Introducción a R

Sesión 5
Programación

Joaquín Ortega Sánchez

Centro de Investigación en Matemáticas, CIMAT
Guanajuato, Gto., Mexico

Verano de Probabilidad y Estadística
Junio-Julio 2008

Outline

Outline

Iteraciones

Una iteración es un ciclo de operaciones que se repiten con cambios menores. En algunos lenguajes de programación la multiplicación de dos matrices requiere al menos tres ciclos encadenados. En `R` estas operaciones son mucho más sencillas de formular y también más eficientes. Siempre que sea posible, trate de evitar iteraciones. La gran mayoría de las veces una operación entre matrices es mucho más rápida. Trate de usar vectores y funciones como `apply`.

Iteraciones

Por ejemplo, definamos un vector con 50.000 componentes y calculemos el cuadrado de cada componente primero usando las propiedades de R de realizar cálculos componente a componente y luego usando un ciclo, y comparemos los tiempos que tarda cada una de estas operaciones.

```
> x <- 1:50000
> y <- x^2
> for (i in 1:50000) z[i] <- x[i]^2
```

La segunda expresión no sólo es más larga sino que resulta en un cálculo mucho más lento que la primera.
Sin embargo, hay situaciones en las cuales es imposible evitar el uso de bucles. A continuación examinaremos las instrucciones de R que permiten construir bucles en los programas.

for

La sintaxis de esta instrucción es

```
> for (i in ivalores) {instrucciones de R}
```

como en el siguiente ejemplo

```
> for (i in 1:10) {print (i)}
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
[1] 6
[1] 7
[1] 8
[1] 9
[1] 10
```

for

Si hay una sola instrucción, las llaves {} pueden omitirse. El objeto `ivalores` puede ser de cualquier modo: un vector, una variable, una matriz, etc. `R` recorre todos los elementos del argumento reemplazando sucesivamente la variable `i` por los elementos. La variable `i` también puede tener cualquier modo: numérico, carácter, lógico o una combinación de estos.

```
> for (i in c(3,2,9,6)) {print (i^2)}
[1] 9
[1] 4
[1] 81
[1] 36
```

for

```
> medios.transporte <- c("carro", "camion",
    "metro", "moto")
> for (vehiculo in medios.transporte)
    {print (vehiculo)}
[1] "carro"
[1] "camion"
[1] "metro"
[1] "moto"
```

for

Para escribir valores en medio de un bucle es necesario usar la función `print`. Una instrucción como

```
> for (x in 1:4) { x }
```

no escribe los resultados en la pantalla. Hay que escribir

```
> for (x in 1:4) { print(x) }
```

para imprimir el resultado dentro del bucle.

for

Como ejemplo vamos a escribir una función que usa el método MonteCarlo para estimar el valor de π. Generamos puntos al azar con distribución uniforme sobre el cuadrado con vertices $(1,1),(1,-1),(-1,1)$ y $(-1,-1)$. La probabilidad de que el punto caiga dentro de la circunferencia de centro el origen y radio 1 es igual al área del círculo correspondiente, que es $\pi/4$. Vamos a generar 1000 números al azar y contamos cuántos de ellos caen dentro de la circunferencia.

for

```
> s <- 0
> for (i in 1:1000)
{
  x <- runif(1,-1,1)
  y <- runif(1,-1,1)
  if(x^2+y^2 < 1) s <- s+1
}
> piest <- 4*s/1000
> piest
[1] 3.128
> error = abs(pi - piest)
> error
[1] 0.01359265
```

Ejercicio

Ejercicio 5.1

1. Escriba un algoritmo usando la instrucción `for` que calcule el promedio móvil de orden 2 de un vector `x`.
2. Pruebe el algoritmo usando el archivo `Nile`.

while

Si no se conoce el número de ciclos que se desea realizar antes de comenzar, se usa `while` que permite iterar hasta que cierto criterio se cumpla. Como ejemplo, vamos a sumar los enteros positivos hasta que la suma pase de 1000.

```
> n <- 0
> suma <- 0
> while (suma <= 1000)
{
  n <- n+1
  suma <- suma + n
}
> suma
[1] 1035
> n
[1] 45
```

Vemos que el primer valor de la suma que pasa de 1000 es 1035 y que hicieron falta 45 iteraciones para llegar a este valor.

