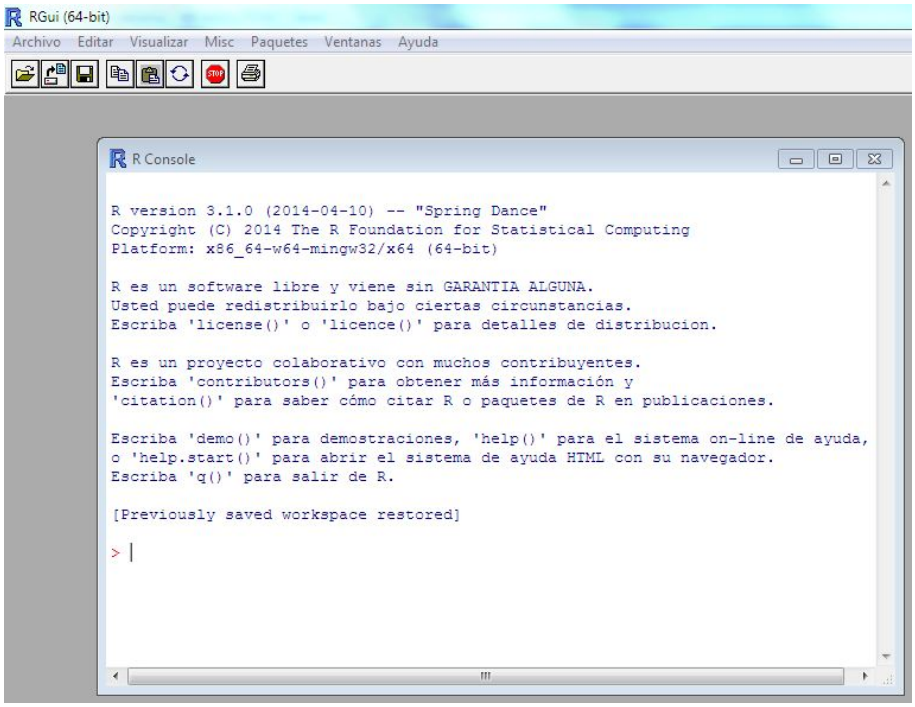

Figura 4.2 *Aspecto de la consola R.*

R es un lenguaje modular que admite varios complementos. Uno de estos complementos es la interfaz gráfica R-Studio. Si bien se puede trabajar directamente en la consola de R sin necesidad de interfaz gráfica, es recomendable utilizar R-Studio por la facilidad de uso y la eficiencia que proporciona a la hora de trabajar.

Con el módulo base, se instalan una serie de paquetes básicos para la realización de cálculos y representación de resultados. Sin embargo, R es un lenguaje modular que permite la adición de nuevos paquetes con funcionalidades específicas. A medida que aparecen nuevos paquetes adicionales, se suben al CRAN, donde son revisados y mejorados por los autores y los usuarios.

4.1.3. Instalación en otras plataformas

El entorno R esta disponbile para una gran variedad de plataformas. Si la estación de trabajo es un Apple o cualquier otro Linux o distribución de Unix, en la página del proyecto R se encuentran los binarios para todas estas plataforma. El procedimiento de instalación de los binarios, como se ha en el punto anterior, es muy sencillo, y en cualquier caso, en la página oficial vienen las indicaciones a seguir paso a paso para la instalación. El proyecto R, es un Open Source, lo cual significa que también esta disponible al público el código fuente del programa, lo que permite su modificación, adaptación y compilación para casi cualquier plataforma.

4.1.4. Instalación del entorno gráfico

El entorno que proporciona R, es un entorno de línea de comandos, muy potente para la realización de cálculos, pero carente de algunas funcionalidades muy cómodas en el desarrollo normal de una sesión de trabajo.

Existen en internet varios interfaces gráficos para R, pero el más popular es el denominado RStudio, que está disponible en la web de sus creadores http:
www.rstudio.com. Es también Open Source, por tanto no hay que preocupase de la gestión de licencias para su uso. En caso de necesitar soporte técnico oficial, existen versiones comerciales y licencias para educación que incluyen soporte técnico 24 horas.

En el sitio web de Rstudio se pueden encontrar los ejecutables del programa para varias plataformas, así como el código fuente compilable. Para una instalación rápida, se recomienda descargar directamente el fichero ejecutable. El proceso de instalación es muy sencillo y se obvia su explicación. Una vez instalado, RStudio presenta un entorno gráfico como se muestra a continuación:

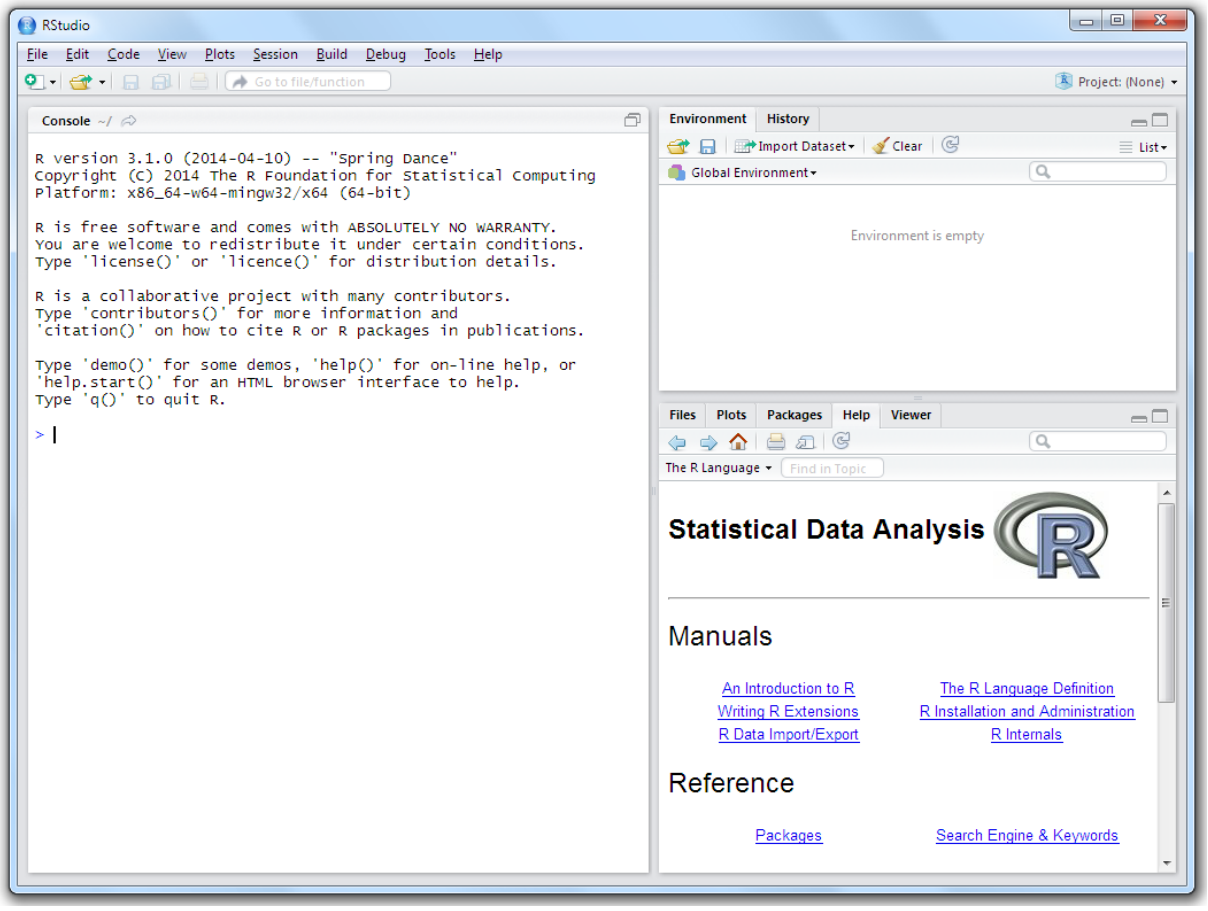

Figura 4.3 *Aspecto de la consola gráfica RStudio.*

4.1.5. Instalación de Paquetes

El entorno R viene con una serie de funciones básicas incluidas en la librería base. Sin embargo existen multitud de funciones adicionales empaquetadas en librerías o paquetes R que no se incluyen en la instalación por defecto. Para instalar un paquete o libreria que no está incluido en el módulo básico hay que ejecutar el comando:

```
# Instalación del paquete
install.packages("Nombre_del_Paquete")
```

Desde la interfaz gráfica R-Studio se pueden instalar paquetes. Para ello acceder al menú Tools/install packages y teclear el nombre el paquete a instalar.

Una vez que el paquete está instalado en el sistema, cada vez que se quiera hacer uso de él hay que cargalo en el área de trabajo. Para ello se utiliza la función `library`. Ejemplo:

```
# Carga del paquete en la sessión
library("Nombre_del_Paquete")
```

La funcionalidad más básica de R es la de calculadora. Podemos realizar operaciones básicas de la misma forma que en una calculadora electrónica. R permite la ejecución de un gran número de funciones a las que pasando un valor nos devuelven el resultado de la función para ese valor. También se puede asignar valores a variables y almacenarlas para su uso posterior. Sin embargo, R es un lenguaje orientado a objetos que está diseñado principalmente para la realización de cálculos de probabilidad y estadística, por lo que incluye sobre todo funciones mayormente orientadas a este uso, tal y como se irá viendo a lo largo del documento.

