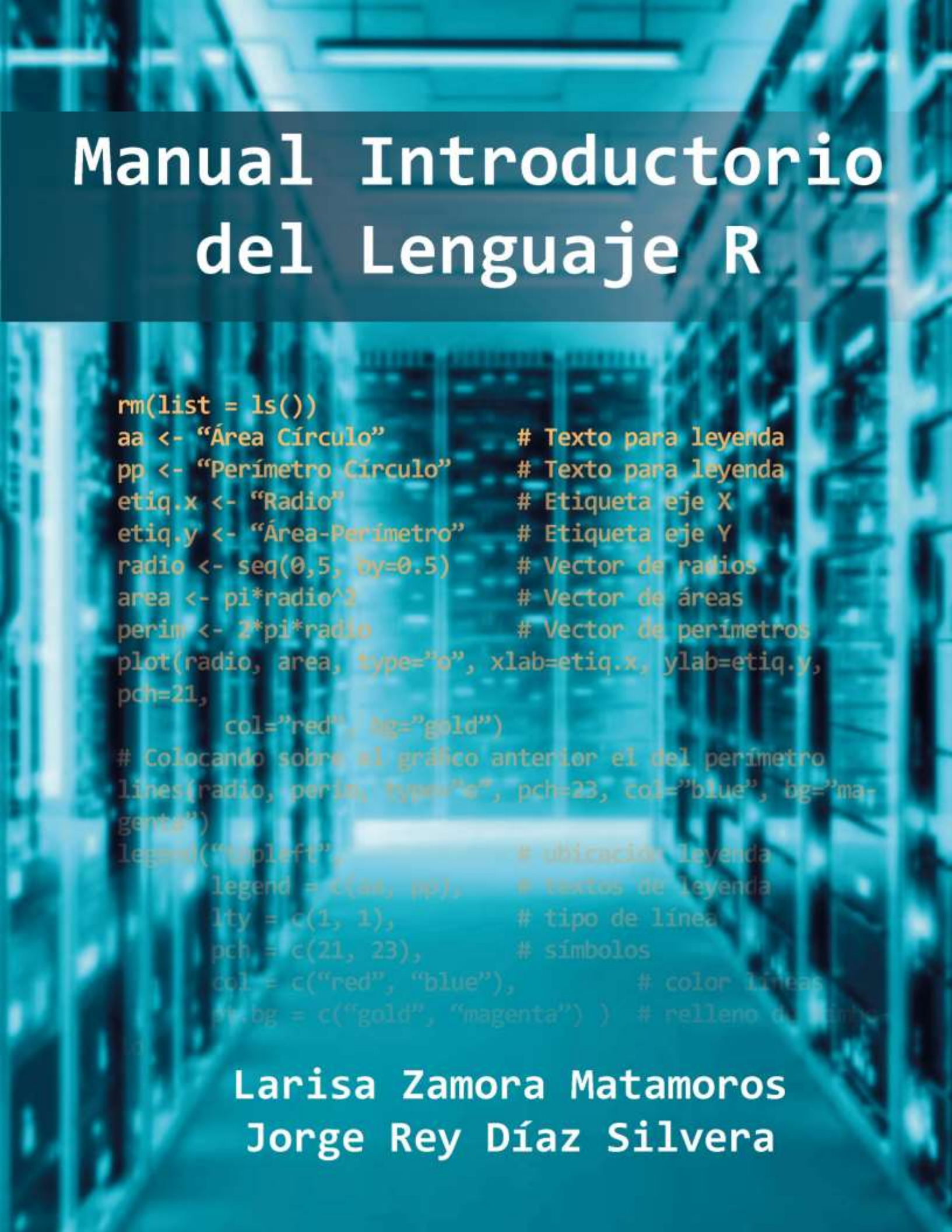
Manual Introductorio del Lenguaje R



Larisa Zamora Matamoros
Jorge Rey Díaz Silvera

Manual Introductorio del Lenguaje R

Dra. C. Larisa Zamora Matamoros

Dr. C. Jorge Rey Díaz Silvera

Edición: Lic. Nora Nuñez Gollot
Composición: Alina Montoya Revilla
Diseño de cubierta: MSc. Lidia de las Mercedes Ferrer Tellez

ISBN: 978-959-207-738-6

Ediciones UO
Ave. Patricio Lumumba No. 507, e/ Ave. de Las Américas y Calle 1ra,
Reparto Jiménez. Consejo Popular José Martí Norte.
Santiago de Cuba, Cuba. CP: 90500
Telf.: +53 22644453
e-mail: jdp.ediciones@uo.edu.cu
edicionesuo@gmail.com

Prólogo

El lenguaje R se emplea fundamentalmente en cálculos estadísticos, análisis de datos (big data) y su visualización, convirtiéndose en los últimos años en uno de los lenguajes más prominentes para la ciencia de los datos y la estadística. No por gusto ha estado evaluado en los últimos años entre los 10 mejores lenguajes en el ranking de la IEEE (Institute of Electrical and Electronics Engineers). Los programas en R están constituidos por secuencias de comandos o instrucciones, llamadas script, en un archivo a ser interpretado.

El presente material ha sido concebido para ser utilizado en cursos optativos por estudiantes de las carreras de Licenciatura en Matemática, Licenciatura en Ciencia de la Computación y afines, preferentemente con conocimientos elementales sobre conceptos básicos de programación, de probabilidades y estadística, y alguna experiencia en el procesamiento de datos, apoyado en su carácter didáctico y el detalle empleado en las explicaciones y las soluciones a los ejemplos. También puede ser usado como material de consulta en la docencia o investigación en otras disciplinas donde sea necesario el desarrollo de programas basado en big data, su análisis y visualización.

En este manual se enseña a desarrollar software basado en el sistema base del lenguaje y al mismo tiempo brinda herramientas metodológicas para poder extender las potencialidades del lenguaje con el empleo de los múltiples paquetes de datos que se han desarrollado para el lenguaje y que están a disposición de forma libre en muchos repositorios mundiales.

El manual se ha dividido en 11 unidades temáticas que se describen sintéticamente a continuación:

Unidad 1: resumen histórico del surgimiento del lenguaje R, su instalación, empleo de RStudio como ambiente de desarrollo integrado para facilitar el trabajo con el lenguaje y las primeras indicaciones para escribir programas en R.

Unidad 2: elementos básicos del lenguaje, uso de operadores aritméticos, lógicos y de asignación, funciones de impresión y aritméticas predefinidas, números aleatorios y el carácter e importancia de la vectorización en el lenguaje.

Unidad 3: descripción general de las clases de objetos en R y sus atributos, y los mecanismos de coerción de datos.

Unidad 4: estudio de estructuras de datos homogéneas como vectores, matrices, arreglos y factores, con su descripción y operatoria (incluida aritmética de conjuntos).

Unidad 5: estudio de estructuras de datos heterogéneas, como listas y data frames.

Unidad 6: lectura/escritura de datos desde ficheros de texto, incluyendo la importación/ exportación de datos con Excel, SPSS, Minitab y similares, acceso a ficheros que acompañan al sistema base de R y procesamiento de datos .rds y .RData.

Unidad 7: estructuras de control del lenguaje (alternativas, ciclos, alternativa vectorizada y selección de casos).

Unidad 8: definición de funciones propias por el programador, que incluye la definición de una función en R, mecanismos de transferencia de parámetros, nombres de funciones como variables, funciones como argumentos de funciones, reglas de alcance y recursividad.

Unidad 9: funciones de transformación y agregación de datos, la familia apply(), funciones anónimas, funciones del análisis matemático y numérico, funciones que sobrecargan operadores binarios y funciones para importar y exportar código.

Unidad 10: funciones del sistema base para elaborar diversos gráficos (ploteo de curvas, diagramas de barras, histogramas, diagramas de cajas y bigotes, diagramas de pastel) y la importación/exportación de gráficos.

Unidad 11: instalación y elaboración de paquetes, objetos de tipo expression y ejecución con código C++ incrustado con el Rcpp.

Es nuestro mayor deseo que al final del estudio de este manual, que necesariamente debe verse complementado por otros libros de texto y manuales de referencia, sirva a los propósitos arriba mencionados y motive a una profundización imprescindible en sus campos de empleo, todos con gran actualidad.

Los autores.

Contenido

Unidad 1. Introducción al lenguaje R

1.1 Breve historia del surgimiento del lenguaje R

R fue creado en **1992** en **Nueva Zelandia** por **Ross Ihaka** y **Robert Gentleman,** del Departamento de Estadística de la Universidad de Auckland en Nueva Zelanda. La intención inicial con R, era hacer un lenguaje didáctico para ser utilizado en el curso de Introducción a la Estadística de dicha Universidad. Para ello decidieron adoptar la sintaxis del **lenguaje S** desarrollado por John Chambers, de Bell Telephone Laboratories, en **1976**. Como consecuencia la sintaxis es similar al lenguaje S, pero la semántica, aparentemente parecida a la de S, es en realidad sensiblemente diferente, sobre todo en los detalles un poco más profundos de la programación. A modo de broma Ross y Robert comienzan a llamar "R" al lenguaje que implementaron, por las iniciales de sus nombres, y desde entonces así se le conoce en la muy extendida comunidad amante de dicho lenguaje.

Luego de la creación de R (en 1992), se da un primer anuncio al público del software R en **1993**. En el año de **1995** Martin Mächler, de la Escuela Politécnica Federal de Zúrich, convence a Ross y Robert a usar la Licencia GNU[1] para hacer de R un software libre, por lo que a partir de **1997**, R forma parte del proyecto GNU. El término software libre se refiere a la libertad de los usuarios para ejecutar, copiar, distribuir, estudiar, cambiar y mejorar el software.

Con el propósito de crear algún tipo de soporte para el lenguaje, en **1996** se crea una lista pública de correos; sin embargo, debido al gran éxito de R, los creadores fueron rebasados por la continua llegada de correos. Por esta razón, se vieron en la necesidad de crear, en **1997**, dos listas de correos: **R-help** y **R-devel**, que son las que actualmente funcionan para responder las

[1] El proyecto GNU se inició en 1984 con el propósito de desarrollar un sistema operativo compatible con Unix que fuera de software libre. Para más información sobre el proyecto GNU consultar http://www.gnu.org

diversas dudas que los usuarios proponen en los diversos asuntos relativos al lenguaje. Además, se consolida el grupo núcleo de R, donde se involucran personas asociadas con S-PLUS, con la finalidad de administrar su código fuente. No fue hasta **febrero 29 del 2000**, que se considera al software completo y lo suficientemente estable para liberar la **versión 1.0**.

Muchos consideran al lenguaje R como un sistema estadístico, aunque es preferible catalogarlo como un ambiente de trabajo en el que se aplican técnicas estadísticas. Algunas de ellas implementadas en el núcleo de R (el sistema base); otras muchas disponibles en la CRAN[2], como paquetes (packages), las cuales han sido desarrolladas por usuarios y programadores alrededor del mundo. Actualmente existen miles de paquetes disponibles en la CRAN y otros en redes personales.

El sistema R está dividido en 2 partes conceptuales:

1) El sistema base de R, que es el que se puede bajar de la CRAN.
2) Otras contribuciones al lenguaje en forma de módulos o paquetes de programación.

El **sistema base de R** contiene el paquete básico que se requiere para su ejecución y la mayoría de las funciones fundamentales. Los otros paquetes contenidos en la "base" del sistema incluyen a ***utils, stats, datasets, graphics, grDevices, grid, tools, parallel, compiler, splines, tcltk, stats4***, etc.

La capacidad gráfica de R es muy sofisticada y mejor que la de la mayoría de los paquetes estadísticos. Cuenta con varios paquetes gráficos especializados, por ejemplo, ***GISTools***, para crear y trabajar con sistemas de información geográficos y análisis espacial, permitiendo dibujar mapas, hacer contornos sobre mapas en distintas proyecciones, graficado de vectores, etc.

---

[2] Acrónimo de Comprehensive R Archive Network. En la web https://cran.r-project.org se encuentra disponible toda la información acerca de R.

También existen paqueterías que permiten manipular y crear datos en distintos formatos como netCDF[3], Matlab, Excel entre otros.

Características de la programación en R

- Soporta la programación procedural, que está basada en la ejecución de sentencias (en particular asignaciones) que influencia el subsiguiente cálculo a través de cambios de valores en la memoria, y la programación orientada a objetos, donde se representa el cómputo como interacciones entre objetos semi-independientes, cada uno de los cuales posee su propio estado interno y un conjunto de procedimientos que administran ese estado.
- Soporta aspectos de la programación funcional. La característica esencial de la programación funcional es que los cálculos se ven como una función matemática que hace corresponder entradas y salidas.

  Los programas escritos en un lenguaje funcional puro están constituidos únicamente por definiciones de funciones, entendiendo éstas, no como subprogramas clásicos de un lenguaje imperativo, sino como funciones puramente matemáticas, en las que se verifican ciertas propiedades como la transparencia referencial. La transparencia referencial, significa que el significado de una expresión depende únicamente del significado de sus subexpresiones o parámetros, no depende de cálculos previos ni del orden de evaluación de sus parámetros o subexpresiones, por tanto, implica la carencia total de efectos colaterales, pues no considera la existencia de variables globales.

[3] Conjunto de bibliotecas de software y formatos de datos auto descriptivos e independientes de la máquina que soportan la creación, el acceso y el intercambio de datos científicos orientados a array.

Otras características propias de estos lenguajes (consecuencia directa de la ausencia de estado de cómputo y de la transparencia referencial) son la no existencia de asignaciones de variables y la falta de construcciones estructuradas como la secuencia o la iteración (no hay for, while, etc.). Esto obliga en la práctica, a que todas las repeticiones de instrucciones se lleven a cabo por medio de funciones recursivas.

Existen dos grandes categorías de lenguajes funcionales: los funcionales puros y los funcionales híbridos. La diferencia entre ambos radica en que los lenguajes funcionales híbridos son menos dogmáticos que los puros, al incluir conceptos tomados de los lenguajes imperativos, como las secuencias de instrucciones, la iteración o la asignación de variables. En particular R es un lenguaje funcional híbrido.

- La programación en R permite la creación de paquetes. Los mismos son útiles para encapsular colecciones de funciones de R en una única unidad.
- Permite gestionar bases de datos, importar y exportar datos, etc.
- Es un lenguaje interpretado. El intérprete de R procesa el programa fuente instrucción a instrucción para determinar las operaciones a ser realizadas y las mismas son ejecutadas. Una vez que una instrucción es completamente interpretada y ejecutada se pasa a la siguiente.
- Posee muy buena aritmética de vectores y matrices, con una cantidad grande de operadores.
- Realiza un manejo y almacenamiento de datos eficiente.
- Posee buenas características para la salida de reportes gráficos o impresos.