while

Como ejemplo vamos a escribir otra función para estimar el valor de π usando la instrucción `while` para controlar el error.

```
> ss <- 0; nn <- =0; error <- 1
> while (error > .001)
{
  nn <- nn+1
  x <- runif(1,-1,1)
  y <- runif(1,-1,1)
  if(x^2+y^2 < 1) ss <- ss+1
  piest <- 4*ss/nn
  error = abs(pi - piest)
}
> piest
[1] 3.142278
> error
[1] 0.000685417
> nn
[1] 2467
```

Ejercicio

Ejercicio 5.2

1. Escriba un algoritmo usando la instrucción `while` que genere números al azar con distribución uniforme en [0,1] y los sume hasta llegar a 100. El resultado es el número de sumandos.

repeat

La tercera alternativa para hacer bucles en un programa es la instrucción `repeat`. Este comando repite un conjunto de instrucciones hasta que se satisfaga un criterio de parada. En `R` se usa la palabra `break` para indicar el momento de parar. Como ejemplo vamos a resolver el mismo problema anterior usando la instrucción `repeat`.

repeat

```
> n <-0
> suma <- 0
> repeat
{
  n <- n+1 +
  suma <- suma + n
  if ( suma > 1000) break
}
> suma
[1] 1035
> n
[1] 45
```

repeat

Otra instrucción que se usa con la instrucción `repeat` es `next`. Si `R` se encuentra esta instrucción en un bucle iniciado con `repeat`, deja de ejecutar las instrucciones del bucle y recomienza el bucle desde el principio. Es importante observar que si olvidamos incluir la instrucción `break`, `R` puede entrar en un bucle infinito.
Veamos como ejemplo otra versión de la estimación de π usando MonteCarlo, pero ahora con la instrucción `repeat`.

repeat

```
> ss <- 0; nn <- 0; error <- 1
repeat
{
nn <- nn+1
x <- runif(1,-1,1)
y <- runif(1,-1,1)
if(x^2+y^2 < 1) ss <- ss+1
piest <- 4*ss/nn
error = abs(pi - piest)
if (error < .001) break
}
> piest
[1] 3.140741
> error
[1] 0.0008519128
> nn
[1] 270
```

Ejercicio

Ejercicio 5.3

1. Escriba un algoritmo usando la instrucción `repeat` que genere números al azar con distribución uniforme en [0,1] y los sume hasta llegar a 100. El resultado es el número de sumandos.

Vectorización

Dada la facilidad de trabajar con vectores en R, usualmente es posible reescribir bucles en términos de operaciones con vectores que resultan más breves y legibles. Veamos de nuevo el ejemplo de la suma de los enteros. Resulta más simple calcular 1000 valores y verificar en que lugar se satisface el criterio de parada.

```
> n <- 1:1000
> su <- cumsum (n)
> su [su > 1000][1]
[1] 1035
> n [su > 1000][1]
[1] 45
```

Vectorización

Para bucles grandes este enfoque es más rápido y más fácil de leer. Sin embargo es necesario tener una idea aproximada del número de iteraciones que se requieren.
Para er otro ejemplo vamos a estimar el valor de π por el método MonteCarlo usando operaciones vectoriales.

```
> x <- runif(1000,-1,1)
> y <- runif(1000,-1,1)
> z <- x^2+y^2
> suma <- sum(as.numeric(z<1))
> piest4 <- 4*suma/1000
> piest4
[1] 3.144
> error4 = abs(pi-piest4)
> error4
[1] 0.002407346
```

Ejercicio

Ejercicio 5.4

1. Escriba un algoritmo sin usar las instrucciones `repeat` o `while` que genere números al azar con distribución uniforme en [0,1] y los sume hasta llegar a 100. El resultado es el número de sumandos.

Outline

Funciones

Una función es, simplemente, una sucesión de instrucciones que se juntan para formar una nueva instrucción, que es el nombre de la función. Las funciones de R tienen flexibilidad y capacidad similar a la de otros lenguajes de programación modernos, como PASCAL o C. Las funciones reciben argumentos y devuelven valores. Todas las otras variables utilizadas en la definición de la función son variables internas y desaparecen una vez que la función ha sido ejecutada. El uso de una función en R es similar al uso matemático. En matemáticas escribimos $y = f(x)$ y en R

```
> y <- f(x)
```

Funciones

Para demostrar como se escribe una función veamos un ejemplo. Vamos a definir una función llamada `cubo` que toma un número y lo eleva a la potencia tres:

```
> cubo <- function (x)
  { return (x^3) }
> cubo(2)
[1] 8
```

Después de declararla, esta función puede usarse como cualquier otra función de `R`. Sólo puede ser distinguida de las funciones residentes por su ubicación, pues se almacena en un directorio diferente.