4.2. La Ayuda

El entorno R proporciona varias funciones de ayuda:

- `help.search()`
- `RSiteSearch()`
- `help()`
- `example()`

4.3. Configuración del Área de Trabajo

Cuando se inicia un sesión en R, siempre debe existir un área de trabajo. El área de trabajo es el directorio en el que se almacenan todos los datos y los archivos que se crean durante una sesión. Si esta área de trabajo no se define, entonces el sistema asigna una por defecto, que suele ser:

Para definir un área de trabajo se utiliza el comado `setwd` (set working directory), ejemplo:

```
setwd(/Ruta/del/directorio)
```

Si trabajamos con RStudio, entonces también se puede establecer desde la consola gráfica: session → Set Working Directory → Chose directory

Cuando estemos trabajando y queramos saber cual es el área de área de trabajo actual, se puede ejecutar el comando `getwd()` (get working directory)

```
getwd()
```

R permite también guardar imágenes del área de trabajo. Esto son ficheros que contienen todo el trabajo realizado hasta ese momento en una sesión. Estas imágenes se guardan en ficheros con extensión RData y pueden cargarse en cualquier momento para recuperar una sesión de trabajo anterior. Ejemplo:

```
Save.image(NombreArchivo.RData)  Load(NombreArchivo.RData)
```

4.4. Datasets

Los datasets son conjuntos de datos que normalmente tienen un formato rectangular de filas y columnas, donde cada fila representa una observación y cada columna un dato relativo a las observaciones. En todos los sentidos podemos afirmar que un dataset es una estructura similar a lo normalmente denominamos tabla.

R dispone de varias estructuras de almacenamiento de datos. Hay que diferenciar entre lo son las estructuras de datos y los tipos de datos. Las estructuras de datos almacenan datos que pueden ser de diversos tipos, así pues, los tipos de datos que soporta R son:

- Integer. Almacenamiento de números enteros.
- numeric. Datos de tipos númerico, tanto enteros como double.
- complex. Almacenamiento de números complejos.
- character. Permite el almacenamiento de cadenas de caracteres.
- logical. Solo permite los valores lógicos de verdadero (T) o falso (F).

Estos datos se almacenan en estructuras de datos que pueden ser:

- **Factores.** Variables que pueden ser nominales u ordinales

- **Vectores.** Los vectores son estructuras unidimensionales que pueden contener datos numéricos, de carácter o lógicos.
- **Matrices.** Una matriz es una estructura de dos dimensiones donde todos los elementos son del mismo modo, es decir, numéricos, logicos o caracteres.
- **Arrays.** Los arrays son estructuras similares a las matrices, pero pueden tener más de dos dimensiones.
- **Dataframes.** Un dataframe es una estructura de dos dimensiones pero cuyo contenido puede ser más amplio que el de una matriz, ya que en un dataframe se pueden almacenar distintos tipos de datos en distintas columnas.
- **Listas.** Las listas son colecciones ordenadas de objetos.
- Functions. Objetos que permiten extender la funcionalidad de R. Se almacenan en el espacio de trabajo y admiten la codificación de estructuras cíclicas y condicionales.

Además de las funciones mátemáticas, R proporciona algunas funciones propias para el manejo de los objetos:

- `str(NombreOjeto)` . Muestra la estructura del objeto que se le pasa entre paréntesis.

4.4.1. Trabajando con vectores. Ejemplos

Creación y operación de vectores

La creación de vectores re realiza con la función `c()` . A continuación se muestran algunos ejemplos:

```
v <- c(1,2,3,4,5)
w <- c("uno", "dos", "tres", "cuatro", "cinco")
z <- c(TRUE, TRUE, TRUE, FALSE, TRUE, FALSE)
```

Los datos almacenados en un vector han de ser obligatoriamente todos del mismo tipo, en el ejemplo anterior, `a` es un vector numérico, `b` es un vector de caracteres y `c` es un vector de tipos lógicos. Los vectores se pueden sumar, restar o multiplicar por un número, ejemplo:

- Multiplicación de un vector por un escalar:

  ```
  5 * v
  ```

- Suma de vectores:

  ```
  v + w
  ```

- Resta de vectores

`5 * v` multiplica el vector `v` por 5 `v + w` suma los vectores `v w`

Funciones para la manipulación de vectores

Además de las estás operacioens, R proporciona una serie de funciones que facilitan la manipulación y el trabajo con vectores.

- `length(v)` Calcula la longitud del vector.
- `sum(v)` Realiza la suma de todos los componentes del vector.
- `min(v)` Obtiene el componente del vector con el mínimo valor.
- `mode(v)` Devuelve el tipo de vector, es decir, númeric, caracter, etc.
- `sort(v)` Ordena los componentes del vector.

Los vectores en R pueden ser numéricos si solo contienen números o alfanuméricos si contienen algo más que números. Los caracteres en un vector deben ir entre comillas.

4.4.2. Trabajando con matrices. Ejemplos

Creación y configuración de matrices

Las matrices se crean con la función `matrix`. Es necesario indicar al menos el número de filas y de columnas.

```
mimatriz <- matrix(vector, nrow=3, ncol=2, byrow=TRUE
```

4.4.3. Trabajando con arrays. Ejemplos

Como se ha comentado anteriormente, los arrays son matrices n-dimensionales. La función que se utiliza para la creación de arrays es: `array`.

4.4.4. Trabajando con dataframes. Ejemplos

Los dataframes pueden contener arreglos de datos n-dimensionales, pero a diferencia de las matrices, no es necesario que todos los datos sean del mismo tipo. La forma de crear una dataframe es mediante la función `data.frame`. Es necesario especificar las columnas.

La forma más usual de crear dataframes es mediante la agregaciòn de vectores. Todos los elementos de un vector deben ser del mismo tipo. Pero en vectores distintos los tipos pueden ser distintos.

A modo de ejemplo vamos a crear la ficha de los parámetros meteorológicos de una semana.

```
> dia <- c("lunes","martes","miércoles","Jueves","Viernes","Sabado","Domingo")
> temperatura <- c(25, 31, 29, 28, 28, 26, 29)
> humedad <- c(70, 80, 82, 83, 77, 79, 81)
> lluvia <-c(TRUE, FALSE, FALSE, FALSE, FALSE, TRUE, TRUE)
> datosMeteo <- data.frame(dia, temperatura, humedad, lluvia)
> datosMeteo
        dia temperatura humedad lluvia
1     lunes          25      70   TRUE
2    martes          31      80  FALSE
3 miércoles          29      82  FALSE
4    Jueves          28      83  FALSE
5   Viernes          28      77  FALSE
6    Sabado          26      79   TRUE
7   Domingo          29      81   TRUE
```

Nótese que R añade un siempre añadirá un primer campo que numera la cantidad de registros introducidos.Una vez creado el dataframe se puede acceder selectivamente a sus componentes mediante el operador `[]`. El formato para realizar esta consulta es: scriptsize `dataframe[c(çol1",çol2", ... )]`

Ejemplo: Obtener tan solo la temperatura y la humedad de cada día.

```
> datosMeteo[c("temperatura", "humedad")]
  temperatura humedad
1          25      70
2          31      80
3          29      82
4          28      83
5          28      77
6          26      79
7          29      81
```

Manipulación de dataframes con attach, detach y with

Los dataframes vectores que componen el dataframe pueden definirse en el mismo momento de creación del dataframe, en ese caso, cuando se quiera acceder a alguno de ellos directamente, sera necesario referenciar el dataframe en el que fueron creados.

Para acceder a la variable hay que utilizar el operador `$`. Para tener que evitar cualificar los vectores cada vez que queremos acceder a ellos se utilizan las funciones `attach`, `detach` y `with`.

4.4.5. Trabajando con listas. Ejemplos

Una lista, es una colección de objetos que previamente han sido creados en R. La función para crear listas es `list()` . Dentro de una lista se pueden tener objetos de distinta naturaleza y sin relación alguna entre ellos. Una lista podría utilizarse para crear un documento que tuviese un título, un texto y una salida de datos relacionados.

4.5. Distribuciones de Probabilidad

El lenguaje R esta equipado con un gran número de funciones para el cálculo y la representación de distribuciones de probabilidad. Todas las distribuciones de probabilidad de uso habitual están contempladas en R. Los nombres de las distribuciones tienen la nomenclatura siguiente:

```
[dpqr]nombreDistribucion(parsObl[parsOps]
```

El primer caracter es una de las cuatro letras siguientes, el cual indica un aspecto concreto de la distribución:

d = Función densidad. Ejemplo: dnorm

p = Función de distribución

q = Funcion de quantiles

r = Generación aleatoria de variables.

A continuación viene el nombre de la función estadística, con sus parámetros. Cada función estadística puede tener su propia variedad de parámetros obligatorios y opcionales. Las funciones estadísticas que tenemos son:

- Distribución Beta. `dbeta(x, ...), pbeta(q, ...), qbeta(p, ...), rbeta(n, ...)`
- Distribución Binomial `dbinom(x, ...),pbinom(q, ...), qbinom(p, ...), rbinom(n, ...)`
- Distribución de Cauchy `dcauchy(x, ...), pcauchy(q, ...), qcauchy(p, ...), rcauchy(n, ...)`
- Distribución χ^2 `dchisq(x, ...), pchisq(q, ...). qchisq(p, ...), rchisq(n, ...)`
- Distribución Exponencial `dexp(x, ...), pexp(q, ...), qexp(p, ...), rexp(n, ...)`
- Distribución F `df(x, ...), pf(q, ...), qf(p, ...), rf(n, ...)`
- Distribución Gamma `dgamma(x, ...), pgamma(q, ...), qgamma(p, ...), rgamma(n, ...)`
- Distribución Geométrica `dgeom(x, ...), pgeom(q, ...), qgeom(p, ...), rgeom(n, ...)`
- Distribución Hypergeométrica `dhyper(x, ...), phyper(q, ...), qhyper(p, ...), rhyper(nn, ...)`
- Distribución Log-Normal `dlnorm(x, ...), plnorm(q, ...), qlnorm(p, ...), rlnorm(n, ...)`
- Distribución Logística `dlogis(x, ...), plogis(q, ...), qlogis(p, ...), rlogis(n, ...)`
- Distribución Normal `dnorm(x, ...), pnorm(q, ...), qnorm(p, ...), rnorm(n, ...)`
- Distribución de Poisson `dpois(x, ...), ppois(q, ...), qpois(p, ...), rpois(n, ...)`