1.1.1 Ventajas del R

- Es muy bueno para el trabajo con la estadística, el análisis de datos, el análisis financiero, el marketing, entre otros campos.
- Es muy útil para el **trabajo interactivo** (participativo, de intercambio), pero también es un poderoso **lenguaje de programación** para el desarrollo de nuevas herramientas.
- Es un lenguaje independiente de plataforma. Programas desarrollados en R pueden ser ejecutados en ambientes de trabajo de cualquier sistema operativo, gozando de portabilidad.
- Posee una **comunidad internacional muy activa de programadores y empleadores**, por lo que, haciendo las preguntas correctas, se encuentra rápidamente la solución a los problemas que se presenten en el ámbito de su programación. Estas características han promovido que el número de sus usuarios en el área de las ciencias se incremente enormemente.
- Es un **software libre** y **open source**[4] hacen a R un lenguaje atractivo, debido a que no hay que preocuparse por licencias y cuenta con la libertad que garantiza GNU.

Debido a su estructura, **R consume mucho recurso de memoria**, por lo tanto, si se utilizan datos de gran tamaño el programa se haría muy lento o, en el peor de los casos, no podría procesarlos. Sin embargo, en la mayoría de los casos los problemas que pudieran surgir con referencia a la lentitud en la ejecución del código tienen solución, principalmente empleando la vectorización del código siempre que sea posible y aprovechar el procesamiento paralelo.

---

[4] El **open source** o **código abierto** es el software desarrollado y distribuido libremente. Se focaliza más en los beneficios prácticos (acceso al código fuente) que en cuestiones éticas o de libertad que tanto se destacan en el software libre.

1.1.2 Manuales de R

Disponible en http://cran.r-project.org se puede encontrar:

- An Introduction to R.
- R Reference.
- R Data Import/Export.
- R Language Definition.
- Writing R Extensions.
- R Internals.
- R Installation and Administration.
- Sweave User Manual

Estos materiales también están disponibles a través de las ayudas que brindan diversas versiones del lenguaje R o de GUI's montadas sobre el mismo, como RStudio.

1.1.3 Instalación de R

1. Entrar en la página de la CRAN, https://cran.r-project.org/.
2. Elija en la parte izquierda de la pantalla la opción **R Binaries**, en el apartado de *Software.*
3. Aparecerán en la parte derecha de la pantalla una serie de carpetas con nombres de sistemas operativos. Elija **Windows**.
4. Aparecerá un fichero ReadMe (conviene leerlo) y nuevas carpetas. Debe entrar en la carpeta **base**.
5. Dar clic en **Download R yyy for Windows**, donde **yyy** informa el nombre de la versión más actual que será descargada. Una vez bajada se ejecuta, la versión de 32 bits o la de 64 bits, según sea su instalación de Windows. Se instalará el programa R, así como una serie de librerías que amplían el número de funciones disponibles.

Una vez instalado el R en su computadora, el programa se puede iniciar corriendo el fichero ejecutable correspondiente. El cursor, que por defecto es el símbolo ">", indica que está listo para recibir un comando.

1.2 RStudio

RStudio es un **entorno de desarrollo integrado** (IDE) libre y de código abierto que auxilia al programador al escribir, editar, ejecutar y revisar su programa en R, además de brindar múltiples conexiones a paquetes y otras herramientas auxiliares disponibles en Internet, desarrollado por J. J. Allaire, creador del lenguaje de programación ColdFusion. Está escrito en el lenguaje de programación C++ sobre la plataforma de desarrollo de Qt, que emplea también para desarrollar su interfaz gráfica de usuario.

Esta IDE está disponible en dos ediciones: **RStudio Desktop**, donde el programa se ejecuta localmente como una aplicación de escritorio normal y **RStudio Server**, que permite acceder a RStudio utilizando un navegador web mientras se está ejecutando en un servidor Linux remoto. Las distribuciones pre configuradas de RStudio Desktop están disponibles para Windows, macOS y Linux.

El trabajo en RStudio comenzó alrededor de diciembre de **2010** y la primera versión beta pública (v0.92) fue anunciada oficialmente en febrero de **2011**. La versión 1.0 fue lanzada el **1 de noviembre de 2016**.

La dirección web para acceder a RStudio es www.rstudio.com/products/rstudio/download

1.2.1 Instalación de RStudio

1. Localice el fichero de instalación de RStudio (ejemplificamos en este caso con **RStudio-1.3.1093.exe**) y dé doble clic sobre él. Se abre el asistente de instalación de RStudio:

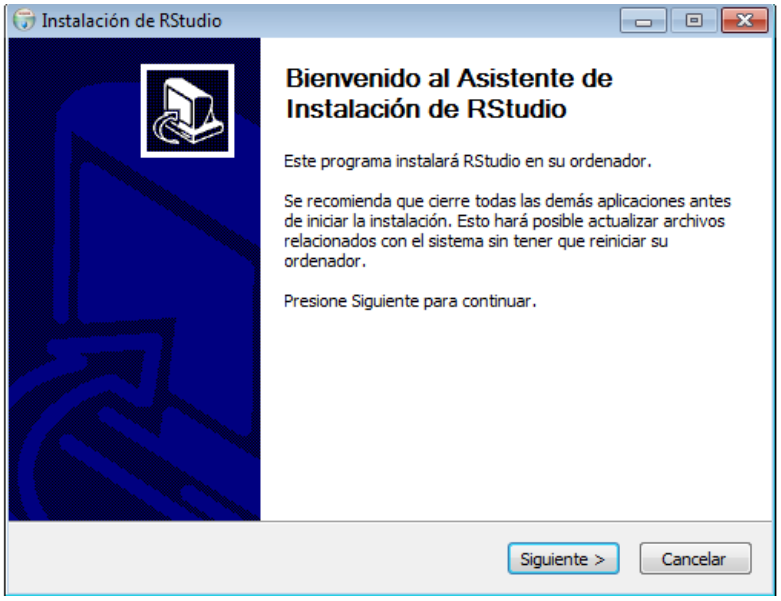


Figura 1. Cuadro de instalación de RStudio.

2. Dar clic en el botón **Siguiente >**. Aparece la pantalla:

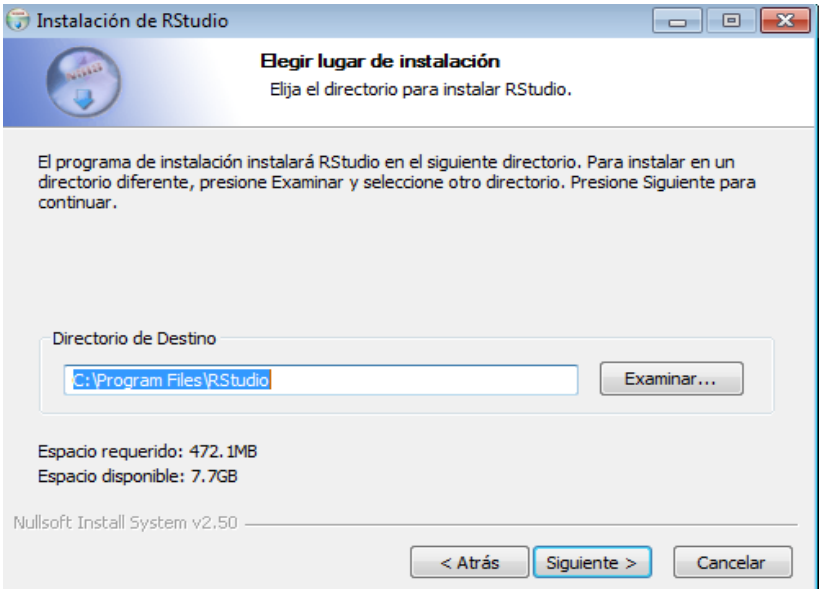


Figura 2. Cuadro de selección del directorio de trabajo.

3. Se elige el lugar de instalación (directorio de destino). Se propone un camino por defecto, aunque puede elegir otro pulsando el botón **Examinar...**. Una vez elegido el lugar de instalación dar clic en **Siguiente >**. Aparece:

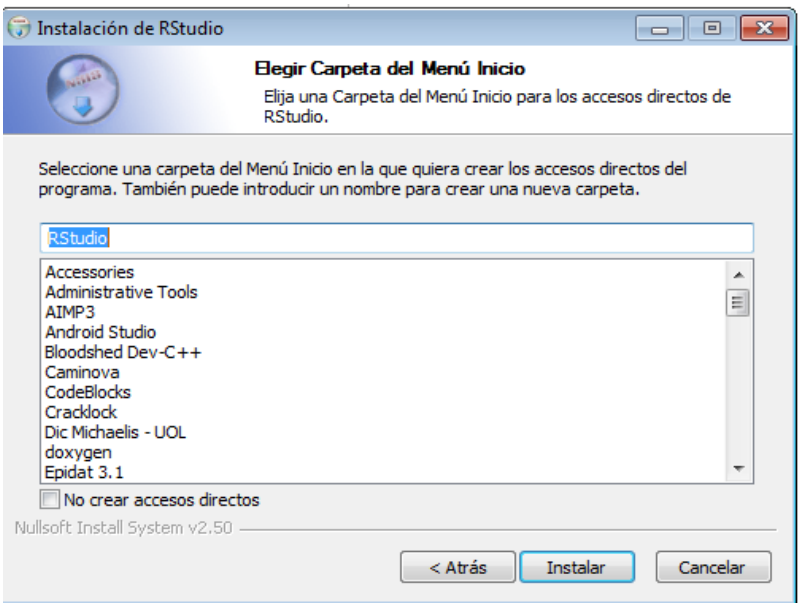


Figura 3. Cuadro para crear los accesos directos del RStudio.

para elegir la carpeta de destino, que puede ser la que se propone (RStudio) o alguna del menú de inicio que elija de la lista que se muestra en la figura 3 u otra con un nombre que remplace RStudio. Luego dar clic en **Instalar**.

1.3 Para comenzar a explotar el sistema de trabajo con R

Una vez que se han instalado R y RStudio:

1. Cree una carpeta o directorio de trabajo, en la cual guardará los ficheros, por ejemplo, **"Asignatura_ Optativa_R"**.
2. Cargue el RStudio y seleccione la opción **"Session"**.
3. Se desplegará una pestaña en la cual deberá seleccionar la opción **"Set Working Directory"**, con lo cual se desplegará una nueva pestaña.
4. Seleccione la opción **"Choose Directory"** y luego la carpeta o directorio definido en el paso 1. En la consola del RStudio aparecerá, por ejemplo,

   > setwd("E:/Larisa/Asignatura_Optativa_R")

5. Si en la carpeta aún no tiene ningún fichero, elegirá la opción **"New File"**. Si tiene ya algún fichero creado y desea continuar trabajando con él, elegirá la opción **"Open File**".
6. Con la opción **"New File"** se despliega una nueva pestaña, en la cual elegirá la opción **"R Script[5]"** y la parte izquierda de la pantalla se dividirá en dos. En la parte superior puede empezar a escribir su programa.

1.3.1 Primeros programas en R usando RStudio

El trabajo con R se simplifica bastante usando RStudio. En el presente material se emplea la versión 4.0.2 de R y la 1.3.1093 de RStudio.

Una posible vista de apertura de RStudio se muestra en la figura 4, en la cual se observa que RStudio cuenta con varios paneles o ventanas de trabajo:

a) Panel fuente o editor de script: Es el sitio donde se escribe, edita y salva el código fuente (script) del programa, el cual puede ser ejecutado completamente, en parte o línea a línea apretando el botón **Run,** el cual se encuentra en la parte superior del panel, lo que provoca que el código seleccionado sea ejecutado en el panel de consola.

   Observe que este panel no existe por defecto cuando se abre RStudio; aparece cuando se abre un script de R, por ejemplo, mediante **File → New File → R Script**.

   Se recomienda guardar el nuevo script creado con un nombre adecuado, seleccionando **File → Save as** en el menú principal. De inmediato se renombra el tabulador correspondiente en el panel fuente y le agrega la extensión **.R**. Entonces el fichero aparecerá (nombrefichero.R) en el panel de ficheros.

---

[5] Script: En informática, un *script*, fichero de órdenes, fichero de procesamiento por lotes o, cada vez más aceptado en círculos profesionales y académicos, guión, es un programa usualmente simple, que por lo regular se almacena en un fichero de texto plano. El uso habitual de los guiones es realizar diversas tareas como combinar componentes, interactuar con el sistema operativo o con el usuario.

R-Studio salva el espacio de trabajo, el cual es el ambiente en que se está trabajando, en el directorio de trabajo, que por defecto es **C:\Users\<User Name>\Documents**, pero que el programador puede direccionar donde le sea más conveniente.