Funciones

Esta función tiene la misma flexibilidad de las otras funciones de R y podemos usarla no sólo con variables, sino también con vectores:

```
> x <- 1:5
> cubo(x/2)
[1]  0.125  1.000  3.375  8.000 15.625
```

iterativamente:

```
> cubo (cubo (x) )
[1]      1    512  19683 262144 1953125
```

Funciones

o usar como argumento una matriz:

```
> x <- matrix (1:4, 2, 2)
> x
     [,1] [,2]
[1,]    1    3
[2,]    2    4
> cubo (x)
     [,1] [,2]
[1,]    1   27
[2,]    8   64
```

Funciones

Una función puede tener más de un argumento (o ninguno). La siguiente función divide el primer argumento entre el segundo.

```
> divide <- function (x,y) { return (x/y) }
> divide (8,3)
[1] 2.666667
> a <- 1:5
> b <- 11:15
> divide (b,a)
[1] 11.000000 6.000000 4.333333 3.500000
3.000000
> divide (11:15, 3)
[1] 3.666667 4.000000 4.333333 4.666667
5.000000
```

Funciones

La sintaxis general para definir una función es la siguiente

nombre`<- function (`*argumentos*`)`
`{`
instrucciones de la función
`return (`*resultados*`)`
`}`

Las expresiones en cursiva deben reemplazarse por expresiones y nombres válidos. Los argumentos son una lista de parámetros que serán usados internamente por la función. Las instrucciones pueden ser cualesquiera instrucciones válidas de `R`, que serán evaluadas a medida que `R` las ejecuta, y los resultados pueden ser datos o variables.

Funciones

En lugar de usar `return`, es posible poner el nombre de una variable como expresión final. `R` devuelve el resultado de la última expresión en la función, pero usar `return` es una manera más apropiada de devolver los resultados. Esto asegura que el resultado de la función sea realmente el que uno desea. Si no desea que la función escriba ningún resultado, use la instrucción `invisible`. La expresión `return(invisible(x))` devuelve el contenido de la variable `x` pero no lo escribe en pantalla.

Ejercicio

Ejercicio 5.5

1. Escriba una función que transforme grados Fahrenheit a Centigrados:

$$c = \frac{5}{9}(f - 32).$$

Ambito

Todas las variables declaradas dentro de una función son locales y desaparecen luego de ejecutada la función. Su ámbito (el ambiente en el cual son conocidas) no se extiende más allá de los límites de la función. Para demostrarlo veamos el siguiente ejemplo. Observe el valor de `z` antes y después de la ejecución de la función `f`.

```
> z <- "prueba"
> f <- function (x, y) {z <- x/y; return(z)}
> a <- 3
> b <- 4
> z
[1] "prueba"
> f(a,b)
[1] 0.75
> z
[1] "prueba"
```

Ejercicio

Ejercicio 5.6
La fórmula para calcular la cuota mensual de un prestamo es la siguiente:

$$C = P\frac{r/1200}{1-(1-r/1200)^{12a}}$$

donde P es el monto de la deuda, a es el número de años del prestamo y r es la tasa de interés anual.

1. Escriba una función en `R` que calcule el monto de las cuotas.

Parámetros y valores automáticos

Una función puede tener muchos argumentos y en esos casos es cómodo no tener que incluir todos los parámetros. En esta situación es conveniente poder asignar valores 'por defecto' o automáticos a los parámetros. Esto es posible en R usando el signo =. Si llamamos una función sin especificar explícitamente el parámetro, se usará su valor 'por defecto'. Veamos un ejemplo. La siguiente función asigna valores automáticos a sus parámetros.

```
  > ff <- function (x = 1:10, y = (1:10)^2,
showgraph = T)
  {
  if (showgraph)  plot (x,y)
  else print (cbind (x,y))
  return (invisible ())
  }
```

Parámetros y valores automáticos

Al declarar esta función, `x` tiene como valor por defecto `1:10` e `y` tiene `(1:10)^2`. El parámetro `showgraph` fue fijado en `T` (TRUE). En consecuencia, todos los comandos siguientes producen el mismo resultado

```
> ff (1:10, (1:10)^2, T)
> ff (1:10, (1:10)^2)
> ff (1:10)
> ff (y = (1:10)^2)
> ff (showgraph = T)
> ff ()
```

que es un gráfico de `x` versus `x^2` para los enteros del 1 al 10. Si escribimos

```
> ff(11:20)
```

obtenemos un gráfico esencialmente igual, el único cambio son los puntos del eje *x* que ahora van de 11 a 20, en lugar de ir de 1 a 10.