- Distribución t de Student `dt(x, ...), pt(q, ...), qt(p, ...), rt(n, ...)`
- Distribución Uniforme `dunif(x, ...), punif(q, ...), qunif(p, ...), runif(n, ...)`
- Distribución de Weibull `dweibull(x, ...), pweibull(q, ...),qweibull(p, ...), rweibull(n, ...)`

Ejemplo. Obtener una distribución normal en el intervalo [-3,3]

```
x <- pretty(c(-3,3), 30)
y <- dnorm(x)
plot(x, y, type = "l", xlab = "Desviación Normal", ylab = "Densidad", yaxs =i")
```

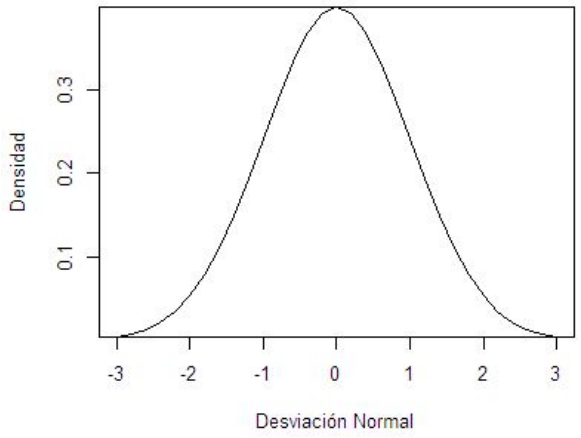

Figura 4.4 *Runción dnorm()*

Ejemplo. Obtener una distribución normal en el intervalo [-3,3]

```
x <- pretty(c(-3,3), 30)
y <- dnorm(x)
plot(x, y, type = "l", xlab = "Distribución Exponencial", ylab = "Densidad", yaxs =i")
```

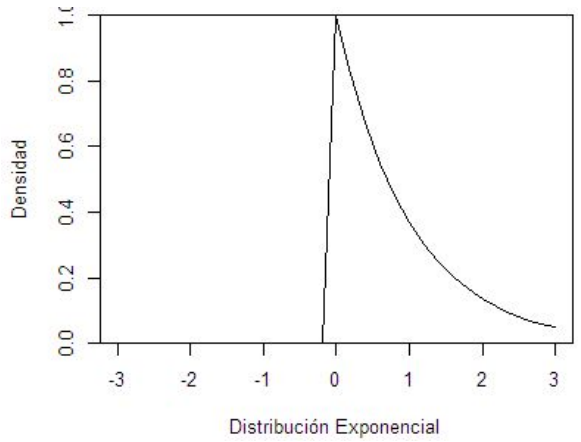

Figura 4.5 *Runción dexp()*

4.6. Lectura de ficheros

El lenguaje R, trabaja con datos en memoria, esta es una de las limitaciones que se achaca al sistema cuando se trata con grandes volúmenes de datos. Sin embargo, R posee mecanismos para trabajar con datos almacenados en soportes externos, sea por ejemplo ficheros de distintos tipos, hojas de cálculo o incluso bases de datos relacionales.

A continuación se muestra como R puede acceder a la información localizada en distintos tipos de almacenamiento externo.

4.6.1. Leer un fichero de texto

Para trabajar con datos almacenaos en archivos, es necesario leerlos y cargarlos en una variable de R. Para leer un fichero con formato `.txt` se utiliza la función `read.table`.

Ejemplo: Leer un fichero con datos de texto `datos.txt` y cargarlo en la variable de trabajo `datos`.

```
datos <- read.table("directorio/datos.txt", header=TRUE)
```

El argumento `header` que aparecen en la función sirve para indicar si se cargan también los nombres de las columnas.

Una vez que el fichero ha sido cargado en el dataframe `datos` , ya se puede acceder a la información que contiene y utilizar las funciones pertinentes para manejarlo.

Las funciones disponibles para trabajar con dataframes son:

- `attach()`. Carga directamente la variable en memoria indexada con el dataframe al que pertenece, con lo que se puede referenciar la variable sin incluir el dataset al que pertenece.
- `detach()`. Vuelve a requerir que se referencie al dataset al que pertenece cada variable.
- `dim()`. Dimensión del dataframe
- `length()`. Longitud del dataframe.
- `table()`. Muestra el dataframe en formato tabla.
- `str()`. Muestra la estructura del dataframe.

4.6.2. Leer un fichero SPSS

SPSS es un popular programa de cálculo estadístico que actualmente produce y comercializa IBM. Desde R se puede acceder a ficheros de SPSS y de otros muchos programas como iremos viendo más adelante. La librería que permite a R el acceso a ficheros de programas externos se denomina `foreign` . Esta librería viene incluida en un paquete con el mismo nombre que no instala con la distribución base, por lo que es necesario instalar el paquete y después cargar manualmente la librería.

Ejemplo: Instalar el paquete `foreign` y la librería del mismo nombre `foreign` que permite el acceso a ficheros SPSS.

```
# Instalación del paquete
install.packages("foreign")
# Carga de la librería
library(foreign)
datos <- read.spss(file=?datossps.sav", to.data.frame=TRUE)
str(data)
```

4.6.3. Leer una hoja de cálculo excel

El acceso a las hojas de cálculo excel con extensión `xls` se realiza mediante las funciones incluidas en la libreria `xlsx` . Ejemplo: Cargar en R los datos almacenados en una hoja de cálculo excel con formato xls.

```
# Cargar la librería
library(xlsx)
datos <- read.xlsx(?datossps.xls", sheetIndex=1)
str(data)
```

Nota: La libreria `xlsx` exige tener instalada la máquina virtual de Java en la estación de trabajo.

Una forma más sencilla de acceder a datos almacenados en hojas de calculo excel, es diponerlos en formas CSV. El propio programa excel permite la exportación a este formato. La función que realiza la lectura de este formato es `read.csv` que viene con el paquete `utils` instalado por defecto con el modulo base.

4.6.4. Acceso a Bases de Datos Relacionales

Mucha de la información se haya en Bases de Datos Relacionales y R dispone de mecanismos acceder a la información que se encuentra en los gestores de bases de datos más populares tales como MySQL, ORACLE, SQLServer, etc. Para aquellos gestores de bases de datos, tales como IBM DB2, para los que no existe en R un conector específico, siempre es posible acceder mediante el conector de acceso a datos ODBC.

4.7. Programación en R

Tal y como se viene mencionando, R es un lenguaje de programación y como tal, permite la creación de programas con toda su lógica de ejecución y control y todos los elementos y objetos pertinentes.

El primer paso para programar en R, es aprender a escribir funciones. Algunos ejemplos de funciones familiares de uso muy común son `mean` o `var`. Estas funciones están codificadas en R, aunque por motivos de rendimiento es posible crear funciones en lenguajes compilados, tales como C o C++, esto nos dá la posibilidad de reutilizar programas en C/C++ que ya se tengan en uso, e incorporarlos en R.

4.7.1. Primera función en R

La creación de una función en R se realiza según la siguiente sintaxis:

`NombreDeFunción` <− **function**`( ListaDeParámetros){DefincionDeFunción}`

El comando **function**(es realmente una función built-in de R que lo que hace es crear objetos de la clase **function**(

Ejemplo.

```
g <- function(x) {
+ return(x+1)
+ }
```

Una función siempre devuelve un valor. Una función en R siempre devuelve por defecto, el último valor calculado en la misma. Sin embargo, al igual que en otros lengueajes, se utiliza de forma explícita la función **return**(). La función **return**() en R, puede devolver cualquier objeto, por ejemplo un valor o una función.

Si se desea devolver más de un valor, entonces hay que construir con ellos una lista y devolver la lista.

Las funciones no tienen efectos colaterales, es decir, las variables que se tratan en las funciones permanecen inalteradas fuera del alcance de la función, aunque existen excepciones a esta regla cuando se trata con variables globales.

Sea un variable z que aparecen dentro de una función y que tiene el mismo nombre que una variable definida globalmente. Lo que ocurre en esta situación es que la variable z que aparece dentro de la función se trata como una variable local, pero con un valor inicial que se corresponde con el valor de la variable global del mismo nombre.

Las asignaciones siguientes a la variable local z dentro de la función no cambiarán el valor de la variable global. Se produce una excepción a este comportamiento si se utiliza el operador de superasignación.

Ocurre lo mismo si la variable z es un argumento formal de la función. Su valor inicial será el del argumento, pero los cambios siguientes a la variable no afectarán al valor del argumento.

4.7.2. Operador de Superasignación

Si se quiere escribir sobre variables globales, o mejor dicho, sobre variables que están un nivel por encima de la alcance actual, se utiliza el operador de superasignación <<. Ejemplo:

```
> two <- function(u) {
+ u <<- 2*u
+ y <<- 2*y
+ z <- 2*z
+ }

> x <- 1
> y <- 2
> z <- 3

> two(x)
> x
[1] 1
> y
[1] 4
> z
[1] 3
```

4.7.3. Definición de funciones dentro de funciones

Dado que las funciones son objetos, es completamente lícito definir funciones dentro de funciones. Estas funciones seguirán las reglas de alcance descritas anteriormente.

```
> f <- function(x) {
+ v <- 1
+ g <- function(y) return((u+v+y)?2)
+ gu <- g(u)
+ print(gu)
+ }
> u <- 6
> f()
[1] 169
```

Aquí la variable u sería visible dentro del cuerpo de f(), inclusive en la construcción de g(). La variable v, sería un variable local para f() y al mismo tiempo una variable global para g(), a que se

definió por última vez en f().