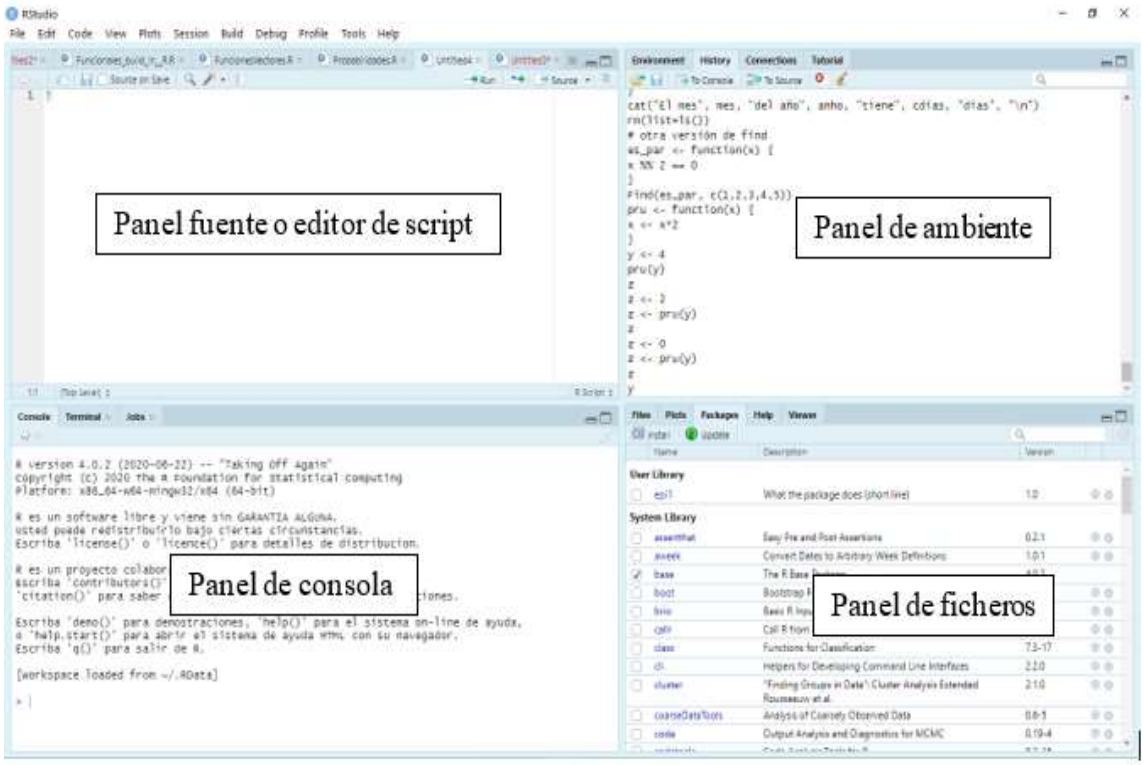


Figura 4. Paneles de trabajo del RStudio.

b) Panel de consola (**Console**): Es el panel donde se corre el código de R. Un elemento distintivo de ese panel es su señalizador (prompt) que es un signo de mayor (>). Un ejemplo de salida en el panel de consola, en el que se construyen dos vectores y luego se comparan para investigar si son iguales y se suman, respectivamente, se muestra a continuación:

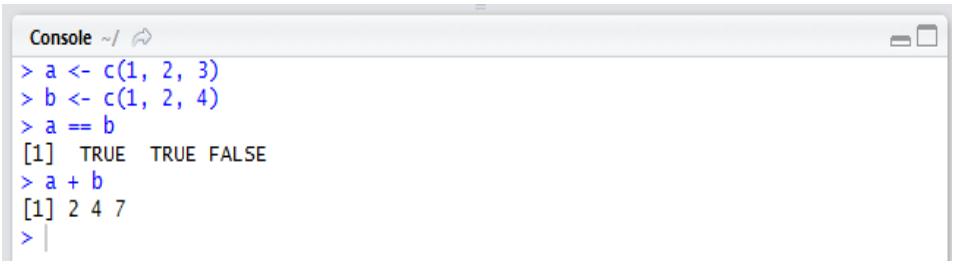


Figura 5. Panel de consola.

Una forma de correr código en R es escribiéndolo en el panel de consola al lado del prompt. Cualquier código entrado aquí es procesado línea a línea. Si se teclea un comando incompleto y presiona Enter (↵), R mostrará un indicador de comandos (+), que significa que está esperando que se teclee el resto del comando. Entonces debe terminar de escribir el comando o dar escape (**Esc**) para volver a empezar.

Ejemplo:

```
> 5-
+ 4
[1] 1
```

El panel de consola es ideal para probar ideas interactivamente antes de salvar códigos finales en el panel fuente o de script. Sin embargo, escribir comandos en la consola no es lo usual cuando se trabaja con datos o se quiere escribir un programa, lo cual se hace en el panel de scripts.

c) Panel de ambiente: Muestra las variables y las funciones que el programador ha creado y están disponibles para su uso (figura 6). Por defecto, muestra las variables en el ambiente global, o sea, la zona de trabajo usuaria en la que está trabajando.

Cada vez que se crea un nuevo objeto (una variable o función), se agrega una nueva entrada al panel de ambiente, que indica el nombre de la variable y una descripción breve

de su valor. Cuando se modifica el valor de un objeto o se elimina, se modifica el ambiente y el panel de ambiente refleja tal cambio.

Este panel tiene cuatro tabuladores: ambiente, historia, conexiones y tutorial. El tabulador de ambiente (**Environment**) registra los objetos creados.

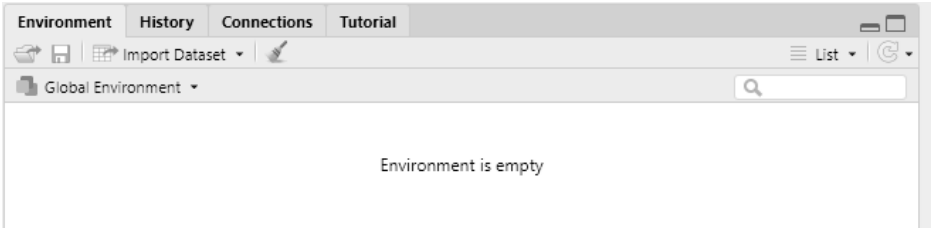


Figura 6. Panel de ambiente.

El tabulador del historial (**History)** registra todos los comandos introducidos/ejecutados a través del panel de consola. El historial puede ser guardado en un fichero de extensión .Rhistory.

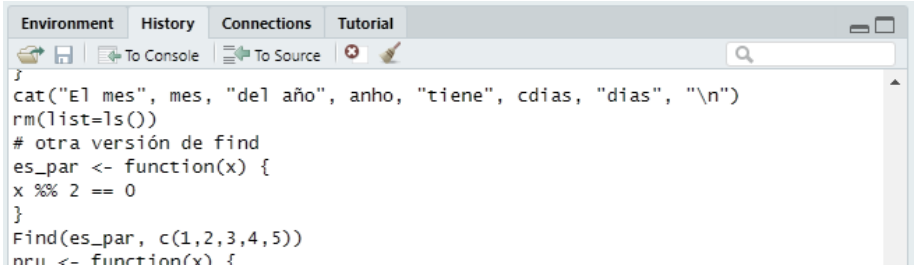


Figura 7. Ejemplo de salida del tabulador History del panel de ambiente.

El tabulador **Connections** posibilita conectar fácilmente diferentes fuentes de datos y explorar objetos y datos dentro de la conexión. **Tutorial** permite realizar acciones convenientes para acceder a tutoriales de RStudio.

d) Panel de ficheros: Se encuentra abajo a la derecha y consta de varios tabuladores, como se muestra en la figura 8.

- El tabulador **Files** muestra los ficheros que están en la carpeta de trabajo que seleccionó. Se puede navegar entre ellos, crear nuevas carpetas, borrar o renombrar

carpetas o ficheros, etc.

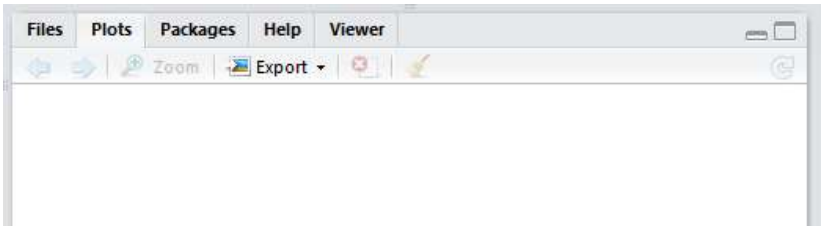


Figura 8. Panel de ficheros.

- El tabulador **Plots** contiene los gráficos creados con los datos procesados por el código en R. Si hay más de un gráfico, se puede navegar entre ellos, si antes no los ha borrado. Puede ocurrir que el sistema gráfico de R no esté soportado por la versión actual de RStudio, en tal caso aparecerá un mensaje de advertencia al estilo:

```
R graphics engine version XX is not supported by
this version of RStudio.
```

En ese caso la ventana Plot se deshabilita hasta que se instale una nueva versión de RStudio, pero los gráficos continuarán apareciendo en una ventana adicional.

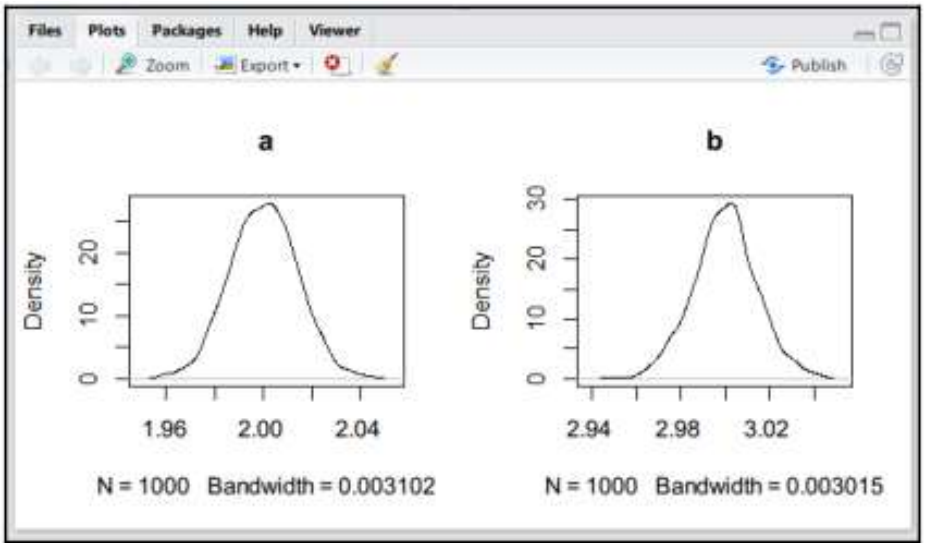


Figura 9. Ventana grafica del RStudio.

- El tabulador **Packages** muestra todos los paquetes instalados, para luego ser puestos en uso. Se pueden instalar o actualizar paquetes de la CRAN o eliminar paquetes existentes en la biblioteca particular del usuario.

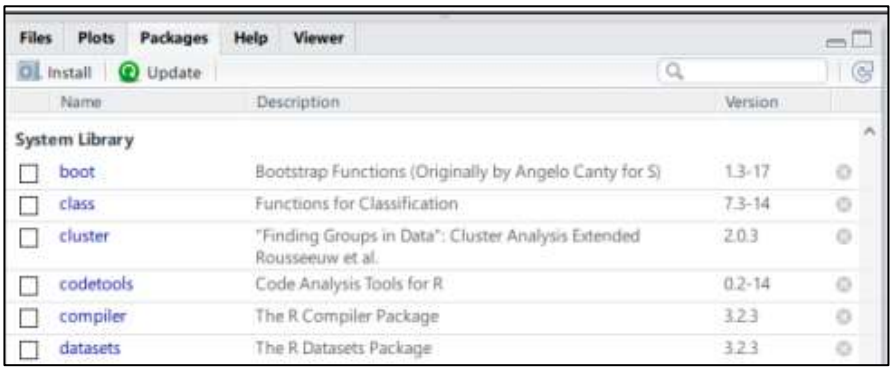


Figura 10. Tabulador Packages en la ventana grafica.

- El tabulador **Help** conecta al programador con documentación sobre R, que le permite aprender y emplear su funcionalidad.

Figura 11. Tabulador help en la ventana grafica.

Entre las vías de ver la documentación de una determinada función están:

- ✓ Escribir el nombre de la función en la caja **Search** y encontrarla directamente.
- ✓ Escribir el nombre de la función en el panel de consola y dar **F1**.

1.3.2 El primer script

Ahora se comenzará a escribir un primer programa en R. El programa (también llamado script fuente o simplemente script) está compuesto por diferentes **instrucciones** o **comandos** (son enunciados que generan una acción en el intérprete de R), preferentemente uno por línea, aunque hay comandos que pueden ocupar varias líneas.

Con el objetivo de construir un vector (el principal objeto de datos de R) con varios elementos usando la función **c**, escriba en el panel de script:

```
x <- c(5,10,15,20,25,30,35,40)
```

y luego oprima ***Run*** sobre la línea. Aparecerá en el panel de consola:

```
> x <- c(5,10,15,20,25,30,35,40)
```

Ahora teclee **x** en el panel de scripts (para indicar que quiere ver los valores del vector) y dé ***Run***, entonces aparecerá en la consola:

```
> x
 [1] 5 10 15 20 25 30 35 40
```

De forma similar pudo haber escrito en el panel fuente el script completo:

```
x <- c(5,10,15,20,25,30,35,40)
x
```

Entonces al seleccionarlo todo y dar ***Run***, le aparecerá en la consola:

```
> x <- c(5,10,15,20,25,30,35,40)
> x
 [1] 5 10 15 20 25 30 35 40
```

El código escogido a ejecutar debe estar completamente bien especificado, de lo contrario podría aparecer un error. Por ejemplo si se intenta ejecutar la línea donde está **x** y antes no se ejecutó la línea `x <- c(5,10,15,20,25,30,35,40)` ocurriría un error:

```
> x

Error: object 'x' not found
```

En lo adelante, esta ejecución se representará en una de las dos formas siguientes:

a) Simulando lo que se observa en los paneles de scripts y de consola.

Script de entrada en R

```
x <- c(5,10,15,20,25,30,35,40)
x
```

Consola de salida de R

```
> x <- c(5,10,15,20,25,30,35,40)
> x
[1] 5 10 15 20 25 30 35 40
```

b) De forma abreviada, el script de programa y el resultado que genera su ejecución.

```
x <- c(5,10,15,20,25,30,35,40)

x

[1] 5 10 15 20 25 30 35 40
```

Observe en todos los casos el símbolo **[ddd]** usado por el ambiente del lenguaje al comienzo de la respuesta, donde **d** es un dígito, en este caso **[1]**.

En una línea se puede escribir más de una instrucción (comando), separando con punto y coma (**;**) cada una de ellas.

1.4 Ayuda en R

La ayuda de R puede ser muy útil cuando se desea saber cómo utilizar una función determinada. Para obtener información de cualquier función en específico, por ejemplo **log**,

se emplea el comando **help**(nombre de la función) o su forma abreviada **?nombre de la función**.

Ejemplo:

```
help(log)
?log
```

Además de la ayuda de la función también puede solicitarse ayuda acerca de sus argumentos o ver ejemplos; para la función `log` sería respectivamente `args(log)` y `example(log)`.

Cuando se desea información sobre caracteres especiales de R, el argumento se debe encerrar entre comillas dobles o simples, para que lo identifique como una cadena de caracteres. Esto también es necesario hacerlo para las palabras con uso reservado en el lenguaje, por ejemplo:

```
help("for")
help("if")
help("[[")
help('function')
```

Por otra parte, el sistema puede mostrar un listado de contenidos acerca de algún tópico cualquiera invocando la función **help.search(**tópico_de_interés**),** que abreviadamente se invoca con **??** tópico_de_interés.