Parámetros y valores automáticos

La función no calcula el cuadrado de *x* sino que siempre calcula el cuadrado de los números del 1 al 10. Si queremos que calcule `x^2` es necesario modificar la definición. Los nuevos parámetros usan otros parámetros en las asignaciones automáticas:

```
  f1 <- function (x=1:10, y=x^2,
error=(length(x)!= length(y)))
  {
  if (error) return ("longitudes de x e y no
son iguales.")
  else return (cbind (x,y))
  }
  f1(2:8)
  f1(x=1:5, y=1:9)
  [1] "longitudes de x e y no son iguales"
```

Funciones

Si quiere ver la definición de una función en lugar de ejecutarla, escriba el nombre sin los paréntesis. Hay muchas funciones residentes en `R` y es posible no sólo ver su código sino incluso modificarlo.

Ejercicio

Ejercicio 5.7

1. Escriba una función llamada `pazar` que comience en `x` y sume variables aleatorias de valores ± 1 con probabilidades 1/2 hasta llegar a `b` o a 0. El resultado debe ser el número de pasos y el lugar donde termina. Asigne por defecto los valores `x=50` y `b=100`.

Ejemplo

Veamos un ejemplo inicial sobre el tiempo de ejecución de procesos iterativos usando ciclos y cálculos vectorizado.
En el ejemplo que consideramos vamos a generar un vector de tamaño 100000 de la distribución uniforme en $[-1, 1]$ y vamos a sumar la valores positivos de esta serie. En primer lugar usamos una instrucción `for` y en segundo lugar hacemos una cálculo vectorizado.

Ejemplo

```
> x <- runif(100000,-1,1)
> t <- Sys.time()
> k <- 0
> for (a in x) {
if(a>0) y <- a
else y <- 0
k <- k + y
}
> Sys.time()-t
Time difference of 0.3440001 secs
> t <- Sys.time()
> k <- sum(x[x>0])
> Sys.time() - t
Time difference of 0.01600003 secs
> 0.3440001/0.01600003
[1] 21.49997
```

Ejemplo

Veamos como segundo ejemplo una función que calcula los elementos de la sucesión de Fibonacci. Esta sucesión se define de la siguiente manera. los dos primeros elementos son $x_0 = 0$, $x_1 = 1$, y de allí en adelante los términos están dados por la fórmula

$$x_n = x_{n-1} + x_{n-2}.$$

La función que definimos a continuación calcula los n primeros valores de la sucesión.

Ejemplo

```
> fib1 <- function(n)
{
res <- c(0,1)
for (i in 3:n)
  res[i] <- res[i-1] + res[i-2]
return(res)
}

> fib1(10)
```

Ejemplo

Sin embargo, esta función tiene un inconveniente: estamos aumentando el tamaño del vector `res` constantemente para almacenar un nuevo valor de la serie, y esto resulta computacionalmente costoso.
En este caso es fácil de evitar porque sabemos de entrada el tamaño del vector.

Ejemplo

```
> fib2 <- function(n)
{
res <- numeric(n)
res[2] <- 1
for (i in 3:n)
  res[i] <- res[i-1] + res[i-2]
return(res)
}

> fib2(10)
```

Ejemplo

Para ver cuánto hemos ganado vamos a comparar los tiempos de ejecución de ambos programas.

```
  > system.time(fib1(10000))
  [1] 0.34 0.00 0.35 NA NA
  > system.time(fib2(10000))
  [1] 0.09 0.00 0.09 NA NA
  >
system.time(fib1(10000))/system.time(fib2(10000))
  [1] 4.125 NaN 4.375 NA NA
```

El primer número corresponde a tiempo de cpu del usuario, el segundo a tiempo de cpu de la máquina, el tercero es el tiempo transcurrido y los siguientes dos tiempos son de subprocesos pero en windows son siempre `NA`.