Si g() solo va a ser utilizado dentro de f(), entonces su ubicación en f() es consistente con el principio de encapsulación. Para no complicar el código en exceso se recomienda que las funciones encapsuladas sean de pocas líneas.

4.7.4. Creación de operadores binarios

En R se pueden crear operadores binarios de la siguiente forma: Se crea una función que comience y termine con %, se definen dos argumentos del mismo tipo y un valor de retorno. Ejemplo:

```
> "\%a2b\%" <- function(a,b) return(a+2*b)
> 3 \%a2b\% 5
[1] 13
```

4.7.5. Edición de funciones

Una consecuencia de que las funciones en R sean objetos es que pueden editarse dentro del modo interactivo. Para realizar pequeños cambios en una función se puede utilizar la función edit(). Ejemplo, para cambiar la función f1() se puede teclear:

```
> f1 <- edit(f1)
```

que abrirá el editor por defecto y permitira realizar modificaciones en el código de la función f1(). El editor que se ejecute dependerá del valor de la variable interna de R editor. En los sistemas de tipo UNIX, R toma el valor de esta variable de la variable del sistema EDITOR o VISUAL, pero se puede modificar manualmente de la siguiente manera:

```
> options(editor="/usr/bin/vim")
```

Cuando se modifica el código de una función, el interprete de R debe recompilar el código, cuando se utiliza la misma función, como el ejemplo anterior, si hay un error se pierde la función original. Si se da esta situación se puede recuperar la función con el comando:

```
> x <- edit()
```

4.7.6. Bucles y lógica de ejecución

Dentro de las funciones se pueden implantar estructuras repetitivas mediante bucles y conducir la lógica de la ejecución mediante la orden condicional `if`.

- Bucles con `for`.

```
> for (nombreVar in expr_1) {expr_2}
```

Siendo:

`nombreVar` : Nombre de la variable de control de la iteración.

`expr1` : Suele ser un vector con los valores a recorrer por el bucle.

`expr2` : Es la expresión que se ejecutará en cada iteración del bucle.

- Condicional con `if`

```
> if (expr1) {expr2} else {expr3}
```

Siendo:

`expr1` : Expresión que produce un valor lógico.

`expr2` : Expresión que se ejecuta si `expr1` produce un valor logico igual a TRUE.

`expr3` : Expresión que se ejecuta si `expr1` produce un valor logico igual a FALSE.

4.8. Representación Gráfica

Con R se pueden representar gráficamente los resultados generados por las distintas operaciones. Las funciones más utilizadas para la representación de gráficos son:

- **`plot.`** Es una función genérica que puede generar una amplia gama de gráficos en función de los argumentos que reciba. Algunos ejemplos del uso de la función `plot`:

  - **`plot(x,y).`** Cuando se pasan dos variables aleatorias como parámetros, la función `plot` muestra un gráfico de dispersión de la variables `y` sobre la variable `x`.

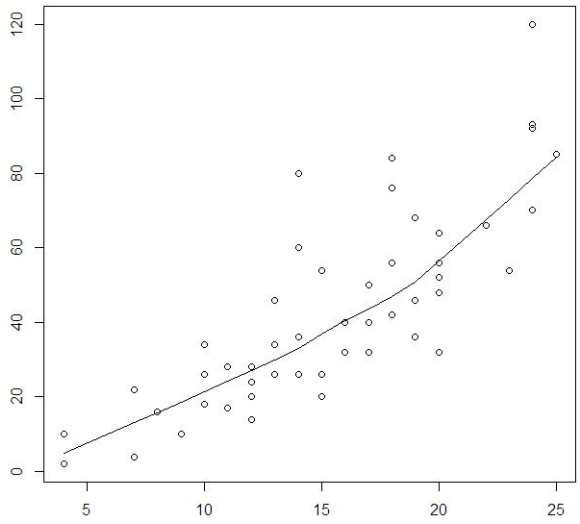

Figura 4.6 *Representación gráfica con la función plot()*

- **`pairs`.** Esta función toma como parámetro una matriz numérica o una hoja de datos, y representa los valores de cada una de las columnas frente a cada una de las otras columnas, produciendo una matriz de gráficos en la que en cada elemento se muestra un par de columnas. Ejemplo de uso:

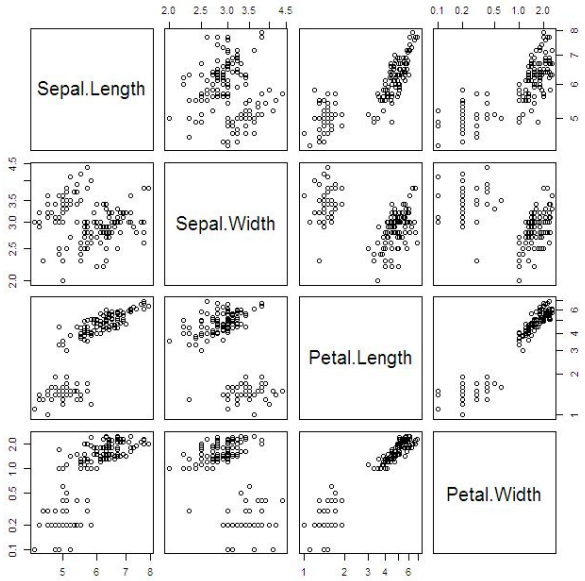

Figura 4.7 *Representación gráfica con la función pairs()*

- **`coplot`.** Esta función es muy útil cuando se trabaja con tres o más variables. Lo que hace es representar las dos primeras para cada valor que tome la tercera, o para cualquier combinación que se pueda hacer del resto de variables.

```
> coplot(a ~ b | c)
> coplot(a ~ b | c + d )
```

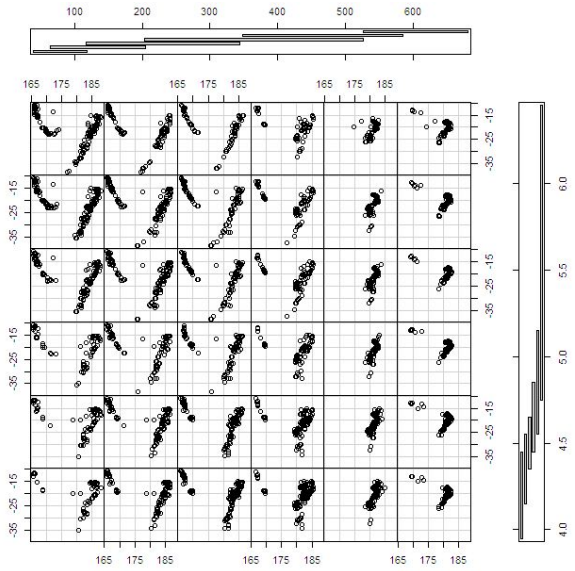

Figura 4.8 *Representación gráfica con la función qqnorm()*

- **qqnorm.** Esta función toma un vector como argumento y lo representa en función de los valores esperados de una distribución normal.

```
> x=c(31,32,33,44,45,36,57,38,56,43,23, 43, 43, 23,29,38)
> qqnorm(x)
```

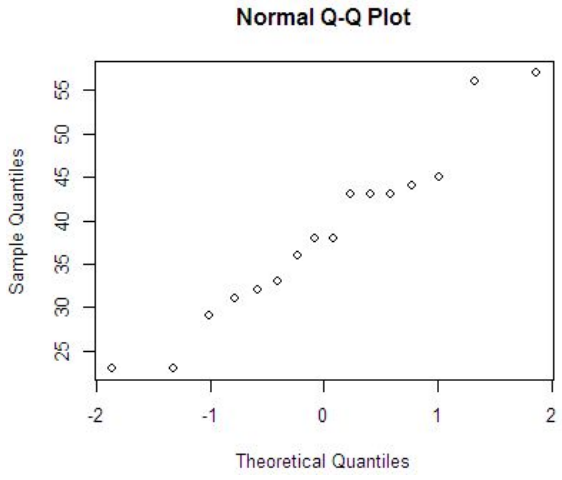

Figura 4.9 *Representación gráfica con la función qqnorm()*

- **hist.** Toma como argumento un vector numérico y produce un histograma del mismo.

```
> x=c(31,32,33,44,45,36,57,38,56,43,23, 43, 43, 23,29,38)
> hist(x)
```

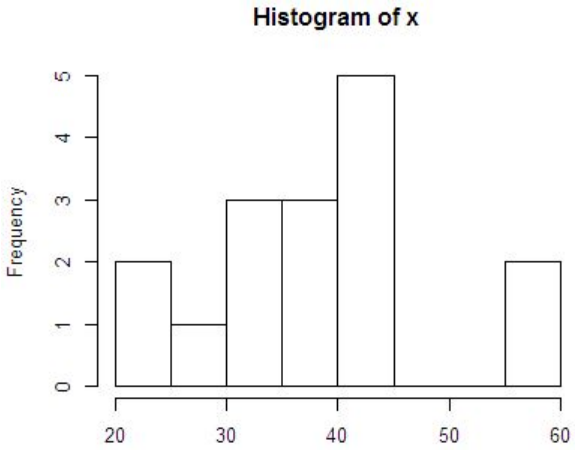

Figura 4.10 *Representación gráfica con la función hist()*

- **persp.** Muestra superficies tridimesionales.