Ejemplo:

Si se desea saber acerca del tópico **stats**, lo que puede incluir, funciones, paquetes, variables, etc., se hace de una de las maneras siguientes:

```
help.search("stats")
?? "stats"
```

Unidad 2. Elementos básicos del lenguaje

2.1 Alfabeto del lenguaje R

El alfabeto del lenguaje R lo componen:

- las letras minúsculas y mayúsculas del alfabeto inglés:

  a b c d e f g h i j k l m n o p q r s t u v w x y z

  A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

- los dígitos numéricos: **0 1 2 3 4 5 6 7 8 9**
- caracteres especiales de puntuación: **_ { } [ ] # ( )** < > **% : ; . ? * + - / ^ & | ~ ! = , \ " '**
- espacios en blanco

El lenguaje es ***sensitivo al caso***, es decir, las letras mayúsculas y minúsculas son consideradas ***distintas*** en la composición de un nombre de objeto del lenguaje.

A su vez R es un lenguaje interpretado. Pueden ser entrados comandos (instrucciones) de uno en uno en el prompt del panel de consola para su ejecución inmediata o ejecutar un grupo de comandos que están en un fichero fuente.

Un programa en R opera con **objetos de datos** (simplemente objetos), los cuales tienen un tipo (***tipo de objeto***), que son ***contenedores*** de valores representados por ciertas estructuras de datos. Las constantes, las variables, las expresiones, evaluaciones de funciones, etc., son objetos y sus tipos pueden ser: vector, matriz, arreglo, data frame y lista, las cuales se estudiarán en detalles en unidades posteriores.

El principal tipo de objeto de R es el **vector**. Un ejemplo de vector es un vector numérico, formado por una secuencia de números. Desde la unidad anterior se conoce que para crear un vector (por ejemplo, `x`) con ocho números `5,10,15,20,25,30,35,40` puede emplearse el comando:

```
x <- c(5,10,15,20,25,30,35,40)
```

Aquí la función **`c()`** toma un número arbitrario de argumentos y forma un vector por concatenación. Un valor simple (escalar) es considerado en el lenguaje como un vector de longitud 1.

Los datos que componen los objetos son en última instancia valores simples y se clasifican en los siguientes ***tipos de datos fundamentales o atómicos***:

- **numeric**: abstracción del conjunto de los números reales $\mathbb{R}$, o sea, números enteros y reales con doble precisión, abarcando los tipos:
  - **integer:** abstracción del conjunto $\mathbb{Z}$ de los números enteros, o sea valores exclusivamente enteros.
  - **double**: valores representados en notación interna flotante.
- **character**: cadenas alfanuméricas de texto.
- **complex**: números complejos.
- **logical**: lógicos o booleanos, sólo toman los valores TRUE o FALSE.

Constantes literales

Una constante es un valor almacenado en una localización de memoria, que posee un tipo dado y su valor no varía durante la ejecución del programa. Pueden ser **valores literales**; en tal caso se representan por valores explícitos del tipo al que pertenecen.

Ejemplo de constantes literales son:

- numeric: Pueden ser,
  - enteras: un número sin punto decimal seguida de la letra **L**. Ejemplo: 34L, -20L.
  - en punto flotante: el número se escribe con otra notación numérica: 23.0, - 23.45, 67.
- character: Se escriben entre comillas dobles " ", por ejemplo "casa", "Z".

- logical: TRUE o T, y FALSE o F.
- complex: 2.5+3i.

R tiene algunas constantes predefinidas:

- LETTERS: las 26 letras mayúsculas del alfabeto inglés.
- letters: las 26 letras minúsculas del alfabeto inglés.
- month.abb: abreviaturas en tres letras del nombre de los meses en inglés.
- month.name: el nombre de los meses en inglés.
- pi: 3.14159…

Variable

Una variable es un objeto de datos caracterizado por poseer una localización de memoria donde se almacena un valor, que es de un tipo dado, un nombre que lo identifica de otros y la posibilidad de poder modificar su valor durante su tiempo de vida.

R no requiere que se declare un tipo a una variable y durante su tiempo de vida puede aceptar valores de tipos diferentes.

Los nombres de variables pueden contener cualquier combinación de caracteres alfanuméricos, así como punto (.) y subrayado (_), pero no puede comenzar por un dígito o subrayado. Es recomendable usar nombres concisos y significativos, escribir los nombres de variables y funciones en minúscula, usar subrayado o punto para separar palabras de un mismo nombre, así como usar sustantivos para nombres de variables y verbos para nombres de funciones.

Debe evitarse el empleo de nombres de variables o funciones ya existentes en sus scripts, pues causaría confusión. Por ejemplo, son malos nombres:

- T – forma abreviada de la constante lógica TRUE,

- c – nombre de la función predefinida para concatenar o agrupar objetos,
- mean – nombre de una función predefinida del lenguaje, etc.

Comentarios

El carácter **#** es usado para indicar que lo que aparece a su derecha es un comentario, por ejemplo:

```
# Esto es un comentario.
```

Los comentarios permiten dar aclaraciones de las partes principales o más oscuras del programa, proporcionando una mayor legibilidad y constituyen una vía para su documentación.

En R no hay comentarios de varias líneas, por lo que de ser necesario se coloca el símbolo # al comienzo de cada línea consecutiva que compone el bloque de comentarios.

2.2 Operador de asignación

Para asignar un valor o una expresión a una variable se emplea el operador "**<-**", también se puede usar la función **assign("objeto",valor)** o los operadores **=** o **->.** En lo adelante se usará con más frecuencia el operador **<-**.

Ejemplos:

❖ Usando el operador "**<-**":

Script de entrada en R

```
z <- 25
z
```

Consola de salida de R

```
> z <- 25
> z
[1] 25
```

Lo cual también se puede obtener con la asignación entre paréntesis `(z <- 25).`

❖ Usando la función assign: **assign("z",25)**

- Usando el operador "**->**": **25 -> z**
- Usando el operador "**=**": **z = 25**

La asignación puede ser múltiple, cuando se desea asignar un mismo valor a varias variables, por ejemplo:

Script de entrada en R

```
A <- B <- 1   # Asignación múltiple
A
B
```

Consola de salida de R

```
> A <- B <- 1   # Asignación múltiple
> A
[1] 1
> B
[1] 1
```

Es recomendable usar **<-** para la asignación.

2.3 Operadores aritméticos y lógicos

Las operaciones permiten formar expresiones de varios tipos. Por defecto, las operaciones se aplican elemento por elemento de la estructura de datos de sus operandos, en tal caso se dice que la operación es **vectorizada** y el proceso se llama **vectorización**.

2.3.1 Operadores aritméticos

Las operaciones aritméticas básicas son las siguientes:

- Suma: **+**
- Resta: **-**
- Multiplicación: *****
- División: **/**
- Potenciación: **^** (también ******)
- Cociente de división entera: **%/%**

- Resto de división entera: **%%**

Ejemplos:

- Cociente y resto de división entera: **16 %/% 3** es **5** y **16 %%% 3** es **1**.
- Para:

```
a <- 2

f <- c(2,3,4)

d <- c(10,10,10)

e <- c(1,2,3,4)
```

se ejecutaría:

```
> d + e # vector + vector

[1] 12 13 14

> d + a # vector + escalar

[1] 12 12 12
```

Si los operandos (vectores) son de longitudes diferentes, como ocurre con la suma anterior de `d+a` (el vector `d` es de longitud 3 y el vector `a` de longitud 1), internamente R completa el vector de menor longitud, con los propios elementos del vector, lo que denomina el **reciclado** del vector (aunque sería deseable que ambos tengan la misma longitud). Para ello el vector `a` se completa internamente con dos veces el valor 2 y la suma `d+a` es 12 12 12. El vector reciclado **no** cambia su valor, R hace una transformación temporal del mismo completando con sus propios elementos comenzando por el primero, hasta alcanzar la longitud del otro operando vector.

Una **expresión aritmética** se define como una combinación de constantes, variables o funciones aritméticas usando los operadores aritméticos, donde pueden ser empleados

paréntesis para agrupar las operaciones. Al evaluar una expresión aritmética se obtiene como resultado un valor numérico.

En la evaluación de cualquier expresión hay que tener en cuenta dos clases de reglas:

- **Reglas de precedencia**: Permiten determinar, de entre varias operaciones diferentes cuál aplicar primero.
- **Reglas de asociatividad**: Permiten determinar, de entre varias operaciones de igual precedencia, en qué orden son aplicadas a sus operandos.

La precedencia entre operadores aritméticos es:

1) Términos entre paréntesis (del más interno al más externo).
2) Potenciación.
3) Multiplicación, división (ordinaria y entera), módulo (resto de división).
4) Adición y sustracción.

Por ejemplo, en **3 + 8 * (x – 5)** las operaciones se efectúan en el siguiente orden:

1) operación entre paréntesis,
2) producto,
3) suma.

Observe la evaluación de esta expresión en la consola de R, asumiendo x igual a 10:

Script de entrada en R

```
x <- 10
d <- 3 + 8 * (x - 5)
d
```

Consola de salida de R

```
> x <- 10
> d <- 3 + 8 * (x - 5)
> d
[1] 43
```

Si se eliminan los paréntesis, el producto se efectúa primero, luego la suma y por último la resta:

Script de entrada en R

```
x <- 10
d <- 3 + 8 * x - 5
d
```

Consola de salida de R

```
> x <- 10
> d <- 3 + 8 * x - 5
> d
[1] 78
```

Si una expresión se utiliza como un comando, su valor se imprime y se pierde. Así el comando **1/x** simplemente imprime el inverso del valor del objeto **x** sin modificar su valor.

Script de entrada en R

```
x <- c(25,26,27,28,29,30,31,32,33,34)
print(x)
1/x
x
```

Consola de salida de R

```
> x <- c(25,26,27,28,29,30,31,32,33,34)
> print(x)
[1] 25 26 27 28 29 30 31 32 33 34
> 1/x
[1] 0.04000000 0.03846154 0.03703704 0.03571429 0.03448276 0.03333333
[7] 0.03225806 0.03125000 0.03030303 0.02941176
> x
[1] 25 26 27 28 29 30 31 32 33 34
```

En este caso se ejecutaron las líneas correspondientes a **`1/x`** y **`x`** del script de entrada, para obtener la salida descrita.

2.3.2 Funciones de impresión `print()` y `cat()`

Note que en el ejemplo precedente se ha usado la función **`print`** como forma alternativa de salida de valores de una variable. En la sintaxis **`print(x)`**, **`x`** representa el valor o valores a

imprimir. La función **print** invoca una operación de nueva línea que causa que la próxima función **print** a ejecutar muestre sus salidas al comienzo de una nueva línea.

Si el vector es muy largo y no cabe en una línea, R usa las líneas siguientes para continuar imprimiendo el vector. Los números entre corchetes no forman parte del objeto e indica la posición del vector en ese punto. Puede verse en el ejemplo anterior, que [1] indica que el primer elemento del vector está en esa línea, [7] indica que la línea siguiente comienza por el séptimo elemento del vector y así sucesivamente.

En las salidas numéricas se puede especificar con cuantos dígitos se desea mostrar el número, usando el parámetro **digits** de la función **print**, al que se asigna esa cantidad, por ejemplo:

Orden de impresión	**Salida**
`print(2.3678)`	`[1] 2.3678`
`print(2.3678, digits=3)`	`[1] 2.37`
`print(23456.3678, digits=3)`	`[1] 23456`
`print(23456.3678, digits=8)`	`[1] 23456.368`

Otra forma de imprimir es usando la siguiente forma simplificada de la función **cat()**:

```
cat(objetos, sep = " ")
```

donde **objetos** son los elementos a imprimir, separados por coma, **sep** indica cómo se separarán esos objetos en la impresión (por defecto un espacio). Los saltos de línea deben ser informados explícitamente, mediante la expresión `"\n"`. Por ejemplo, con:

```
nom1 <- "Larisa"
nom2 <- "Jorge"
cat("Bienvenida", nom1, "\n")
```

```
cat("Bienvenido", nom2, "\n")
```

se obtiene:

```
> cat("Bienvenida", nom1, "\n")
Bienvenida Larisa
> cat("Bienvenido", nom2, "\n")
Bienvenido Jorge
```

Sin embargo, con:

```
cat("Bienvenida", nom1)
cat("Bienvenido", nom2)
```

hubiera salido:

```
> cat("Bienvenida", nom1)
Bienvenida Larisa> cat("Bienvenido", nom2)
Bienvenido Jorge
```

excepto si lo ejecuta línea a línea.

En la lista de parámetros de la función **`cat()`** pueden aparecer expresiones de control de la salida, como **`"\n"`**. Ellas se caracterizan por comenzar por **`\`** seguida de combinaciones que determinan la función de control que se cumplirá. Algunas expresiones de control son:

- Nueva línea: **`"\n"`**
- Tabulación horizontal: **`"\t"`**
- Carácter comilla: **`"\""`**
- Carácter apóstrofo: **`"\'"`**
- Carácter barra invertida: **`"\\"`**

Las expresiones de control tienen efecto también cuando aparecen como parte de una cadena de caracteres en una operación de salida.

2.3.3 Operadores lógicos

Una **expresión lógica** es una expresión que al evaluarse nos da una de dos posibles interpretaciones: se cumple (verdadero) o no se cumple (falso), denotados en el lenguaje con las literales `FALSE` o `TRUE`, o abreviadamente `F` y `T`, respectivamente. Una expresión lógica está constituida por las constantes lógicas antes mencionadas, objetos que toman esos valores, expresiones aritméticas y comparaciones, combinadas con operadores lógicos y eventualmente paréntesis para agrupar operaciones.

Una **comparación** es la combinación de dos expresiones fundamentales a través de un operador de comparación:

Expresión_1 Op_R Expresión_2

Los operadores de comparaciones (***Op_R***) son:

- Mayor que: `>`
- Menor que: `<`
- Mayor o igual a: `>=`
- Menor o igual a: `<=`
- Igualdad: `==`
- Desigualdad: `!=`

Ejemplos de comparaciones son:

```
a > b
b != a+9
```

Si para los ejemplos anteriores el valor de `a` es 5 y el de `b` es 6 entonces la primera expresión es FALSE, en tanto la segunda es TRUE, ya que 6 es distinto de 14.

Los objetos lógicos se pueden utilizar con la aritmética ordinaria, en cuyo caso son transformados en valores numéricos, FALSE se convierte en 0 y TRUE en 1.