Ejercicio

Ejercicio 5.8

1. La siguiente función supuestamente calcula el *n*-ésimo término de la sucesión de Fibonacci, pero en realidad tiene varios errores de diverso tipo (de sintaxis, de programación, de concepto). Encuéntrelos y corríjalos.

Ejercicio

```
fib3 <- function(nb)
{
x <- 0
x1 _ 0
x2 <- 1
while (n>0))
x <- x1 + x2
x2 <- x1
x1 <- x
n <- n-1
}
```

Argumentos

Hay dos maneras de asignar valores a los argumentos de una función, por nombre o usando la posición del argumento en la definición de la función. Por ejemplo, para una función definida de la siguiente manera

```
  > fn1 <- function(datos, lista, graficar,
limite)
      { ... }
```

las siguientes instrucciones producen el mismo resultado:

```
  > fn1(a,ll,TRUE,20)
  > fn1(a,ll,graficar=TRUE,limite=20)
  >
fn1(datos=a,lista=ll,graficar=TRUE,limite=20)
  >
fn1(graficar=TRUE,limite=20,datos=a,lista=ll)
```

Instrucciones Condicionales

En el desarrollo de funciones en `R` es posible usar instrucciones condicionales de la forma

`if (`*condición*`)` *expresión 1* `else` *expresión 2*

donde *condición* debe tener como resultado un valor lógico. Si este valor es `TRUE`, se ejecuta *expresión 1* y en caso contrario se ejecuta *expresión 2*.
Hay que tener en cuenta que si `expresión 1` resulta en un vector de valores lógicos, sólo el primero de ellos se usará para determinar el valor de la instrucción condicional.

Instrucciones Condicionales

```
> x <- 1:3
> x > 2
[1] FALSE FALSE TRUE
> if(x > 2) print('a') else print('b')
[1] 'b'
Warning message:
la condición tiene longitud > 1 y sólo el
primer elemento será usado in: if (x > 2)
print('a') else print('b')
```

Instrucciones Condicionales

Veamos un ejemplo del uso de una instrucción condicional en una función. Vamos a definir una función llamada `mg` que calcula la media geométrica de un vector: Si v es un vector numérico, la media geométrica se define por

$$\bar{v}_g = \left(\prod_{i=1}^{n} v_i \right)^{1/n}$$

donde n es la longitud del vector v. Vamos a calcular este valor tomando logaritmos, dividiendo por la longitud y luego exponenciando:

Instrucciones Condicionales

```
> mg <- function(v)
{
exp(sum(log(v))/length(v))
}
> mg(1:20)
[1] 8.304361
```

Tal como la hemos definido, la función no verifica si los argumentos son positivos antes de calcular el logarítmo:

```
> mg(c(-2,-1,1,2))
[1] NaN
Warning message:
Se han producido NaNs in: log(x)
```

Instrucciones Condicionales

Vamos a corregir esto introduciendo una instrucción condicional que determine si los argumentos son positivos

```
> mg <- function(v)
{
if(any(v <= 0))
  {
  print('Todos los numeros deben ser
  positivos'); return(NULL)
  }
else return(exp(sum(log(v))/length(v)))
}
> mg(c(-2,-1,1,2))
[1] "Todos los numeros deben ser positivos"
NULL
```

Ejercicio

Ejercicio 5.9

1. Dentro de un bucle queremos incluir una instrucción que imprima el índice correspondiente, pero sólo cada cinco iteraciones. Escriba una instrucción que haga esto usando `if`.

Instrucciones Condicionales

Existe en `R` una versión vectorizada de esta instrucción condicional, cuya sintaxis es

`ifelse` (*condición*, *expresión 1*, *expresión 2*)

El resultado de esta expresión es un vector con elementos `a[i]` si `condición[i]` es cierta y `b[i]` si es falsa.

```
  > (c <- ifelse(x > 2, 'a', 'b'))
[1] 'b' 'b' 'a'
```

Instrucciones Condicionales

Como ejemplo vamos a usar esta instrucción para generar una muestra de una mezcla de distribuciones gaussianas. Queremos que con probabilidad 0.4 la población provenga de una normal con media -2 y varianza 2, y con probabilidad 0.6 de una normal con media 1 y varianza 1. Las instrucciones son las siguientes:

```
> uu <- runif(100)
> norm1 <- rnorm(100,mean=-2, sd=sqrt(2))
> norm2 <- rnorm(100,mean=1)
> vv <- ifelse(uu < 0.4, norm1,norm2)
```