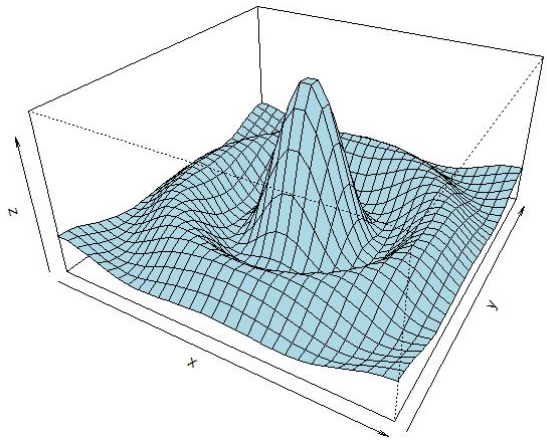

Figura 4.11 *Representación gráfica en 3d de la funión persp(), generada con example(persp)*

Existen un gran número de posibilidades de representación gráfica con R, y cada una de ellas se aplican con más o menos intensidad en distintos tipos de trabajo. Para encontrar los formatos gráficos que más se adecuan al tipo de trabajo a generar, es recomendable consultar la documentación oficial en la página web del proyecto R.

5. Implementación del Método Monte Carlo con R

5.1. Números Aleatorios

A diferencia de otros métodos de cálculo, el método Monte Carlo proporciona una estimación del error absoluto que decrece como $N^{1/2}$, mientras que los otros métodos proporcionan un estimación de error que decrece como $N^{-1/m}$, siendo N y $m \in \mathbb{N}$ el tamaño de la muestra, lo cual proporciona a Monte Carlo una ventaja considerable en lo que se refiere a la eficiencia computacional. Es decir, mientras que la solución exacta a un problema con N elementos suele tener un coste que crece exponencialmente con N, el método Monte Carlo proporciona una estimación de la solución con un error tolerable a un coste que se incrementa como mucho polinómicamente con N.

Como se ha explicado anteriormente, uno de los elementos base de los métodos de Monte Carlo es la generación de variables aleatorias. El lenguaje R proporciona varias funciones incorporadas que permiten la generación de variables pseudo-aleatorias. Todas las distribuciones de probabilidad tienen su función en R para generar datos que se ajusten a la distribución.

- Distribución Uniforme.

```
runif(n, min=0, max=1)
```

Siendo:

`n` : Tamaño de la muestra,

`min` : Valor mínimo,

`max` : Valor máximo.

Ejemplo. Generar una serie de 10 valores aleatorios que sigan un distribución uniforme.

```
> runif(10)
[1] 0.56409675 0.75598845 0.70382940 0.45590208 0.04692889 0.63183683
[7] 0.06478649 0.37299561 0.96142053 0.46595012
```

Si no se especifica ningún parámetro `max`, el rango en el que se generan estos valores es, por defecto el intervalo [0,1]. Los parámetros `min` y `max`, permiten establecer un rango para la generación de valores. Ejemplo, la generación de 10 valores aleatorios en el intervalo [2,5], será:

```
> runif(10, 2,5)
[1] 2.146359 3.947241 2.951471 4.569262 2.548054 2.913454 2.436633 4.554776
[9] 4.836651 3.357484
```

La función `runif()` toma un valor distinto como semilla cada vez que se ejecuta, es por eso que con cada ejecución de obtiene un conjunto distinto de números pseudo-aleatorios. Existe

la posibilidad de establecer una valor fijo para la semilla, lo cual permitirá controlar que el conjunto de valores generados sea siempre el mismo, si esto fuese un requisito para la ejecución de las pruebas.

La forma de establecer el valor de la semilla es mediante la función `set.seed()` . A continuamos se muestra en el siguiente ejemplo, como al establecer un valor fijo para la semilla, se obtienen siempre los mismos resultados.

```
 > set.seed(12345)
 > runif(10, 2,5)
 [1] 2.146359 3.947241 2.951471 4.569262 2.548054 2.913454 2.436633 4.554776
[9] 4.836651 3.357484
```

- Distribución Normal.

```
rnorm(n, mean=0, sd=1)
```

Siendo:

`n` : Tamaño de la muestra,

`mean` : Media de la distribución,

`sd` : Desviación estándar de la distribución.

Ejemplo. Para generar un conjunto de 20 datos pseudo-aleatorios que sigan una distribución normal estándar ($\mu = 0$ y $\sigma = 1$), se utilizará la función `rnorm` de la siguiente manera:

```
> rnorm(20,0,1)
 [1] -0.92303399  1.16613734  1.03068192 -0.80055156  1.41304849 -1.13019577 -1.21067171 -0.67354026
 [9] -1.36346464 -1.08924499 -0.35105368  0.03109505 -2.30775110  0.09383612 -0.58900671 -0.69562592
 [17] -0.71807993 -0.71201950  0.08178194 -0.95160763
```

- Distribución LogNormal.

```
rlnorm(n, meanlog=0, sdlog=1)
```

Siendo:

`n` : Tamaño de la muestra,

`meanlog` : Media de la distribución,

`sdlog` : Desviación estándar de la distribución.

- Distribución Beta.

```
rbeta(n, shape1, shape2, ncp=0)
```

Siendo:

`n` : Tamaño de la muestra,

`shape1, shape2` : Parámetros positivos de la Distribución Beta,

`ncp` : Parámetro de no-centralidad (non-centrality-parameter).

- Distribución Gamma.

```
rgamma(n, shape, rate=1, scale=1/rate)
```

Siendo:

`n` : Tamaño de la muestra,

`rate` : Forma alternativa de especificar la escala,

- Distribución Chi Cuadrado.

```
rchisq(n, df, ncp=0)
```

Siendo:

`n` : Tamaño de la muestra,

`df` : Grados de libertad,

`ncp` : Parámetro de no-centralidad (non-centrality-parameter).

- Distribución Exponencial.

```
rexp(n,rate=1)
```

Siendo:

`n` : Tamaño de la muestra,

`rate` : Vector de proporciones (ratios).

Ejemplo. Generación de un conjunto de 20 datos pseudo-aleatorios para una distribución exponencial sería:

```
> rexp(20)
 [1] 0.26776548 0.61564128 0.94474276 0.23896882 0.73579087 0.99661414 0.38453590 0.37099159 2.12116504
[10] 0.18054439 0.09471638 0.08212750 1.20332298 3.88951107 1.79746329 0.44572107 0.24222064 0.37893140
[19] 2.56470015 1.57873653
```

- Distribución Discreta Finita.

```
sample(x, size, replace=TRUE, prob=q)
```

Siendo:

`n` : Tamaño de la muestra,

`size` : Número de intentos (cero o más),

`replace` : Especifica si se utiliza reemplazo en la extracción de la muestra,

`prob` : Vector de probabilidades.

- Distribución Binomial.

```
rbinom(n, size, prob)
```

Siendo:

`n` : Tamaño de la muestra,

`size` : Número de intentos (cero o más),

`prob` : Probabilidad de éxito de cada intento.

- Distribución Binomial Negativa.

```
rnbinom(n, size, prob, mu)
```

`n` : Tamaño de la muestra,

`size` : Número de intentos (cero o más)

`prob` : Probabilidad de éxito de cada intento

`mu` : Parametrización alternativa utilizando la media.

- Distribución de Poisson.

```
> rpois(n, lambda)
```

Siendo:

`n` : Tamaño de la muestra,

`lambda` : Parámetro λ de la distribucion.

Sin embargo, donde realmente muestran su potencia los métodos de Monte Carlo es en la evaluación de integrales multidimensionales y para ellos los datos pseudo-aleatorios tienen que obedecer a distribuciones en varias dimensiones. Con R se puede disponer de datos de distribuciones multivariantes. En el caso de un distribución normal multivariante con una vector media y una matriz de covarianza, se tiene la función `mvrnorm` que viene en el paquete MASS y cuya definición es la siguiente:

- Distribución Normal Multivariante.

```
> mvrnorm(n, media, sigma)
```

Siendo:

`n` : tamaño de la muestra,

`media` : vector de medias y

`sigma` : matriz de correlación.

Ejemplo. Simulación de 500 datos para de una distribución multivariante normal.

```
> library(MASS)
> options(digits=3)
> set.seed(1234)
> mean <- c(230.7, 146.7, 3.6)
> sigma <- matrix( c(15360.8, 6721.2, -47.1, 6721.2, 4700.9, -16.5, -47.1, -16.5, 0.3), nrow=3, ncol=3)
> midato <- mvrnorm(500, mean, sigma)
> midato <- as.data.frame(midato)
> names(midato) <- c("y", "x1", "x2")
> dim(midato)
[1] 500   3
> head(midato, n=20)
       y    x1   x2
 1   98.8  41.3 3.43
 2  244.5 205.2 3.80
 3  375.7 186.7 2.51
 4  -59.2  11.2 4.71
 5  313.0 111.0 3.45
 6  288.8 185.1 2.72
 7  134.8 165.0 4.39
 8  171.7  97.4 3.63
 9  167.2 101.0 3.50
 10 121.1  94.5 4.10
 11 154.6 154.6 4.42
 12  94.2 116.2 4.69
 13 135.1 101.1 3.43
 14 255.8 114.9 3.26
 15 342.5 216.2 3.36
 16 202.7 170.1 3.79
 17 123.1 209.2 4.02
 18 127.0  75.7 2.96
 19 135.0  82.3 3.73
 20 549.1 245.2 2.08
```

Como se ha comentado anteriormente, los paquetes de R se cargas con la función `library`

```
>library(MASS)
```

Se establece un número aleatorio como semilla, de tal forma que los resultados se puedan reproducir en cualquier momento del tiempo. Se introduce el vector de medias y la matriz de varianza-covarianza y con estos objetos como parámetros se genera la muestras de 500 observaciones pseudo-aleatorias. Por conveniencia en el manejo de los datos, se pasa la matriz a un dataframe y se les asigna un nombre a cada una de las 3 variables. Finalmente se imprimen las 20 primeras observaciones.