Ejemplo:

Script de entrada en R

```
1 < 2
3 < 4
(1 < 2) * (3 < 4)
```

Consola de salida de R

```
> 1 < 2
[1] TRUE
> 3 < 4
[1] TRUE
> (1 < 2) * (3 < 4)
[1] 1
```

En el tercer cálculo la comparación 1 < 2 devuelve TRUE, al igual que 3 < 4. Ambos valores son tratados como 1, por ello el producto es 1.

Hay que tener cuidado con la comparación en igualdad (**==)** para valores reales, pues dado su carácter aproximado pueden fallar. Así, **0.9 == (1.1 - 0.2)** da FALSE, cuando se esperaba TRUE. En tales casos es mejor emplear la función **all.equal()**, que proporciona un umbral de aproximación:

all.equal(v1, v2, tolerance = 1e-16)

- **v1** y **v2**: valores a ser comparados.
- **tolerance**: aproximación usada en la comparación. Por defecto **1e-16**, la menor para valores double.

Por ello **all.equal(0.9, 1.1-0.2)** devuelve TRUE.

Para probar la desigualdad entre dos valores, en vez de **x != y** es mejor emplear:

`! all.equal(x,y)`.

Si los operandos se redondean a la misma cantidad de dígitos decimales, también se obtendría el resultado adecuado de la comparación:

```
> round(0.9,1) == round(1.1-0.2,1)
[1] TRUE
```

La tabla 1 muestra los **operadores lógicos** que permiten escribir expresiones más complejas.

Tabla 1. Operadores lógicos

Operación	**Operador**	**Ejemplo**	**Significado**
Negación lógica vectorizada (¬)	`!`	`!x`	Si x es TRUE, entonces !x es FALSE y viceversa.
Conjunción lógica (∧) vectorizada (se aplica elemento a elemento de cada vector)	`&`	`x&y`	x&y == TRUE, si x e y son TRUE, para otros valores da FALSE.
Idem al anterior para escalares (vectores de una sola componente).	`&&`	`x&&y`	Idem
Disyunción lógica (∨) vectorizada (se aplica elemento a elemento de cada vector)	`x|y`	`x|y`	FALSE|FALSE da FALSE, otro par de valores lógicos da TRUE.
Idem al anterior para escalares.	`||`	`x||y`	Idem
Disyunción exclusiva vectorizada (componente a componente)	`xor`	`xor(x,y)`	Si x e y tienen valores diferentes resulta TRUE, en otro caso resulta FALSE.
Para todo (∀)	`all`		Cuantificador universal
Existe (∃)	`any`		Cuantificador existencial

Ejemplo:

Se forman 3 vectores a, b y c, de tres componentes cada uno y se realizan operaciones lógicas y de comparación con ellos.

Script de entrada en R

```
# Se forman 3 vectores de tres componentes cada uno
a <- c(1, 2, 3)
b <- c(1, 2, 3)
c <- c(1, 2, 4)
# Se comparan dos vectores, componente a componente
a == b
b == c
# Se hace la conjunción de los dos resultados anteriores,
# componente a componente
a == b & b == c
```

Consola de salida de R

```
> # Se forman 3 vectores de tres componentes cada uno
> a <- c(1, 2, 3)
> b <- c(1, 2, 3)
> c <- c(1, 2, 4)
> # Se comparan dos vectores, componente a componente
> a == b
[1] TRUE TRUE TRUE
> b == c
[1] TRUE TRUE FALSE
> # Se hace la conjunción de los dos resultados anteriores,
> # componente a componente
> a == b & b == c
[1] TRUE TRUE FALSE
```

Ejemplo de comparaciones entre vectores de longitudes diferentes (exigen reciclado):

Script de entrada en R

```
# Se forman 2 vectores de tres y cuatro componentes, respectivamente
d <- c(1, 2, 3)
e <- c(1, 2, 3, 0)
# Se comparan los vectores, componente a componente
d == e # Reciclado: d se completa internamente con 1
# Se hace la conjunción de d y e, componente a componente
d & e  # Igual que el reciclado anterior
```

Consola de salida de R

```
> # Se forman 2 vectores de tres y cuatro componentes,respectivamente
> d <- c(1, 2, 3)
> e <- c(1, 2, 3, 0)
# Se comparan los vectores, componente a componente
> d == e  # Reciclado: d se completa internamente con 1
[1] TRUE TRUE TRUE FALSE
Warning message:
In d == e : longer object length is not a multiple of shorter object
length
# Se hace la conjunción de d y e, componente a componente
> d & e # Igual que el reciclado anterior
[1] TRUE TRUE TRUE FALSE
Warning message:
In d & e: longer object length is not a multiple of shorter object
length
```

Nota: Observar el mensaje de advertencia que muestra el programa.

Un ejemplo de empleo de las funciones **any** y **all** se muestra a continuación:

Script de entrada en R

```
x <- c(-2, -3, 2, 3, 1, 0, 0, 1, 2)
any(x > 1)   # ¿Hay valores en x mayores que 1?
all(x <= 1)  # ¿Todos los valores en x son menores o iguales que 1?
```

Consola de salida de R

```
> x <- c(-2, -3, 2, 3, 1, 0, 0, 1, 2)
> any(x > 1)   # ¿Hay valores en x mayores que 1?
[1] TRUE
> all(x <= 1) # ¿Todos los valores en x son menores o iguales que 1?
[1] FALSE
```

2.3.4 Reglas de precedencia

Las reglas de precedencia con los operadores principales (algunos aún no estudiados) se expresan en la tabla 2, con precedencia descendente de la primera fila a la última.

Las **operaciones entre paréntesis** se realizan con **mayor prioridad**, por eso estos últimos deben ser usados para cambiar o aclarar el orden en que son efectuadas las operaciones.

Tabla 2. Precedencia de los principales operadores

Prioridad	Operador	Significado
1	`$`	Subindizado en listas y data frames
2	`[ ]` `[[ ]]`	Subindizado de vectores y matrices Subindizado de listas
3	`^ **`	Potenciación
4	`%/% , %%` `%*%`	División entera, resto de división entera Multiplicación de matrices
5	`* , /`	Multiplicación, división
6	`+ , -`	Suma, resta
7	`< , > , <= , >= , == , !=`	Comparaciones
8	`!`	Negación lógica
9	`& , |` `&& , ||`	Conjunción, disyunción lógica vectorizada Conjunción, disyunción lógica
10	`<-`	Asignación

2.4 Funciones aritméticas predefinidas

El lenguaje posee un grupo de **funciones predefinidas**, algunas se listan a continuación, donde x denota un vector de una o varias componentes:

Tabla 3. Algunas funciones aritméticas predefinidas

Función en R	Notación Matemática	Significado
`abs(x)`	$\|x\|$	Valor absoluto de x
`exp(x)`	e^x	Exponencial de x
`log(x)`	$ln\, x$	Logaritmo neperiano (base e) de x
`log10(x)`	$log\, x$	Logaritmo en base 10 de x
`log(x,n)`	$log_n\, x$	Logaritmo en base n de x
`sin(x)`	$sin\, x$	Seno de x (x en radianes)
`cos(x)`	$cos\, x$	Coseno de x (x en radianes)

`tan(x)`	$tan\, x$	Tangente de x (x en radianes)
`sqrt(x)`	$\sqrt{x}$	Raíz cuadrada de x
`factorial(x)`	$x!$	Factorial de x
`choose(n,k)`	C_k^n	Número de formas de escoger *k* elementos a partir de un conjunto de *n* elementos: $\frac{n!}{k!(n-k)!}$
`max(x)`		Máximo valor encontrado en x
`min(x)`		Mínimo valor encontrado en x
`range(x)`		Menor y mayor valor encontrado en x
`length(x)`		Longitud de x (cantidad de valores en x)
`sum(x)`		Suma de los elementos en x
`prod(x)`		Producto de los elementos en x
`mean(x)`		Media aritmética (promedio) de los elementos en x
`sort(x)`		Versión ordenada de x
`ceiling(x)`		Menor entero mayor o igual que x (Redondeo a entero por exceso)
`floor(x)`		Mayor entero no mayor que x (Redondeo a entero por defecto)
`trunc(x)`		Parte entera del valor de x
`round(x,d)`		Redondea valores en x a d lugares decimales (por defecto d=0)
`signif(x,d)`		Redondea los valores de x mostrando únicamente d cifras significativas (que incluyen los de la parte entera y de la parte decimal).

Muchas de las anteriores operaciones son vectorizadas. Para emplear las funciones trigonométricas los ángulos deben ser expresados solo en radianes. Los argumentos en grados deben ser transformados a radianes mediante la operación **grados*pi/180**.

Ejemplo:

Script de entrada en R

```
x <-sqrt(4)
x
```

Consola de salida de R

```
> x <-sqrt(4)
> x
[1] 2
```

Suponga x como una secuencia (vector) de valores del 1 al 5 (1:5). Algunas funciones aplicadas al vector x se muestran en el siguiente script:

Script de entrada en R

```
x <- 1:5
factorial(x)
max(x)
min(x)
range(x)
length(x)
sum(x)
prod(x)
mean(x)
```

Consola de salida de R

```
> x <- 1:5
> factorial(x)
[1]   1   2   6   24   120
> max(x)
[1] 5
> min(x)
[1] 1
> range(x)
[1] 1 5
> length(x)
[1] 5
> sum(x)
[1] 15
> prod(x)
[1] 120
> mean(x)
[1] 3
```

Por último, se ejemplifican algunos redondeos.

Script de entrada en R

```
x <- c(5.314, 5.653, -5.651, -5.378)
ceiling(x)  # Menor entero mayor o igual que x
floor(x)    # Mayor entero no mayor que x
trunc(x)    # Parte entera de x
round(x)    # Redondeo de x con 0 lugares decimales
round(x,2)  # Redondeo de x con 2 lugares decimales
signif(x,2) # Redondeo de x en d cifras significativas
```

Consola de salida de R

```
> x <- c(5.314, 5.653, -5.651, -5.378)
> ceiling(x)  # Menor entero mayor o igual que x
[1] 6  6 -5 -5
> floor(x)    # Mayor entero no mayor que x
[1]  5  5 -6 -6
> trunc(x)    # Parte entera de x
[1]  5  5 -5 -5
> round(x)    # Redondeo de x con 0 lugares decimales
[1]  5  6 -6 -5
> round(x,2)  # Redondeo de x con 2 lugares decimales
[1]  5.31  5.65 -5.65 -5.38
> signif(x,2) # Redondeo de x en d cifras significativas
[1]  5.3  5.7 -5.7 -5.4
```

2.5 Listados, borrado e historial

El comando **`ls()`** nos proporciona un listado de los objetos que hay actualmente en el espacio de trabajo, mostrando sus nombres.

- Si se quiere listar solo aquellos objetos que contengan un carácter en particular, se puede usar la opción **`pattern`** (que se puede abreviar como **`pat`**).

  Ejemplo: `pat="m"`.

- Para restringir la lista a aquellos objetos que comienzan con un carácter, usar ^carácter**.**

  Ejemplo: `ls(pat="^z").`

- La función **`ls.str()`** muestra algunos detalles de los objetos en memoria, puede usarse con la opción **`pat`** vista anteriormente.

Ejemplo:

Script de entrada en R

```
name <- "Python"
n1 <- 10
n2 <- 0.5
nums <- c(10, 20, 30, 40, 50, 60)
print(ls())
print("Detalles de objetos en memoria:")
print(ls.str())
```

Consola de salida de R

```
> name <- "Python"
> n1 <- 10
> n2 <- 0.5
> nums <- c(10, 20, 30, 40, 50, 60)
> print(ls())
[1] "n1"   "n2"   "name"  "nums"
> print("Detalles de objetos en memoria:")
[1] "Detalles de objetos en memoria:"
> print(ls.str())
n1 :  num 10
n2 :  num 0.5
name :  chr "Python"
nums :  num [1:6] 10 20 30 40 50 60
```

Para hacer búsquedas de objetos en el espacio de trabajo se pueden utilizar los comandos **apropos**("nombre del objeto") y **find**("nombre del objeto").

Para eliminar uno o varios objetos del espacio de trabajo se usa el comando **rm(objeto)**; para removerlos todos se puede usar:

rm(list = ls(all = TRUE)) o **rm(list = ls())**.

Mediante el teclado (flechas ↑ ↓) se puede acceder al historial de comandos, o sea, los últimos comandos ejecutados.

2.6 Números aleatorios

Un **número aleatorio** es un resultado de una variable aleatoria especificada por su función de distribución. Cuando no se especifica ninguna distribución, se presupone que se utiliza la distribución uniforme continua en el intervalo [0,1].

Usando la computadora se pueden producir secuencias numéricas pseudo-aleatorias. Un **número pseudo-aleatorio** es un número generado en un proceso que parece producir números al azar, pero no lo hace realmente. Las secuencias de números pseudo-aleatorios no muestran ningún patrón o regularidad aparente desde un punto de vista estadístico, a pesar de haber sido generadas por un algoritmo completamente determinista, en el que las mismas condiciones iniciales producen siempre el mismo resultado.

Los generadores de números pseudo-aleatorios son ampliamente utilizados en campos tales como el modelado por computadora, estadística, diseño experimental, criptografía, método de MonteCarlo, teoría de juegos, etc.

El lenguaje R posee múltiples formas de generación de secuencias pseudo-aleatorias, de las cuales ahora se presentan las opciones más simples.

1. Extraer muestras de tamaño especificado con y sin reposición.

```
sample(x, size, replace = FALSE)
```

donde:

`x`: Es un vector de uno o más elementos de donde se extraen los valores o un número entero positivo n para indicar el vector formado por los números del 1 al n.

`size:` Es un entero no negativo con la cantidad de valores a escoger de x. Si no se especifica devuelve una permutación de x o del vector formado por los elementos del 1 al n.

`replace`: Para indicar si el muestreo se realiza con remplazamiento (TRUE) o sin remplazamiento (FALSE), este último es el valor por defecto.

Ejemplos:

a) Generar una permutación aleatoria de elementos entre 1 y 5.

```
sample(5)
```

b) Generar una permutación aleatoria de elementos del vector (10, 11, 12, 13, 14, 15).

```
sample(10:15)
```

c) Extraer una muestra aleatoria de tamaño 5 de elementos del vector (1, 2, 3, 4, 5, 6, 7, 8, 9, 10) sin y con remplazamiento.

```
sample(1:10, size = 5)
```

Con **replace = TRUE**, la muestra se hace con remplazamiento:

```
sample(1:10, size = 5, replace = TRUE)
```

d) Extraer una muestra aleatoria de tamaño 3 sin reposición, de elementos del vector de letras en minúsculas del alfabeto inglés.

```
sample(letters, size = 3)
```

2. Generar números uniformemente distribuidos en el intervalo [min, max].

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

donde:

`n`: Es un entero no negativo con la cantidad de valores a generar.