Instrucciones Condicionales

Hacemos la observación de que la expresión `ifelse` puede generar alertas sobre valores `NA` o `NaN` aún cuando en la expresión final estos valores no aparezcan. Por ejemplo, en una expresión como

```
> ifelse( y <= 0, 0, y*log(y))
```

Los tres argumentos de la expresión se evalúan, y por lo tanto aparecerá un alerta si hay algún valor menor o igual que 0 al evaluar el logaritmo, aún cuando este valor no aparece en la expresión final:

```
> y <- -1:3
> aa <- ifelse(y <= 0, 0, y*log(y))
Warning message:
Se han producido NaNs in: log(x)
> aa
[1] 0.000000 0.000000 0.000000 1.386294
3.295837
```

Ejercicio

Ejercicio 5.10

1. Genere una muestra de 10 números al azar de la distribución uniforme en $[-1, 1]$. Usando la instrucción `ifelse` cree un vector que tenga el signo de cada elemento de la muestra anterior.

Instrucciones Condicionales

Si es necesario condicionar por varias condiciones una posibilidad es usar varias instrucciones `if` sucesivas. Veamos como ejemplo el siguiente caso. Tenemos un vector de números entre -1 y 2 y queremos evaluar la siguiente función:

$$f(x) = \begin{cases} 0 & \text{si } x \leq 0, \\ -\log(x) & \text{si } 0 < x < 1, \\ \log(x) & \text{si } 1 \leq x \leq 2. \end{cases}$$

Instrucciones Condicionales

Para esto podemos usar dos instrucciones tipo `if` de la siguiente manera:

```
> zz <- runif(100,-1,2)
> ab <- ifelse(zz<0,0,ifelse(zz<1,-log(zz),
    log(zz)))
Warning messages:
1: Se han producido NaNs in: log(x)
2: Se han producido NaNs in: log(x)
```

Instrucciones Condicionales

Veamos un segundo ejemplo, modificando la función `mg` que construimos de modo que verifique también si todas las componentes del vector son numéricas.

```
> mg <- function(v)
{
if(!is.numeric(v))
  {print('Argumento debe ser numerico')
  return(NULL)}
if(any(v <= 0))
  {print('Todos los numeros deben ser
positivos')
  return(NULL)}
else return(exp(sum(log(v))/length(v)))
}
> mg(letters)
[1] 'Argumento debe ser numerico'
NULL
```

Instrucciones Condicionales

Una alternativa a las instrucciones tipo `if` anidadas es `switch`. Veamos un ejemplo tomado de la ayuda de `switch`. Queremos definir una función para obtener una medida de centramiento de una muestra. Esta función puede ser la media, la mediana o la media recortada en 10 %.

Instrucciones Condicionales

```
> centrar <- function(x,type)
{
switch(type, media=mean(x),
mediana = median(x),
recortada = mean(x,trim=.1))
}
> x <- rcauchy(10)
> centrar(x,'media')
[1] 1.486102
> centrar(x,'mediana')
[1] -0.2941052
> centrar(x,'recortada')
[1] 0.4253272
> centrar(x,'algo')
NULL
```

Instrucciones Condicionales

Si queremos que alguna de las alternativas se tome por defecto cuando no se especifica alguna otra o cuando la especificación no concuerda con ninguna de las posibilidades, hay que escribir la opción por defecto de última, sin especificar nombre:

```
centrar <- function(x,type)
{
switch(type, media=mean(x),
mediana = median(x),
recortada = mean(x,trim=.1),
mean(x))
}
```

Instrucciones Condicionales

```
> centrar(x,”)
[1] 1.486102
> centrar(x,’algo’)
[1] 1.486102
> centrar(x)
Erro en switch(type, media = mean(x), mediana
= median(x), recortada = mean(x, :
el argumento ’type’ está ausente, sin default
```

Instrucciones Condicionales

Si las alternativas no tienen nombre, la función `switch` puede seleccionar la alternativa que se desea por medio de números.

```
> mtr <- rnorm(50)
> mean(mtr)
[1] 0.1496272
> median(mtr)
[1] 0.08093262
> switch(2, mean(mtr), median(mtr))
[1] 0.08093262
```

Ejercicio

Ejercicio 5.11

1. Defina una función en `R` que calcule la siguiente función

$$f(x) = \begin{cases} 0 & \text{para } x \leq -1 \\ (x+1)/2 & \text{para } -1 < x \leq 0 \\ (x^2+1)/2 & \text{para } 0 < x \leq 1 \\ 1 & \text{para } x \geq 1 \end{cases}$$

...