5.2. Resolución de integrales

De manera general, las integrales a evaluar se puede decir que tienen la forma:

$$E_f[h(X)] = \int_{\chi} h(x)f(x)dx \tag{28}$$

El principio que se utiliza en el método Monte Carlo para realizar una aproximación a esta integral consiste en generar una muestra de variables aleatorias $(X_1, X_2, \ldots, X_n)$ para la función de densidad F y proponer una media empírica tal como:

$$\bar{h_n} = \frac{1}{n} \sum_{j=1}^{n} h(x_j) \tag{29}$$

Por la ley de los grandes números podemos decir que $\bar{h_n}$ converge a $E_f[h(X)]$

Además, si $h^2(X)$ tiene un valor esperado finito bajo la ley de distribución f, entonces:

$$var\left(\bar{h}_n\right) = \frac{1}{n} \int_{\chi} \left(h(x) - E_f[h(X)]\right)^2 f(x)dx \tag{30}$$

que también puede calcularse con la muestra $(X_1, X_2, \ldots, X_n)$, con la expresión:

$$v_n = \frac{1}{n^2} \sum_{j=1}^{n} \left[h(x_j) - \bar{h}_n\right]^2 \tag{31}$$

Más concretamente, debido al teorema central del límite, cuando n sea lo suficientemente grande, se tiene que la expresión

$$\frac{\bar{h}_n - E_f[h(X)]}{\sqrt{v_n}} \tag{32}$$

tiene una distribución $N(0, 1)$

Supongamos una muestra de tamaño n con distribución normal $N(0, 1)$, dada por:

$$\phi(t) = \int_{-\infty}^{t} \frac{1}{\sqrt{2\pi}} e^{-y^2/2} dy \tag{33}$$

Su aproximación por el método Monte Carlo vendra dada por la expresión:

$$\hat{\phi}(t) = \frac{1}{n} \sum_{i=1}^{n} B_{x_i \leq t} \tag{34}$$

Vamos a utilizar R para evaluar algunas probabilidades $P(X \leq t)$ según la distribución normal haciendo para ello varias réplicas de una generación normal.

En la primera línea se indica el número de puntos a tomar.

```
> x <- rnorm(10^8)
> bound=qnorm(c(0.5,0.75,0.8,0.9,0.95,0.99,0.999,0.9999))
> res=matrix(0,ncol=8, nrow=7)
> for (i in 2:8)
+ for (j in 1:8)
+ res[i-1,j]=mean(x[1:10^i]<bound[j])
> matrix(as.numeric(format(res,digi=4)),ncol=8)
      [,1]   [,2]   [,3]   [,4]   [,5]   [,6]   [,7]   [,8]
[1,] 0.5700 0.8200 0.8300 0.9200 0.9600 0.9900 1.0000 1.0000
[2,] 0.4800 0.7390 0.7800 0.8870 0.9510 0.9910 0.9990 0.9990
[3,] 0.4968 0.7459 0.7932 0.8998 0.9511 0.9911 0.9993 0.9998
[4,] 0.5002 0.7504 0.7989 0.8993 0.9492 0.9899 0.9990 0.9999
[5,] 0.5001 0.7498 0.7997 0.8996 0.9498 0.9900 0.9990 0.9999
[6,] 0.5000 0.7501 0.8001 0.9000 0.9501 0.9900 0.9990 0.9999
[7,] 0.5000 0.7500 0.8000 0.9000 0.9500 0.9900 0.9990 0.9999
```

Para valores de t alrededor de $t = 0$ la varianza es aproximadamente $1/4n$ y para alcanzar una precisión de cuatro decimales, se necesitan $2 \times \sqrt{1/4n} \leq 10^{-4}$ simulaciones , es decir, sobre $n = 10^8$ simulaciones. La tabla 3.1 proporciona la evolución de esta aproximación para varios valores de t y muestra una evaluación bastante exacta para una simulación de 100 millones de iteraciones.

Cuando se trata de obtener probabilidades de sucesos muy extraños, por ejemplo aquellos que siguiendo una distribución normal, se hallan situados en los extremos de la campana alejados del centro, la simulación a partir de una distribución $N(0, 1)$ no funciona correctamente, y es preciso hacer uso del método Monte Carlo si se desea obtener un resultado fiable.

Ejemplo. Se desea obtener la probabilidad $P(X > 20)$ en una distribución normal estándar $N(0, 1)$. En este caso la integral que habría que calcular seria:

$$P(X > 20) = \int_{20}^{\infty} \frac{exp(-\frac{x^2}{2})}{\sqrt{2\pi}} dx \tag{35}$$

y se reescribe como el valor esperado bajo una distribución $U(0, 1/20)$

$$\int_{0}^{1/20} \frac{exp(-\frac{1}{2 \cdot u^2})}{20u^2\sqrt{2\pi}} 20 \cdot du \tag{36}$$

```
> h=function(x){ 1/(x^2*sqrt(2*pi)*exp(1/(2*x^2)))}
> par(mfrow=c(2,1))
> curve(h,from=0,to=1/20,xlab="x",ylab="h(x)",lwd="2")
> I=1/20*h(runif(10^4)/20)
> estint=cumsum(I)/(1:10^4)
> esterr=sqrt(cumsum((I-estint)^2))/(1:10^4)
> plot(estint,ty="l",lwd=2,
+ ylim=mean(I)+20*c(-esterr[10^4],esterr[10^4]),ylab="")
> lines(estint+2*esterr,col="gold",lwd=2)
> lines(estint-2*esterr,col="gold",lwd=2)
```

La probabilidad estimada que se obtiene es$2{,}505e^{-89}$ con un error de $\pm 3{,}61e^{-90}$. Graficamente:

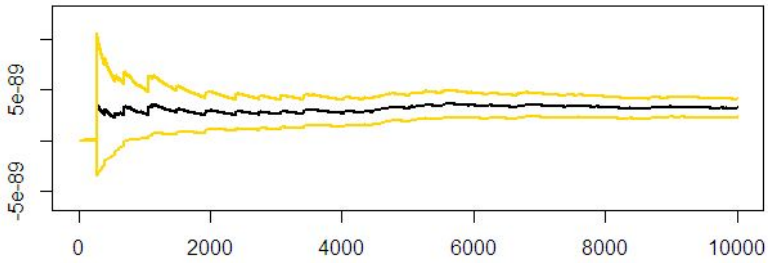

Figura 5.1 *Resultado del cálculo de π para 100 lanzamientos.*

Como ya se ha mencionado, en los extremos de la función de distribución, el método de Monte Carlo directo falla en cuanto nos alejamos demasiado del centro. Por ejemplo, si $Z \sim N(0,1)$ y estamos interesados en la probabilidad $P(Z > 4{,}5)$, la cual es muy pequeña.

La simulación $Z^{(i)} \sim N(0,1)$ solo produce una ocurrencia cada tres millones de iteraciones. Por supuesto, el problema es que estamos interesados en un suceso muy raro y por lo tanto, la simulación ingenua de f requerirá un gran número de simulaciones para obtener una respuesta estable. Sin embargo, la técnica del muestreo por importancia, puede mejorar en gran medida la precisión y consecuentemente permitir una reducción del número de simulaciones en varios órdenes de magnitud.

si tenemos en cuenta una distribución limitaciones en $(4{,}5, \infty)$ entonces, la variación adicional e innecesaria del estimador de Monte Carlo, debido a que simulan ceros (cuando x ¡4.5) desaparece. Una elección natural es tomar g como la densidad de la distribución exponencial e truncada en 4,5

$$g(y) = \int_{4,5}^{\infty} e^{-x} dx = e^{-(y-4,5)} \tag{37}$$

y el estimador de muestreo por importancia correspondiente a la probabilidad en esa parte de la distribución seria:

$$\frac{1}{n}\sum_{i=1}^{n} \frac{f(Y^{(i)})}{g(Y^{(i)})} = \frac{1}{n}\sum_{i=1}^{n} \frac{e^{-Y_i^2/2+Y_i-4,5}}{\sqrt{2\pi}} \tag{38}$$

Donde las $Y_i's$ son las variables aleatorias generadas en base a g

```
> Nsim=10^6
> y=rexp(Nsim)+4.5
> weit=dnorm(y)/dexp(y-4.5)
> plot(cumsum(weit)/1:Nsim,type="l")
> abline(a=pnorm(-4.5),b=0,col="red")
> mean(weit)
[1] 3.399389e-06
```

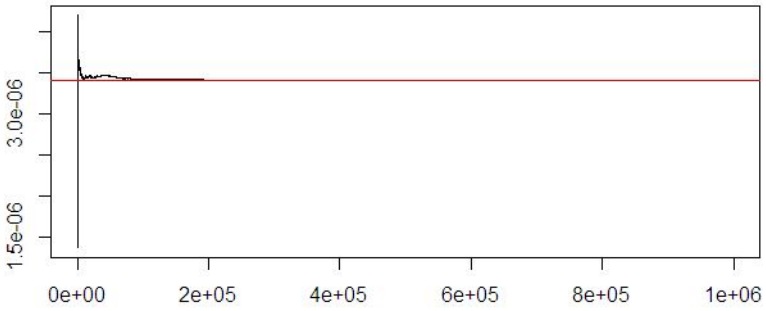

Figura 5.2 *Monte Carlo con Muestreo por Importancia.*

6. Problemas

6.1. La Aguja del Conde de Buffon

Demostrar analíticamente que la probabilidad viene dada por $P = \frac{2l}{\pi d}$

Al caer las agujas sobre el entramado de líneas, independientemente de si corta o no corta alguna línea, existen 3 posiciones que se reducen a 1, veamos:

1. La aguja se queda paralela a las líneas.

2. La aguja se queda perpendicular a las líneas

3. La aguja se queda formando un ángulo ϕ con las líneas.

La primera y la segunda posición corresponden a que la aguja forme un ángulo de 0º o de 90º respectivamente, con lo cual, son casos específicos de la tercera opción. Por tanto podemos decir, que al caer las agujas, forman un ángulo ϕ con las líneas. Establecemos que el ángulo ϕ varia entre 0 y π, por tanto, $\phi \in [0, \pi]$. Establecemos los parámetros del problema con los que trabajar:

- Distancia entre las líneas. D

- Longitud de las agujas. L, tal que $L < D$

- Probabilidad de que la aguja cruce una línea al caer. P

Al caer las agujas sobre la superficie, establecemos la distancia de cada una de ellas a la línea paralela más próxima, como la distancia de su centro a dicha línea, y denotamos esta distancia por H.