`min`: Cota inferior del intervalo (por defecto 0).

`max`: Cota superior del intervalo (por defecto 1).

Ambas cotas son valores reales.

Ejemplos:

a) Generar 5 números aleatorios en el intervalo [0,1].

```
runif(5)
```

b) Generar 5 números aleatorios en el intervalo [2,10].

```
runif(5, min = 2, max = 10)
```

3. Generar números aleatorios con distribución normal.

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

donde:

`n`: Es un entero no negativo con la cantidad de valores a generar.

`mean`: Valor de la media (por defecto 0).

`sd`: Valor de la desviación estándar (por defecto 1).

Ambos valores, media y sd, son reales.

Ejemplo:

Generar 5 números con distribución normal con media 0 y desviación estándar 1.

`rnorm(5)` o equivalentemente **`rnorm(n = 5)`**

Otras formas de generar vectores aleatoriamente, usando diversas leyes de distribución, se muestran a continuación. En la mayoría de los casos el primer parámetro **`n`** es la cantidad de números a generar y el o los siguientes parámetros son valores que por defecto son dados a ciertos parámetros de la distribución que se especifica.

— Binomial con **`size`** ensayos y probabilidad de éxito **`prob`**:

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

— Poisson con media y varianza **`lambda`**:

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

— Geométrica con probabilidad de éxito **prob**:

```
rgeom(n, prob)
```

— Hipergeométrica con **m1** objetos del tipo I, **m2** objetos del tipo II y **k** el número de objetos a extraer (k debe tomar valores entre 1 y **m1+m2**):

```
rhyper(n, m1, m2, k)
```

— Binomial negativa con **size** ensayos exitosos (o también parámetro de dispersión) y probabilidad de éxito de cada ensayo **prob**:

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

— Student con **df** grados de libertad (df > 0) y **ncp** parámetro de no centralidad; si se omite se asume FALSE (t-student central):

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

— Fisher con grados de libertad **df1** y **df2**, y **ncp** parámetro de no centralidad:

```
rf(n, df1, df2, ncp = 0)
```

— Exponencial con razón **rate** (rate es el inverso de la media del vector aleatorio):

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

— Beta con parámetros de forma **shape1** y **shape2** (ambos mayores que cero) y **ncp** parámetro de no centralidad:

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

— Logística con parámetros de posición (**location**) y escala **(scale)**:

```
rlogis(n, location = 0, scale = 1)
```

— Log-normal con media **meanlog** y desviación estándar **sdlog**:

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

— Weibull con parámetros de forma **shape** y escala **scale**:

```
rweibull(n, shape, scale = 1)
```

— Chi-cuadrado con **df** grados de libertad y **ncp** parámetro de no centralidad:

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

— Cauchy con parámetros de posición (**location**) y escala **(scale)**:

```
rcauchy(n, location = 0, scale = 1)
```

— Distribución multinomial con **size** objetos puestos en k cajas que usan valores de probabilidad contenidos en el vector k-dimensional **prob**:

```
rmultinom(n, size, prob)
```

Ejercicios

1. Calcular e imprimir con sus descripciones y con 5 dígitos:
   a) El valor de la función de densidad normal de media cero y varianza uno, en 0, esto es, $\varphi(0)$.
   b) $P(Z < 1.64)$, donde $Z \sim N(0,1)$.
   c) Los cuartiles de la función de densidad normal de media cero y varianza uno.
   d) El percentil 0.95 de la función de densidad t-student con 10 grados de libertad.
   e) Genere 20 números aleatorios con distribución normal de media 2 y varianza 9.
2. Hallar la raíz cuadrada de 5, 5+0i, 5+5i e imprima los resultados con 6 lugares decimales.
3. En cada inciso evalúe cada expresión sin usar la computadora, luego compruebe su resultado en la consola de R.

   a) 1.7 * 8 b) 7 ** 2 c) 1 / (2 ^ 3) d) 3 + (4 * 5) e) (8 + 6) / 5 f) 3 * (-2 ** 5)

   g) 7 %/% 3 h) 14 %% 4 i) 9 %%3 j) 14 %/% 4 k) 5 %% 5
4. En cada inciso escriba un pequeño script para evaluar la expresión numérica dada, suponiendo los valores de las variables: $a = 5$, $b = 3$, y $c = 7$.

a) (a * b) + c b) a * (b + c) c) (1 + b) * c

d) a ^ c e) b ^ (c - a) f) (c - a) ^ b

5. En cada inciso escriba un pequeño script para evaluar la función predefinida dada, suponiendo los valores de las variables: $a = 6$ y $b = 4$.

a) trunc((-a + 0.5) / 2) b) round(a / b) c) abs(a - 5)

d) abs(4 - a) e) round(a + 0.5) f) ceiling(b * 0.7)

6. Considere los vectores numéricos `x, y`:

```
set.seed(7)
x <- sample (1:9, size = 5, replace = T)
y <- sample (1:9, size = 5, replace = T)
```

Obtenga:

a) min(x)

b) min(y)

c) max(x)

d) max(y)

e) La suma de los elementos de un nuevo vector obtenido por la multiplicación de x e y.

Unidad 3. Clases de objetos en R y sus atributos

Como ya se presentó en la unidad 1, todas las entidades que manipula R se conocen con el nombre de **objetos**, los cuales están compuestos de elementos. A diferencia de otros lenguajes de programación, no existen diferenciadamente valores escalares, estos son vectores de longitud 1. Los objetos poseen nombre, contenido y un atributo que especifica cual es el tipo de dato asociado al objeto.

El tipo de datos de un objeto puede clasificarse en atómico (fundamental) o en estructuras de datos. Los principales tipos de datos atómicos (no los únicos) son **`numeric`** (`integer`, `double`), **`character`**, **`logical`** y **`complex`**.

Las estructuras de datos en R dan la posibilidad de organizar los datos de maneras diversas con fines de almacenamiento y de análisis. Las estructuras de datos más empleadas en R son:

- **`vector`** (vectores)**:** secuencias de datos de un único tipo.
- **`matrix`** (matrices)**:** secuencias de datos de un único tipo, en representación bidimensional.
- **`array`** (arreglos generales)**:** Contienen datos de un único tipo, en representación mayor a dos dimensiones.
- **`list`** (listas)**:** Similar a un vector, pero puede contener datos de diferentes tipos, inclusive vectores, matrices, data frames e incluso listas).
- **`dataframe`** (conjuntos de datos): Semejan vectores de listas, en una representación bidimensional.
- **`function`** (funciones): funciones de R más las escritas por el usuario. Son objetos que pueden ser llamados y que al tomar un grupo de entradas (argumentos o parámetros) devuelven un valor como salida.

Todos los objetos en R poseen un *modo* (que describe como se almacena el objeto) y una *clase* (que describe la estructura interna del objeto y permite, entre otras cosas, determinar cuáles operaciones se les puede aplicar).

Usando la función **`str(`**`objeto`**`)`** puede obtenerse información acerca de la estructura del objeto.

3.1 Atributos de los objetos en R

Los atributos de un objeto suministran información específica sobre el propio objeto. Todos los objetos tienen dos atributos intrínsecos: el **modo** (mode) y su **longitud** (lenght). Con el modo de un objeto se designa el tipo básico de sus componentes fundamentales y puede ser *numeric* (tanto valores de tipo integer como double se consideran de modo numérico), *complex, logical, character*.

Algunos ejemplos de atributos de objetos de R se muestran en la tabla 4, donde x es un objeto.

Tabla 4. Ejemplos de atributos de objetos de R

Atributo (función que lo denota)	**Significado**

`mode(x)`
`storage.mode(x)`
`typeof(x)`
	Forma de almacenamiento interno o tipo de dato del argumento. Para vectores, matrices y arreglos, mode() retorna *numeric*, *complex*, *logical* o *character*. Para estos objetos, cuando el modo es *numeric*, storage.mode() y typeof() devuelven *double* o *integer*.
	Otros tipos son: *list, function*, etc.

`length(x)`	Cantidad de integrantes del objeto x
`class(x)`	Estructura de clase que soporta el objeto x: para un vector es *numeric* (caso de componentes flotantes), *integer*, *complex*, *logical* o *character*; para una matriz su clase se denota con ambos valores *matrix* y *array*. Otros valores que devuelve esta función son: *factor, array, list, dataframe, function*, etc.

`names(x)`	Permite dar nombres o devolver nombres a componentes de un vector o lista.
`dim(x)`	Dimensiones asociadas al objeto x.
`dimnames(x)`	Permite dar nombres o devolver nombres a las dimensiones de una matriz o arreglo.
`levels(x)`	Valores diferentes presentes en un factor.

Ejemplo:

Script de entrada en R

```
x <- c(2,-1)
x
mode(x)    # Tipo del objeto x
length(x)  # Longitud del objeto x
class(x)   # Clase del objeto x (igual a mode, pues es un vector)
storage.mode(x) # Forma de almacenamiento
```

Consola de salida de R

```
> x <- c(2,-1)
> x
[1]  2 -1
> mode(x)    # Tipo del objeto x
[1] "numeric"
> length(x)  # Longitud del objeto x
[1] 2
> class(x)   # Clase del objeto x (igual a mode, pues es un vector)
[1] "numeric"
> storage.mode(x)
[1] "double"
```

Los atributos de un objeto x pueden ser accedidos (si los posee) usando la función **`attributes(x)`**, que retorna una lista de los atributos no intrínsecos. Si x no contiene atributos no intrínsecos, la función **`attributes(x)`** retorna NULL.

3.2 Datos numéricos

El lenguaje agrupa los datos numéricos en las categorías: numeric (números reales de doble precisión), integer (números enteros) y complex (números complejos). Un valor que se puede

interpretar como un número, por defecto el lenguaje lo trata como un dato de tipo numeric.

Para que un valor entero sea tratado como tal debe llevar el sufijo **L**.

Ejemplo:

Script de entrada en R

```
x<-5
x
class(x)
mode(x)
storage.mode(x)  # Modo de almacenamiento
```

Consola de salida de R

```
> x<-5
> x
[1] 5
> class(x)
[1] "numeric"
> mode(x)
[1] "numeric"
> storage.mode(x)  # Modo de almacenamiento
[1] "double"
```

Si en vez de asignar a x el valor 5 se escribe **5L**, `storage.mode(x)` da como salida "integer".

Script de entrada en R

```
x<-5L
x
class(x)
mode(x)
storage.mode(x)  # Modo de almacenamiento
```

Consola de salida de R

```
> x<-5L
> x
[1] 5
> class(x)
[1] "integer"
> mode(x)
[1] "numeric"
> storage.mode(x)  # Modo de almacenamiento
[1] "integer"
```

El tipo de un valor puede igualmente ser confirmado con la función **`typeof()`**.

```
x <- 12.3

y <- 12L

# confirmar tipos

typeof(x)

[1] "double"

typeof(y)

[1] "integer"
```

Para muchas aplicaciones no es importante diferenciar el uso de un entero o un double, aunque los enteros consumen menos espacio (4 bytes) que un valor double (8 bytes) y menos tiempo operacional.

3.3 Datos especiales

En algunos casos las componentes de un objeto pueden no ser completamente conocidas, se les llama "**not available**" y se le asigna el valor especial **NA**. R considera que los datos NA son lógicos.

En general una operación con elementos **NA** resulta **NA**, a no ser que, mediante una opción de la función, pueda omitirse o tratarse los datos faltantes, o no disponibles, de forma especial.

La opción por defecto en cualquier función es **na.rm = FALSE** (que indica que NO elimina los NA), que da como resultado NA cuando existe al menos un dato faltante. Con la opción **na.rm = TRUE**, la operación se efectúa con los datos válidos.

Script de entrada en R

```
x <- NA
x + 1    # El resultado es NA
y <- c(x,3,5,x)
mean(y)             # Tiene en cuenta los NA's.
mean(y,na.rm=TRUE)  # No tiene en cuenta los NA's.
```

Consola de salida de R

```
> x <- NA
> x + 1    # El resultado es NA
[1] NA
> y <- c(x,3,5,x)
> mean(y)             # Tiene en cuenta los NA's.
[1] NA
> mean(y,na.rm=TRUE)  # No tiene en cuenta los NA's.
[1] 4
```

En determinadas ocasiones los cálculos realizados pueden llevar a respuestas con valor infinito positivo (representado por R como **Inf**) o infinito negativo **(-Inf**). Es posible realizar y evaluar cálculos que involucren **Inf**.

Sin embargo, a veces, determinados cálculos llevan a expresiones que no son números (representados por R como **NaN**, del inglés "**Not a Number**"). El valor **NaN** representa un valor indeterminado, también puede verse como un valor ausente.

Script de entrada en R

```
a <- 5/0     # División por cero, da infinito
a
b <- exp(a)  # Exponencial de infinito, da infinito
b
d <- b-b      # Infinito - Infinito es indeterminado
d
```

Consola de salida de R

```
> a <- 5/0     # División por cero, da infinito
> a
[1] Inf
> b <- exp(a)  # Exponencial de infinito, da infinito
> b
[1] Inf
> d <- b-b      # Infinito - Infinito es indeterminado
> d
[1] NaN
```

El objeto **NULL** representa el vacío en R; es de modo NULL y longitud 0, pero no debe ser confundido con un objeto vacío (sin elementos).

3.4 Números complejos

Para indicar explícitamente que un número introducido corresponde a un número complejo se debe emplear su propia sintaxis, esto es, **`a+bi`**.