Al definir una función es posible definir el argumento `...` en cualquier lugar de la lista de parámetros. Al usarlo la función puede aceptar cualquier número de argumentos. El uso de la función varía dependiendo de dónde se usen los puntos suspensivos. Veamos dos ejemplos

```
> f1 <- function (x, ...)  instrucciones
> f2 <- function (..., x)  instrucciones
```

En el primer caso, si escribimos

```
> f1(3)
```

a `x` se le asigna el valor `3`, pero si escribimos

```
> f2(3)
```

el valor `3` es parte de los punto suspensivos y `x` no tiene valor. Es necesario escribir

```
> f2(x=3)
```

para que a `x` se le asigne el valor `3`.

...

Veamos un ejemplo del uso de los puntos suspensivos.Vamos a hacer una función que grafica los valores de una función en un intervalo

```
> plot.f <- function(f,a,b,...)
  {
  xvals <- seq(a,b,length=100)
  plot(xvals, f(xvals), type='l', ...)
  }
> plot.f(sin,0,2*pi)
> plot.f(sin,0,2*pi,lty=4)
> plot.f(abs,-1,1,lty=4)
```

...

Como segundo ejemplo vamos a modificar la función `centrar` que creamos antes eliminando la opción `recortada` e incluyendo la opción de un recorte en `mean` usando `...`

```
> centrar <- function(x,type=mean,...)
{
switch(type, media=mean(x,...),
mediana = median(x),
mean(x,...))
}
> centrar(x,")
[1] 3.280941
> centrar(x,'mean',trim=0.2)
[1] 3.302633
```

Ejercicio

Ejercicio 5.12

1. Modifique la función `pazar` de modo que también haga gráficas de la trayectoria. Incluya en la definición la posibilidad de especificar parámetros a la función gráfica.
2. Modifique la función del ejercicio 4 de modo que también haga la gráfica de la función. Incluya en la definición la posibilidad de especificar parámetros a la función gráfica.

Funciones Anónimas

En ciertas ocasiones es útil definir una función sin asignarle un nombre. En general se trata de funciones cortas que se utilizan dentro de otra función. Por ejemplo, para calcular el valor de $x^2y + xy^2$ para todas las combinaciones de valores de x y y, podemos usar la función `outer` de la siguiente manera:

```
> x <- 1:5
> y <- 2:4
> f <- function(x,y) x^2*y + x*y^2
> outer(x,y,f)
     [,1] [,2] [,3]
[1,]    6   12   20
[2,]   16   30   48
[3,]   30   54   84
[4,]   48   84  128
[5,]   70  120  180
```

Funciones Anónimas

Sin embargo, si la función `f` no tiene ninguna utilidad posterior, es posible usar una función anónima en el interior de `outer`

```
> outer(x,y,function(x,y) x^2*y + x*y^2)
     [,1] [,2] [,3]
[1,]    6   12   20
[2,]   16   30   48
[3,]   30   54   84
[4,]   48   84  128
[5,]   70  120  180
```

La Función `mapply`

La función `mapply` es una función de la familia `apply` que actua de manera multidimensional. Su sintaxis es

```
> mapply(FUN, ...)
```

El resultado es la aplicación de la función `FUN` a los primeros elementos de todos los argumentos contenidos en `...`, luego a todos los segundos elementos, y así sucesivamente. Por ejemplo, si `x` y `y` son dos vectores, `mapply(FUN, x,y)` devuelve `FUN(v[1], w[1]), FUN(v[2], w[2])`, etc.

La Función `mapply`

```
> mapply(rep, 1:4, 4:1)

> mapply( seq, 1:6, 6:8)

> aa <- lapply(rep(12,4), sample, x=1:100)
> mapply(mean, aa, 0:3/10)
```

La Función `replicate`

Esta función es adecuada para las simulaciones. Por ejemplo, si la función `FUN` hace los cálculos necesarios en una simulación, para obtener los resultados de 10000 simulaciones podemos escribir

```
> replicate(10000, FUN(...))
```

Por ejemplo, para obtener 10 muestras aleatorias de tamaño 15 de los números del 1 al 100 escribimos

```
> replicate(10, sample(1:100, 15))
```