Entonces, la condición necesaria para que la aguja corte la línea sera: $H = \frac{L}{2} sen\phi > \frac{D}{2}$

Haciendo uso del concepto de probabilidad geométrica, podemos decir que la probabilidad de que una aguja corte una línea vendrá dada por el ratio entre el área de la sinusoide descrito por $L \cdot sen\phi$ sobre el área $[0, \pi] \times [0, D]$ del rectángulo que la contiene:

$$P = \frac{\int_0^\pi L \cdot sen(\phi) d\phi}{\pi \cdot D} \tag{39}$$

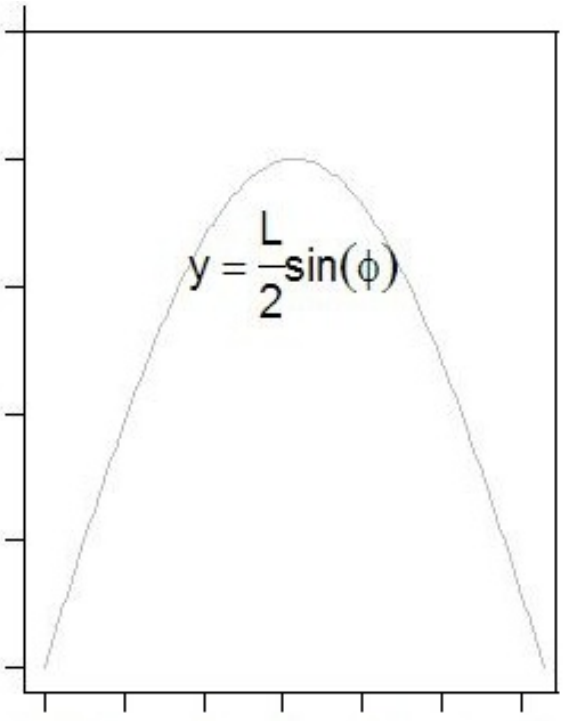

Figura 6.1 *Sinusoide que contiene la probabilidad de que la aguja corte alguna línea*

al integrar esta expresión se obtiene una solución analítica a la probabilidad de corte:

$$p = \frac{2 \cdot L}{\pi \cdot D} \tag{40}$$

6.1.1. Simulación del Problema de la Aguja de Buffon

El lenguaje de programación R dispone de una librería que contiene una simulación bastante ilustrativa del problema de Buffon. A continuación se dan los pasos para la ejecución de programa y se muestran algunos resultados y conclusiones.

1. Instalar el paquete `animation`.
   `>install.packages('animation')`
2. Cargar la libreria `animation` en la sesión de trabajo actual.
   `>library(animation)`
3. Ejecutar el programa.
   `>buffon.needle()`
4. Cambiar el número de agujas.
   `>ani.options(nmax=100)`

El programa viene configurado con unos parámetros por defecto, estos parámetros establecen un número de 50 agujas, con un tamaño de $0,8$ sobre una distancia entre líneas de 1, tal y como se puede apreciar en el código del programa incluido al final.

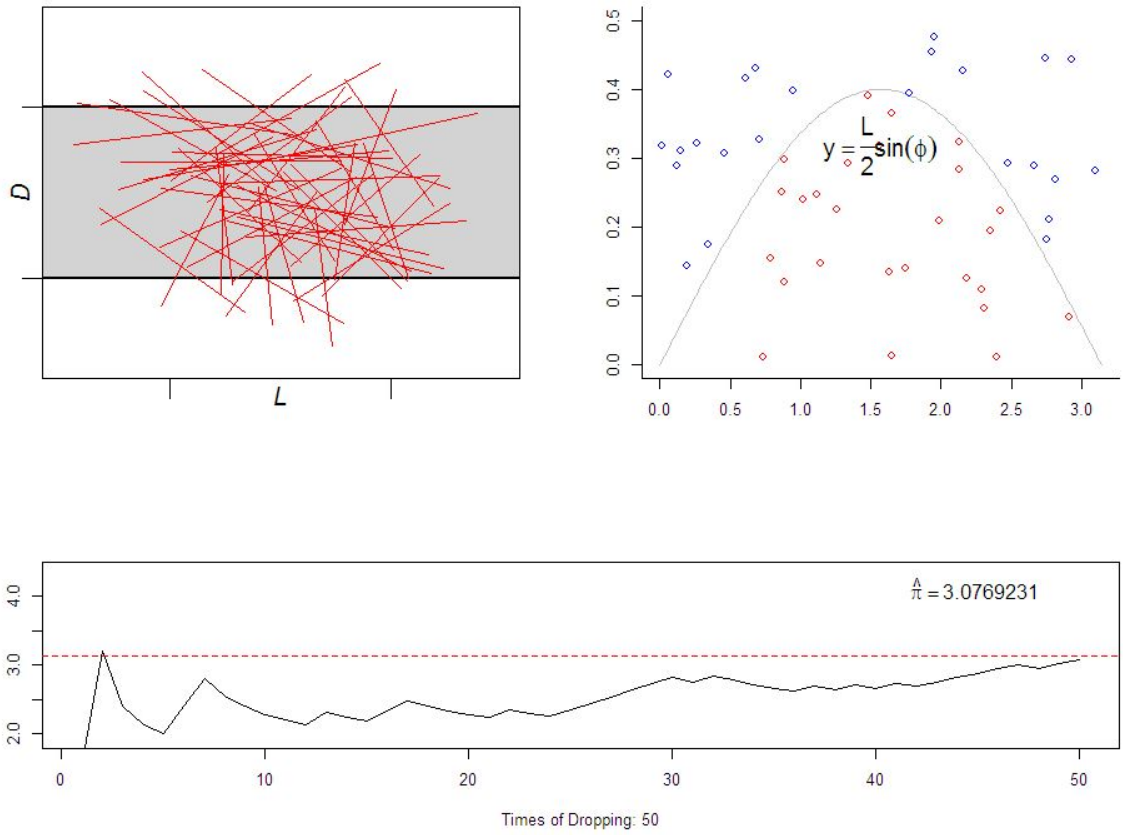

Figura 6.2 *Simulacion del problema de Buffon para 50 lanzamientos.*

Con el fin de observar la convergencia del error, podemos realizar la simulación varias veces con un número distinto de agujas. Al hacer esto comprobamos que efectivamente, al tratarse de la aplicación del método Monte Carlo en básico, el error converge como $1/\sqrt{N}$ siendo N el número de agujas o lanzamientos (da lo mismo lanzar 50 agujas que lanzar 50 veces una misma aguja).

Observamos que los valores de π que se obtienen para 50 y para 100 lanzamientos apenas varían, y no nos dan siquiera ni una cifra decimal correcta.

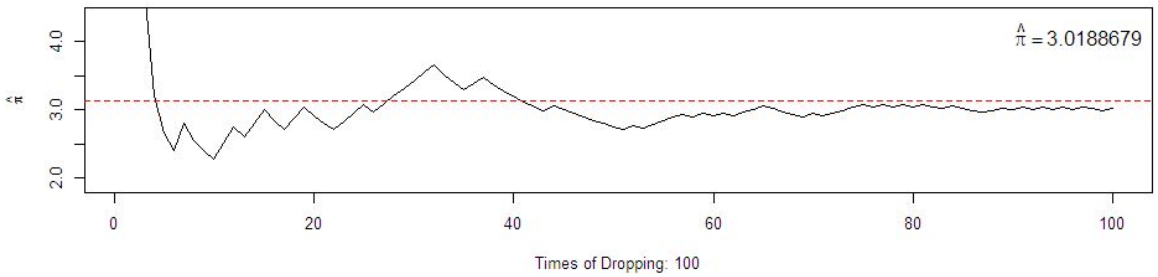

Figura 6.3 *Resultado del cálculo de π para 100 lanzamientos.*

Según la teoría, para obtener al menos dos cifras decimales correctas, tendriamos que realizar al menos:

$$0,001 = \frac{1}{\sqrt{N}} \longrightarrow N = \frac{1}{(0,001)^2} = \frac{1}{0{,}000001} = 1000000 \tag{41}$$

A continuación se exponen los resultados de las distintas simulaciones que corroboran esta hipótesis.