Script de entrada en R

```
# Primer valor z1
z1 <- 5+2i
print(z1)
mode(z1)
class(z1)   # z1 es complex
storage.mode(z1)
print(sqrt(z1))   # Raíz de z1
# Segundo valor z2
z2 <- -1
print(z2)
class(z2)   # z2 es numeric
print(sqrt(z2))  # Problema: raíz de un número real negativo
# Tercer valor z3
z3 <- -1 + 0i
print(z3)
class(z3)   # z3 es complex
print(sqrt(z3))  # Ok: raíz del complejo -1+0i
```

Consola de salida de R

```
> # Primer valor z1
> z1<- 5+2i
> print(z1)
[1] 5+2i
> mode(z1)
[1] "complex"
> class(z1)  # z1 es complex
[1] "complex"
> storage.mode(z1)
[1] "complex"
> print(sqrt(z1))  # Raíz de z1
[1] 2.278724+0.438842i
> # Segundo valor z2
> z2 <- -1
> print(z2)
[1] -1
> class(z2)  # z2 es numeric
[1] "numeric"
> print(sqrt(z2))  # Problema: raíz de un número real negativo
[1] NaN
```

```
Warning message:
In sqrt(z2): Se han producido NaNs
> # Tercer valor z3
> z3 <- -1 + 0i
> print(z3)
[1] -1+0i
> class(z3)  # z3 es complex
[1] "complex"
> print(sqrt(z3))  # Ok: raíz del complejo -1+0i
[1] 0+1i
```

La parte real de un número complejo z puede obtenerse con la función **`Re(z)`**, la imaginaria con **`Im(z)`**, su módulo con **`Mod(z)`** y su argumento **`Arg(z)`**, como se muestra en el script:

Script de entrada en R

```
z <- 4+3i
Re(z)  # Parte real de z
Im(z)  # Parte imaginaria de z
Mod(z) # Módulo de z
Arg(z) # Argumento de z
```

Consola de salida de R

```
> z <- 4+3i
> Re(z)   # Parte real de z
[1] 4
> Im(z)  # Parte imaginaria de z
[1] 3
> Mod(z) # Módulo de z
[1] 5
> Arg(z) # Argumento de z
[1] 0.6435011
```

Debe recordarse que para un número complejo `a+bi` su módulo (valor absoluto) es $\sqrt{a^2 + b^2}$ y su argumento el ángulo (medido en radianes) entre el vector del origen al punto $(Re(z), Im(z))$ y el eje de las X (ver figura 12).

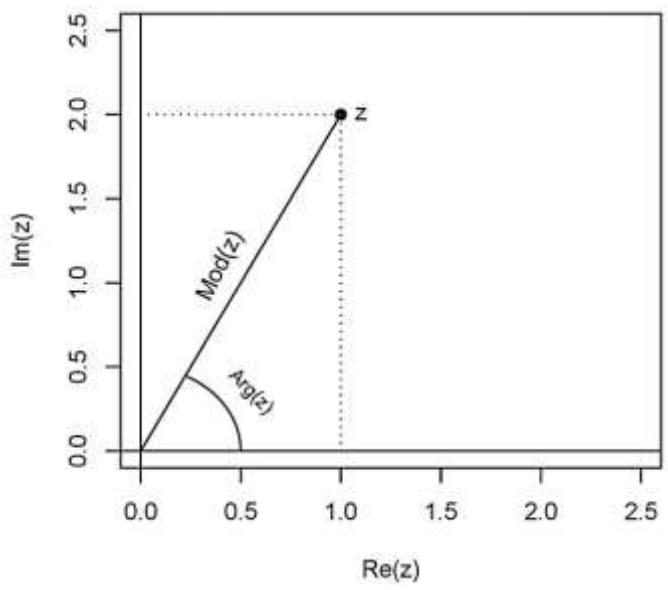


Figura 12. Características de un número complejo.
Tomada de (Lafaye, Drouihett & Liquet, 2011)

3.5 Coerción explícita

La mayoría de las funciones producen un error cuando el tipo de datos que esperan no coincide con los que se ponen en los argumentos. En tales situaciones hay dos caminos a seguir:

1. Comprobar el tipo de datos utilizando las funciones **`is.*()`**, que retornan un valor lógico.

   Una forma de verificar el modo de un objeto es usando la palabra "**is**", seguida de punto y el nombre del tipo a comprobar, por ejemplo:

   Script de entrada en R
   ```
   x <- 1
   is.numeric(x)
   ```

   Consola de salida de R
   ```
   > x <- 1
   > is.numeric(x)
    [1] TRUE
   ```

2. Forzar al tipo de datos deseado por coerción explícita, utilizando funciones del tipo **`as.tipo()`**, que fuerzan el tipo de datos, donde ***`tipo`*** se remplaza por el nombre del tipo

al que se quiere interpretar el objeto. En la tabla 5 se muestran los tipos más importantes que se pueden comprobar o forzar.

Tabla 5. Comprobación y coerción de tipos de datos

Tipo	**Comprobación**	**Coerción al tipo**
numeric	`is.numeric()`	`as.numeric()`
integer	`is.integer()`	`as.integer()`
double	`is.double()`	`as.double()`
logical	`is.logical()`	`as.logical()`
character	`is.character()`	`as.character()`
complex	`is.complex()`	`as.complex()`
vector	`is.vector()`	`as.vector()`
array	`is.array()`	`as.array()`
matrix	`is.matrix()`	`as.matrix()`
factor	`is.factor()`	`as.factor()`
list	`is.list()`	`as.list()`
function	`is.function()`	`as.function()`
NA	`is.na()`	-
NaN	`is.nan()`	-
NULL	`is.null()`	`as.null()`
ts	`is.ts()`	`as.ts()`

Ejemplo:

Script de entrada en R

```
x <- 6
class(x)
y <- as.character(x)
y
class(y)
```

Consola de salida de R

```
> x <- 6
> class(x)
[1] "numeric"
> y <- as.character(x)
> y
[1] "6"
> class(y)
[1] "character"
```

Para conocer si un elemento es **NA**, la comparación lógica "==" no es válida ya que **NA** denota un valor no disponible. Para ello se debe emplear la función **`is.na()`**, que retorna un valor lógico TRUE para los elementos del vector que son **NA**. De manera similar pueden usarse **`is.finite()`**, **`is.nan()`**, etc. En algunos casos R no puede aplicar una coerción a un objeto y produce resultados NAs.

Ejemplo:

Script de entrada en R

```
# Comprobando objetos
x <- c(1:10)
is.numeric(x)
is.vector(x)
is.complex(x)
is.character(x)
# Coerción de objetos
x <- 0:6
class(x)
as.logical(x)
as.character(x)
as.complex(x)
# Coerción a entero
y <- as.integer(6/3)
class(y)
# Comprobación de indeterminación
y <- 0/0
is.nan(y)
# Coerción imposible
z <- c("a", "b", "c")
z
as.numeric(z)
```

Consola de salida de R

```
> # Comprobando objetos
> x <- c(1:10)
> is.numeric(x)
[1] TRUE
> is.vector(x)
[1] TRUE
> is.complex(x)
[1] FALSE
> is.character(x)
[1] FALSE
> # Coerción de objetos
> x <- 0:6
> class(x)
[1] "integer"
> as.logical(x)
[1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
> as.character(x)
[1] "0" "1" "2" "3" "4" "5" "6"
> as.complex(x)
[1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
> # Coerción a entero
> y <- as.integer(6/3)
> class(y)
[1] "integer"
> # Comprobación de indeterminación
> y <- 0/0
> is.nan(y)
[1] TRUE
> # Coerción imposible
> z <- c("a", "b", "c")
> z
[1] "a" "b" "c"
> as.numeric(z)
[1] NA NA NA
Warning message:
NAs introduced by coercion
```

Ejercicios

1. ¿Cuál es el modo de los siguientes objetos? Responda primero sin usar R, luego confirme sus respuestas usando el comando R apropiado.

   a) c('a', 'b', 'c') b) 3.32e16 c) 1/3 d) sqrt(-2i)

2. Dado los números complejos a=3+5i y b=-2+8i, calcule: a*b, a+b y a/b, almacene sus resultados en las variables R1, R2 y R3. Imprima estos valores.
3. Escriba z = 5+5i en su forma polar.
4. Se dispone de la siguiente información sobre los metros cuadrados necesarios de madera para construir diferentes objetos: una puerta, una cama, una mesa y un armario, el precio del metro cuadrado de la madera a emplear (la misma para todos los objetos) es de $200 y el precio de la mano de obra para cada uno de los objetos. Se desea calcular el precio total de cada objeto, para lo cual trabajará con vectores. Al final imprima el precio total para cada objeto de madera dado.

Objeto	**Metros cuadrados**	**Mano de obra**
Puerta	1.9	600.00
Cama	3.0	800.00
Mesa	2.5	400.00
Armario	6.0	1000.00

5. De un edificio, a una altura de 15 m, se ha lanzado con un ángulo de 50 grados, un proyectil a una velocidad de 7 m/s. ¿Cuáles serán las alturas (coordenada y) del proyectil cada 0.1 m de distancia horizontal desde donde se lanzó y hasta los 11 m?

$$\boldsymbol{x = x_0 + v_{0x} t}$$

$$\boldsymbol{y = y_0 + v_{0y} * t - \frac{1}{2} g t^2}$$

Unidad 4. Vectores, matrices, arreglos y factores

4.1 Vectores

El vector es la estructura fundamental del lenguaje R. Un vector es un grupo de valores del mismo tipo. Puede ser un grupo de valores numéricos, de valores lógicos, caracteres o de cualquier otro tipo. Un valor escalar (por ejemplo, un número) se considera el vector más simple, de tamaño 1.

4.1.1 Creación de vectores

Hay diversas maneras de crear vectores, las cuales se presentan a continuación:

a) Usando el operador "**:**" para crear una secuencia de números, incrementando (o decrementando) en 1 en cada iteración.

Script de entrada en R

```
v1 <- 1:9
print(v1)
v2 <- 1.2:9
print(v2)
```

Consola de salida de R

```
> v1 <- 1:9
> print(v1)
[1] 1 2 3 4 5 6 7 8 9
> v2 <- 1.2:9
> print(v2)
[1] 1.2 2.2 3.2 4.2 5.2 6.2 7.2 8.2
```

En este último ejemplo las componentes de vector2 comienzan en 1.2 y se van incrementando en 1 hasta alcanzar la última componente que será menor o igual que 9.

La secuencia puede ser decreciente:

Script de entrada en R

```
v3 <- 40:13
print(v3)
class(v3)
```

Consola de salida de R

```
> v3 <- 40:13
> print(v3); class(v3)
 [1] 40 39 38 37 36 35 34 33 32 31 30 29 28 27 26 25 24 23 22 21
[22] 20 19 18 17 16 15 14 13
 [1] "integer"
```

b) Especificando las componentes del vector con la función **`c()`,** esto es ya conocido:

- $c(x_1, x_2, \dots, x_n)$ vector con los elementos $x_1, x_2, \dots, x_n$.
- $c(j{:}\,k)$ vector con los elementos: $j,\ j+1, \dots, k$.

Script de entrada en R

```
# Creando un vector v4 de tres números complejos
v4 <- c(4+2i, 2-1i, -8+0i)
print(v4)
```

Consola de salida de R

```
> # Creando un vector v4 de tres números complejos
> v4 <- c(4+2i, 2-1i, -8+0i)
> print(v4)
[1]  4+2i  2-1i -8+0i
```

c) A partir de ficheros de texto y la función **`scan()`.**

Se verá en detalles en la unidad 6.

d) Usando la función **`vector(tipo, cant)`**, donde:

- **`tipo`** es el tipo de datos de las componentes del vector,
- **`cant`** es la cantidad de componentes con que se crea el vector.

Script de entrada en R

```
# Un vector de enteros sin elementos
v <- vector("integer", 0)
print(v)
# Un vector numérico de tres ceros
w <- vector("numeric", 3)
print(w)
# Un vector lógico de 5 elementos FALSE
u <- vector("logical", 5)
print(u)
```

Consola de salida de R

```
> # Un vector de enteros sin elementos
> v <- vector("integer", 0)
> print(v)
integer(0)
> # Un vector numérico de tres ceros
> w <- vector("numeric", 3)
> print(w)
[1] 0 0 0
> # Un vector lógico de 5 elementos FALSE
> u <- vector("logical", 5)
> print(u)
[1] FALSE FALSE FALSE FALSE FALSE
```

e) Usando la función **`seq()`**.

`seq(from = vi, to = vf, by = incr)` o abreviadamente **`seq(vi, vf, incr)`**

que generaliza el operador "**:**" y permite obtener una mayor variedad de secuencias numéricas, donde:

`vi`: valor inicial de la secuencia,

`vf`: valor final de la secuencia,

`incr`: incremento (o decremento.)

Con `v<-seq(from = 5, to = 15, by = 1)` o abreviadamente `v<-seq(5, 15, 1)`

el vector v se formaría con los valores del 5 al 15.

Script de entrada en R

```
# Secuencia desde 5 hasta 15 de 1 en 1
v <- seq(from = 5, to = 15, by = 1)
print(v)
```

Consola de salida de R

```
> # Secuencia desde 5 hasta 15 de 1 en 1
> v <- seq(from = 5, to = 15, by = 1)
> print(v)
 [1]  5  6  7  8  9 10 11 12 13 14 15
```

Con `v <- seq(5, 15, 2)` la salida es:

```
[1] 5 7 9 11 13 15
```

Con la variante `v <- seq(from = vi, to = vf, length.out = k)` se obtiene una secuencia de longitud igual a length.out. Para ello se divide el recorrido de la secuencia (vf – vi) en (k – 1) partes iguales y devuelve una secuencia de k valores, los 2 extremos y los de las particiones.

Script de entrada en R

```
# Secuencia de longitud 6 desde 5 hasta 15
v <- seq(from = 5, to = 15, length.out = 6)
print(v)
```

Consola de salida de R

```
> # Secuencia de longitud 6 desde 5 hasta 15
> v <- seq(from = 5, to = 15, length.out = 6)
> print(v)
[1] 5 7 9 11 13 15
```

En los ejemplos anteriores la clase del resultado es **`numeric`** y no **`integer`**; de ser necesario el vector puede ser convertido a entero mediante la función **`as.integer()`**.

Con secuencias crecientes el valor inicial es por defecto 1. Entonces, por ejemplo, **`seq(5)`** es equivalente a **`seq(1,5)`**.

f) Empleando la función **`rep(x, times=n)`** - más corto y usado **`rep(x, n)`** - para crear un vector con **`n`** repeticiones de la secuencia **`x`**.