La Función `replicate`

Para dar un ejemplo del uso de la función `replicate` vamos a explorar por simulación el sesgo de la media muestral como estimador de la media de una población normal. Para esto vamos a calcular la media de una cantidad grande de muestras normales de tamaño 100.

```
> ff <- function(n, mean, sd)
     mean(rnorm(m, mean=mean, sd=sd))
> res <- replicate(10000, ff(100, 0, 1))
> hist(res)
> mean(res)
```

Ejercicio

Ejercicio 5.13

1. Cree los siguientes vectores en `R`:

   ```
   A <- c('Antonio', 'Maria', 'Ana',
   'Manuel')
   B <- c('Martinez', 'Perez', 'Gomez',
   'Lopez')
   ```

   Usando las funciones `mapply` y `paste` construya un vector que una nombres y apellidos, componente a componente.
2. Usando la función `replicate` calcule una aproximación de π por el método Montecarlo, como estudiamos la sesión pasada.

Verificación

Para evitar errores, a veces es necesario verificar si un parámetro de una función ha sido omitido. Para esto se puede usar la función `missing`. El resultado de la función es un valor lógico, `T` si el parámetro fue omitido y `F` si fue incluido. Una línea típica en el código de una función para verificar si el parámetro x fue incluido o no es la siguiente

```
If (missing (x)) print (”no se especifico el valor de x.”)
```

Si una variable tiene un valor por defecto y no fue cambiado al llamar a la función, el resultado de `missing` es `T`.

Outline

cat

Además de la función `print`, que escribe el valor de una variable, `R` tiene la función `cat` que tiene mayor versatilidad. Veamos algunos ejemplos

```
> x <- 2*(1:5)
> cat (x)
2 4 6 8 10
> cat("Hola")
Hola
> y <- 7
> cat("El valor de y es", y, ".")
El valor de y es 7 .
> cat('El valor de y es', y,'\b.')
El valor de y es 7.
```

cat

Esta función tiene una serie de caracteres de control que permiten modificar la presentación de las líneas. En la última instrucción usamos `\b` que retrocede el cursor un espacio. En el próximo ejemplo usamos `\n` que permite dividir una expresión en dos líneas.

```
> cat ("Esta es una prueba.\nSegunda linea")
Esta es una prueba.
Segunda linea
```

Observe que no se debe dejar espacio después de la expresión `\n`.

cat

Control	Efecto
\n	Línea nueva
\t	Tabulador
\\	Backslash (\)
\”	”
\’	’
\#	#
\b	Regresar un espacio
\r	Return

Tabla 4.1 Expresiones de control en R

cat

Hay varias maneras de usar esta función para interacutar con el usuario buscando la información necesaria para algún procedimiento. En el primer ejemplo le solicitamos al usuario introducir alguna frase, Al introducir una frase vacía el proceso termina.

```
> texto <- c()
> repeat {
cat('Introduzca una frase (frase vacia
termina)')
fr <- readLines(n=1)
if (fr==”) break else texto <- c(texto,fr)
}
```

cat

Al ejecutar estas instrucciones obtenemos

```
Introduzca una frase (frase vacia termina)
'Primera frase'
Introduzca una frase (frase vacia termina)
'Segunda'
Introduzca una frase (frase vacia termina)

> texto
[1] "'Primera frase' " "'Segunda' "
```

cat

Otra posibilidad es usar la función `cat` combinada con la función `scan`. Veamos un ejemplo que muestra además el uso de `next` y `break` en un ciclo.

```
> pos <- numeric(0)
> repeat {
cat('Introduzca un numero positivo (cero
    termina)')
num <- scan(n=1)
if (num < 0) next
if (num == 0) break
pos <- c(pos, num)
}
```

cat

```
  Introduzca un numero positivo (cero termina)
1: 3.5
  Read 1 item
  Introduzca un numero positivo (cero termina)
1: -2
  Read 1 item
  Introduzca un numero positivo (cero termina)
1: 5.1
  Read 1 item
  Introduzca un numero positivo (cero termina)
1: 0
  Read 1 item
  > pos
  [1] 3.5 5.1
```

Ejercicio

Ejercicio 5.14

1. Usando los comandos `cat()` y `scan()` escriba una función que al correrla escriba en la pantalla
   `Escriba los datos.`
   `Para terminar apriete enter dos veces`
   luego lea lo que el usuario escriba y al terminar lo imprima en pantalla.