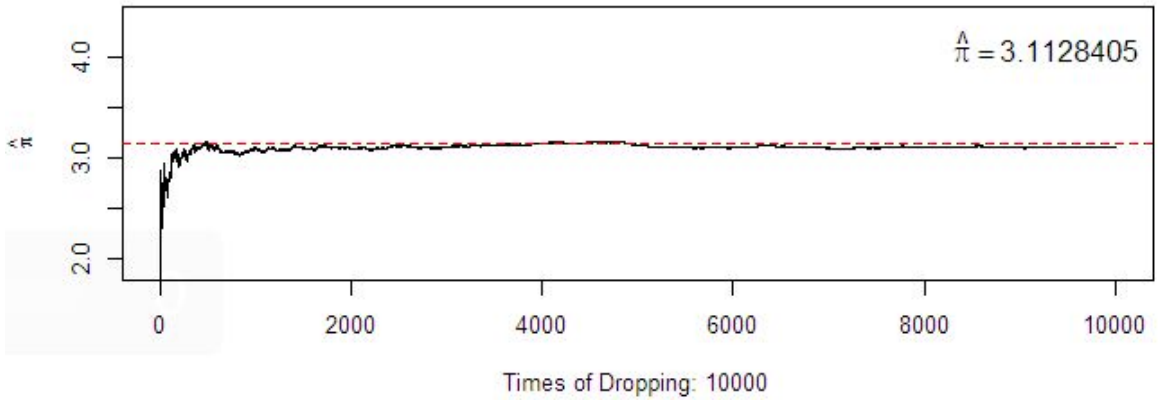

Figura 6.4 *Resultado del cálculo de π para 1000 lanzamientos.*

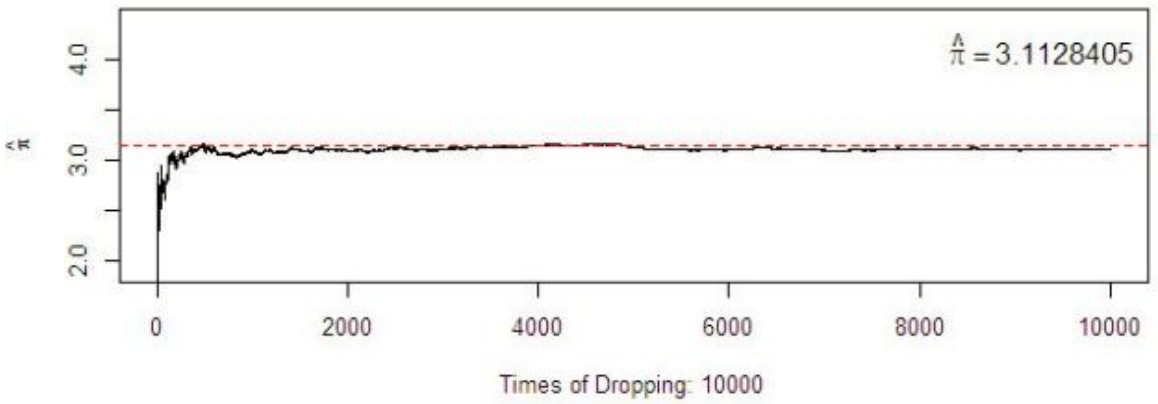

Figura 6.5 *Resultado del cálculo de π para 10000 lanzamientos.*

6.1.2. Código R del programa de la aguja de Buffon

```
function (l = 0.8, d = 1, redraw = TRUE, mat = matrix(c(1, 3, 2, 3), 2), heights = c(3, 2),
col = c("lightgray", "red", "gray", "red", "blue", "black", "red"), expand = 0.4, type = "l", ...)
{
    j = 1
    n = 0
    PI = rep(NA, ani.options("nmax"))
    x = y = x0 = y0 = phi = ctr = NULL
    layout(mat, heights = heights)
    while (j <= length(PI)) {
        dev.hold()
        plot(1, xlim = c(-0.5 * l, 1.5 * l), ylim = c(0, 2 *
            d), type = "n", xlab = "", ylab = "", axes = FALSE)
        axis(1, c(0, l), c("", ""), tcl = -1)
        axis(1, 0.5 * l, "L", font = 3, tcl = 0, cex.axis = 1.5,
            mgp = c(0, 0.5, 0))
        axis(2, c(0.5, 1.5) * d, c("", ""), tcl = -1)
        axis(2, d, "D", font = 3, tcl = 0, cex.axis = 1.5, mgp = c(0,
            0.5, 0))
        box()
        bd = par("usr")
        rect(bd[1], 0.5 * d, bd[2], 1.5 * d, col = col[1])
        abline(h = c(0.5 * d, 1.5 * d), lwd = 2)
        phi = c(phi, runif(1, 0, pi))
        ctr = c(ctr, runif(1, 0, 0.5 * d))
        y = c(y, sample(c(0.5 * d + ctr[j], 1.5 * d - ctr[j]),
            1))
        x = c(x, runif(1, 0, l))
        x0 = c(x0, 0.5 * l * cos(phi[j]))
        y0 = c(y0, 0.5 * l * sin(phi[j]))
        if (redraw) {
            segments(x - x0, y - y0, x + x0, y + y0, col = col[2])
        }
        else {
            segments(x[j] - x0[j], y[j] - y0[j], x[j] + x0[j],
                y[j] + y0[j], col = col[2])
        }
        xx = seq(0, pi, length = 200)
        plot(xx, 0.5 * l * sin(xx), type = "l", ylim = c(0, 0.5 *
            d), bty = "l", xlab = "", ylab = "", col = col[3])
        idx = as.numeric(ctr > 0.5 * l * sin(phi)) + 1
        if (redraw) {
            points(phi, ctr, col = c(col[4], col[5])[idx])
        }
        else {
            points(phi[j], ctr[j], col = c(col[4], col[5])[idx[j]])
        }
        text(pi/2, 0.4 * l, expression(y == frac(L, 2) * sin(phi)),
            cex = 1.5)
        n = n + (ctr[j] <= 0.5 * l * sin(phi[j]))
        if (n > 0)
            PI[j] = 2 * l * j/(d * n)
        plot(PI, ylim = c((1 - expand) * pi, (1 + expand) * pi),
            xlab = paste("Times of Dropping:", j), ylab = expression(hat(pi)),
            col = col[6], type = type, ...)
        abline(h = pi, lty = 2, col = col[7])
        pihat = format(PI[j], nsmall = 7, digits = 7)
        legend("topright", legend = substitute(hat(pi) == pihat,
            list(pihat = pihat)), bty = "n", cex = 1.3)
        ani.pause()
        j = j + 1
    }
    invisible(PI)
}
```

6.2. Generación de números aleatorios

Programación de un generador congruente de la forma: $X_{n+1} = (a \cdot X_n + c) \bmod m$
Mediante la programación R se puede construir una sencilla función que genere un número concreto de valores aleatorios según la formula congruente enunciada.

```
congruente<- function(n, a, x0, c, m) {
% n es el número de valores a generar
% a es el factor multiplcativo
% x0 es la semilla
% c es el factor aditivo
% m es el factor del módulo
x<-numeric(n)
semilla<-x0
x[1]=(a*semilla +c )%%m
for (i in 2:n)
{
x[i] = (a*x[i-1] +c )%%m
}
x
}
```

Se prueba para la generación de 20 valores.

- **caso a)** $m = 31, c = 0, a = 7, X = 1$. Se prueba para la generación de 20 valores.

```
      congruente(20,7,1,0,31)
      [1]  7 18  2 14  5  4 28 10  8 25 20 16 19  9  1  7 18  2 14  5
```

  Se observa un periodo de 15 números, tras el cual se vuelve a repetir la secuencia.

- **caso b)** $m = 31, c = 0, a = 3, X = 1$. Se prueba para la generación de 40 valores.

```
      congruente(40,3,1,0,31)
 [1]  3  9 27 19 26 16 17 20 29 25 13  8 24 10 30 28 22  4 12  5 15 14 11  2  6 18 23  7 21  1  3  9 27 19 26 16
```

  Se observa un periodo de 31 números, tras el cual se vuelve a repetir la secuencia.

- **caso c)** $m = 32, c = 1, a = 5, X = 1$.Se prueba también para la generación de 40 valores.

```
congruente(40,5,1,1,32)
[1] 6 31 28 13  2 11 24 25 30 23 20  5 26  3 16 17 22 15 12 29 18 27  8  9 14  7  4 21 10 19  0  1  6 31 28 13
```

  Se observa igualmente un periodo de 31 números, tras el cual se vuelve a repetir la secuencia.

Para que los números generados estén en el intervalo $[0, 1]$ hay que dividir por el valor del módulo.

```
congruente2<- function(n, a, x0, c, m) {
# n es el número de valores a generar
# a es el factor multiplcativo
# x0 es la semilla
# c es el factor aditivo
# m es el factor del módulo
x<-numeric(n)
semilla<-x0
x[1]=(a*semilla +c )%%m
for (i in 2:n)
{
x[i] = (a*x[i-1] +c )%%m
}
alea <- x/m #normaliza al intervalo [0,1]
alea
}
```

Ejemplo, se ejecuta la función `congruente2()` con los valores del caso c) $m = 32, c = 1, a = 5, X = 1$:

```
      > congruente2(40,5,1,1,32)
 [1] 0.18750 0.96875 0.87500 0.40625 0.06250 0.34375 0.75000 0.78125 0.93750 0.71875 0.62500 0.15625 0.81250 0.0937
[18] 0.46875 0.37500 0.90625 0.56250 0.84375 0.25000 0.28125 0.43750 0.21875 0.12500 0.65625 0.31250 0.59375 0.0000
[35] 0.87500 0.40625 0.06250 0.34375 0.75000 0.78125
```

7. Bibliografía

- Illana J. I. (2013). *Métodos Monte Carlo* . http://www.ugr.es/ jillana/Docencia/FM/mc.pdf Haschke P. (2013) *An Introduction to R*, Creative Commons License http://www.rochester.edu/college/psc/thestarlab/help/rcourse/R-Course.pdf
- Robert I. Kabacoff (2011). *R in Action*, second edition, Manning Publications Co. ISBN: 9781935182399
- Robert, C. and Casella, G. (2004). *Monte Carlo Statistical Methods*, second edition. Springer-Verlag, New York.
- Weinzierl S. (2000). *Introduction to Monte Carlo method* . arXiv:hep-ph/0006269v1
- R Development Core Team (1999). *Introducción a R* . http://cran.r-project.org/doc/contrib/R-intro-1.1.0-espanol.1.pdf
- SOBOL, I. M. (1983). *Método de Montecarlo*, 2ª Ediction. Editorial MIR, Moscú.