Script de entrada en R

```
v <- c(4, 8, -3)
w <- rep(v, times = 3)
print(w)
z <- rep(c("a","b"), 3)
print(z)
# repeticiones con reciclado incompleto
rep(1:5, 2, length.out=7)
```

Consola de salida de R

```
> v <- c(4, 8, -3)
> w <- rep(v, times = 3)
> print(w)
[1]  4  8 -3  4  8 -3  4  8 -3
> z <- rep(c("a","b"), 3)
```

```
> print(z)
[1] "a" "b" "a" "b" "a" "b"
> # repeticiones con reciclado incompleto
> rep(1:5, 2, length.out=7)
[1] 1 2 3 4 5 1 2
```

La función **rep** tiene la variante **rep(m:n, each = k)** que repite **k** veces los valores entre m y n. Por ejemplo:

Script de entrada en R

```
# repetir cada valor un número dado de veces
rep(1:5, each=2)
```

Consola de salida de R

```
> # repetir cada valor un número dado de veces
> rep(1:5, each=2)
 [1] 1 1 2 2 3 3 4 4 5 5
```

Por último, puede ser interesante dar a cada valor una cantidad de repeticiones prefijada. Observe los dos ejemplos siguientes:

Script de entrada en R

```
# Cantidad de repeticiones automáticas (una vez,..., cinco veces)
rep(1:5, 1:5)
# Cantidad de repeticiones señalada en un vector
rep(1:5, c(1,1,1,2,2))
```

Consola de salida de R

```
> # Cantidad de repeticiones automáticas (una vez,..., cinco veces)
> rep(1:5, 1:5)
 [1] 1 2 2 3 3 3 4 4 4 4 5 5 5 5 5
> # Cantidad de repeticiones señalada en vector
> rep(1:5, c(1,1,1,2,2))
[1] 1 2 3 4 4 5 5
```

4.1.2 Operaciones con vectores

Son operaciones básicas con vectores la concatenación, indización, inserción, pertenencia y asignación de nombres.

a) **Concatenación de vectores:**

Esta operación permite concatenar vectores para obtener uno nuevo, resultado de la concatenación, empleando la función **`c()`**.

Script de entrada en R

```
v <- c(4, 8, -3)
w <- c(5, 0, 3, 6)
# Concatenando v y w
vw <- c(v,w)
print(vw)
```

Consola de salida de R

```
> v <- c(4, 8, -3)
> w <- c(5, 0, 3, 6)
> # Concatenando v y w
> vw <- c(v,w)
> print(vw)
[1]  4  8 -3  5  0  3  6
```

La concatenación se define de la siguiente forma: dado dos vectores $x = (x_1, x_2, \ldots, x_m)$; $y = (y_1, y_2, \ldots, y_n)$, la concatenación de x e y es $c(x, y) = (x_1, x_2, \ldots, x_m, y_1, y_2, \ldots, y_n)$, y cumple las siguientes propiedades:

i. La concatenación no es conmutativa: $c(x, y) \neq c(y, x)$, salvo que $x = y$.

ii. La concatenación es asociativa: $c\big(x, c(y, z)\big) = c(c(x, y), z)$. Por otro lado: $c\big(x, c(y, z)\big) = c(x, y, z)$.

iii. Longitud lineal: $|c(x, y, \ldots)| = |x| + |y| + \cdots$. Donde $|y|$ denota la longitud del vector y.

b) **Indización:**

Permite acceder a los elementos individuales de un vector. Puede usarse el operador de indización **[ ]** para, mediante un valor de índice, determinar la componente del vector deseada. El primer elemento de un vector tiene índice 1.

Script de entrada en R

```
v <- c(4, 8, -3)
v1 <- v[1]
print(v1); print(v[2]); print(v[3])
```

Consola de salida de R

```
> v <-c(4, 8, -3)
> v1 <-v[1]
> print(v1); print(v[2]); print(v[3])
[1] 4
[1] 8
[1] -3
```

El operador **[ ]** también puede ser usado para formar subgrupos de elementos de un vector, indicando un rango de índices (con **:**) o un vector de índices enteros entre paréntesis. También puede ser usado para quitar elementos de un vector, para lo cual se debe anteponer el signo "-" a los índices de los valores a quitar o al vector.

Script de entrada en R

```
x <- c("a", "b", "c", "f", "d", "g")
# Elementos 1 al 4 del vector
y <- x[1:4]
print(y)
# Devuelve primer y tercer componente de x
y <- x[c(1,3)]
print(y)
# Elementos del vector x, quitando del 1ro al 4to
y <- x[-1:-4]
print(y)
# Similar, quitando primer y tercer elemento
y <- x[c(-1,-3)]
print(y)
```

Consola de salida de R

```
> x <- c("a", "b", "c", "f", "d", "g")
> # Elementos 1 al 4 del vector
> y <- x[1:4]
> print(y)
[1] "a" "b" "c" "f"
> # Devuelve primer y tercer componente de x
> y <- x[c(1,3)]
> print(y)
[1] "a" "c"
```

```
> # Elementos del vector x, quitando del 1ro al 4to
> y <- x[-1:-4]
> print(y)
[1] "d" "g"
> # Similar, quitando primer y tercer elemento
> y <- x[c(-1,-3)]
> print(y)
[1] "b" "f" "d" "g"
```

c) **Inserción de elementos en un vector**

Esta operación requiere especificar los elementos a insertar y la posición de inserción.

Script de entrada en R

```
x <- c(88,5,12,13)
print(x)
num <- 168
# Se inserta num (168) delante de 13 (que ocupa la posición 4)
y <- c(x[1:3], num, x[4])
print(y)
# Se inserta el vector (16,18) delante de 13
v <- c(16,18)
z <- c(x[1:3], v, x[4])
print(z)
```

Consola de salida de R

```
> x <- c(88,5,12,13)
> print(x)
[1] 88 5 12 13
> num <- 168
> # Se inserta num (168) delante de 13 (que ocupa la posición 4)
> y <- c(x[1:3], num, x[4])
> print(y)
[1] 88 5 12 168 13
> # Se inserta el vector (16,18) delante de 13
> v <- c(16,18)
> z <- c(x[1:3], v, x[4])
> print(z)
[1] 88 5 12 16 18 13
```

d) **Pertenencia de valores a un vector**

Mediante el operador **`%in%`** se comprueba si uno o varios valores en el lado izquierdo del operador aparecen en el vector del lado derecho.

Script de entrada en R

```
1 %in% c(3, 4, 5)          # 1 no pertenece al vector (3,4,5)
c(1, 2) %in% c(3, 4, 5)    # Ni 1 ni 2 pertenecen al vector (3,4,5)
c(1, 2, 3) %in% c(3, 4, 5) # Solo 3 pertenece al vector (3,4,5)
```

Consola de salida de R

```
> 1 %in% c(3, 4, 5)       # 1 no pertenece al vector (3,4,5)
[1] FALSE
> c(1, 2) %in% c(3, 4, 5) # Ni 1 ni 2 pertenecen al vector (3,4,5)
[1] FALSE FALSE
> c(1, 2, 3) %in% c(3, 4, 5) # Solo 3 pertenece al vector (3,4,5)
[1] FALSE FALSE TRUE
```

4.1.3 Asignando nombres a los elementos de un vector

Para una mejor comprensión de los datos que se manipulan, puede ser muy útil dar nombres a cada componente de un vector. Por ejemplo, suponga que hay 3 tipos de frutas: mangos, naranjas y guineos, en las cantidades 100, 50 y 300 respectivamente. Se puede asociar esos nombres de frutas a cada existencia empleando la función **`names()`**.

Script de entrada en R

```
frutas <- c(100, 50, 300)
print(frutas)
names(frutas) <- c("mango", "naranja", "guineo")
print(frutas)
# Forma un vector sfrutas con componentes nombradas mango y guineo
sfrutas <- frutas[c("mango","guineo")]
print(sfrutas)
```

Consola de salida de R

```
> frutas <- c(100, 50, 300)
> print(frutas)
[1] 100  50 300
> names(frutas) <- c("mango", "naranja", "guineo")
> print(frutas)
  mango naranja  guineo
    100      50     300
> # Forma un vector sfrutas con componentes nombradas mango y guineo
> sfrutas <- frutas[c("mango","guineo")]
> print(sfrutas)
 mango guineo
   100    300
```

El script anterior se puede escribir más simple, sin usar **`names(),`** simplemente igualando el nombre a la existencia, como se muestra a continuación:

```
frutas <- c("mango"=100, "naranja"=50, "guineo"=300)
print(frutas)
```

Aquí cada componente del vector es enlazada (usando el operador **=**) con un nombre.

Este nombramiento de las componentes se puede lograr también usando la función **`setNames(x,z)`**, donde **`x`** es una secuencia de valores y **`z`** es una secuencia de nombres. Ambos vectores deben tener igual longitud. A continuación, una modificación del ejemplo anterior usando **`setNames`**:

```
cant_frutas <-c (100, 50, 300)
nombres_frutas <- c("mango", "naranja", "guineo")
vfrutas <- setNames(cant_frutas, nombres_frutas)
print(vfrutas)
```

4.1.4 Operaciones aritméticas con vectores

Las operaciones que se realizan sobre vectores siguen las reglas siguientes:

a) Se realizan elemento a elemento de los vectores involucrados en la operación.

b) Reciclando el vector de menor longitud, aunque sería deseable que ambos tengan la misma longitud.

Ejemplos (ambos operandos de igual longitud):

Script de entrada en R

```
v1 <- c(3, 5, 6)
v2 <- c(-2, 7, 12)
# Operaciones aritméticas con v1 y v2
cat("Suma = ", vsuma <- v1 + v2, "\n")
cat("Resta = ", vresta <- v1 - v2, "\n")
cat("Producto = ", vprod <- v1 * v2, "\n")
cat("Potencia al cuadrado de v1 = ", vpot1 <- v1 ^ 2, "\n")
```

```
# Potencia de cada valor de v1 según el valor del vector indicado
vpot2 <- v1 ^ c(2,3,1)
print(vpot2)
```

Consola de salida de R

```
> v1 <- c(3, 5, 6)
> v2 <- c(-2, 7, 12)
> # Operaciones aritméticas con v1 y v2
> cat("Suma = ", vsuma <- v1 + v2, "\n")
Suma =  1 12 18
> cat("Resta = ", vresta <- v1 - v2, "\n")
Resta =  5 -2 -6
> cat("Producto = ", vprod <- v1 * v2, "\n")
Producto =  -6 35 72
> cat("Potencia al cuadrado de v1 = ", vpot1 <- v1 ^ 2, "\n")
Potencia al cuadrado de v1 = 9 25 36
> # Potencia de cada valor de v1 según el valor del vector indicado
> vpot2 <- v1 ^ c(2,3,1)
> print(vpot2)
[1] 9  125  6
```

Cuando los operandos tienen diferentes longitudes, se aplica el completamiento o **reciclaje,** según el cual el vector más corto se amplía al tamaño del más largo, agregando tantas componentes como sean necesarias, a partir de su primer elemento. Recuerde que el intérprete dará un mensaje de advertencia. Es importante recordar que el reciclaje del vector de menor longitud es virtual; el vector no cambia, sino que se hace representar por un operando temporal con el vector completado. De esto ya se habló en el epígrafe 2.3.1.

Por ejemplo, para sumar los vectores (3, 5, 6) y (-2, 7) el segundo vector es ampliado a 3 elementos, añadiéndole un valor, el -2 en este caso.

Script de entrada en R

```
v1 <- c(3,5,6)  # Longitud 3
v2 <- c(-2,7)   # Longitud 2, se completa añadiendo -2
vsuma <- v1+v2
print(vsuma)
print(v2)      # v2 mantiene su valor original
```

Consola de salida de R

```
> v1 <- c(3,5,6)  # Longitud 3
```

```
> v2 <- c(-2,7)   # Longitud 2, se completa añadiendo -2
> vsuma <- v1+v2
Warning message:
In v1 + v2 :
  longer object length is not a multiple of shorter object length
> print(vsuma)
[1]  1 12 4
> print(v2)      # v2 mantiene su valor original
[1] -2 7
```

Si se desea eliminar el mensaje de advertencia, se debe escribir antes de realizar la operación el comando **options(warn=-1).** Para volverlo a activar, se debe escribir **options(warn=1)**.

Para verificar si los elementos de un vector satisfacen una condición expresada en términos de comparaciones (menor; menor o igual; mayor; mayor o igual; diferente; igual), se emplean los operadores respectivos estudiados en la unidad 2 de la siguiente forma:

```
vsuma <- c(1, 12, 4)

vsuma <= 8

[1]  TRUE FALSE TRUE

vsuma <= c(1,3,20)

[1] TRUE FALSE TRUE

vsuma != c(1,3,20)

[1] FALSE TRUE TRUE
```

Para acceder a los elementos de un vector que satisfagan cierta condición, se pone el nombre del vector y entre corchetes un vector índice, el cual puede ser:

- Un vector lógico: en este caso el vector índice debe ser de la misma longitud que el vector del cual se quieren seleccionar los elementos. Los valores correspondientes a TRUE en el vector índice son seleccionados, los correspondientes a FALSE omitidos.

- Formación del **conjunto de validez** de un predicado: Dado el predicado P, el conjunto de validez de P se denota con {x | P(x)} y se lee "el conjunto de valores x tal que se cumple P(x)". {x | P(x)} se escribe en R como **`x[P(x)]`**.

Lo explicado anteriormente se ejemplifica en el siguiente script:

Script de entrada en R

```
vsuma <- c(1, 12, 4)
# Criterio lógico, devuelve primer y tercer componente
vsuma[c(T,F,T)]
# Forma vector con componentes de vsuma <= 8
cc <- vsuma[vsuma <= 8]
print(cc)
```

Consola de salida de R

```
> vsuma <- c(1, 12, 4)
> # Criterio lógico, devuelve primer y tercer componente
> vsuma[c(T,F,T)]
[1] 1 4
> # Forma vector con componentes de vsuma <= 8
> cc <- vsuma[vsuma <= 8]
> print(cc)
[1] 1 4
```

A continuación, se da un ejemplo del empleo de las operaciones lógicas **`all`** y **`any`**, introducidas en el epígrafe 2.3.3, usando vectores:

```
z <- c("Mon", "Tue", "Fri")

all(z %in% c("Tue", "Wed", "Thu", "Fri", "Sat", "Sun"))

 [1] FALSE

any(z %in% c("Tue", "Wed", "Thu", "Fri", "Sat", "Sun"))

 [1] TRUE
```

Por último, la función **`which()`** genera un vector numérico con las posiciones que cumplen la o las condiciones especificadas. Por ejemplo, suponga que existe un vector de 20 números

aleatorios normales de parámetros 0 y 1, y se desea seleccionar los que son positivos. Se ofrecen dos respuestas, sin usar y usando **which()**.

Script de entrada en R

```
#  Generando 12 números aleatorios con distribución normal (0,1)
normal <- rnorm(12,0,1)
print(normal, digits=5)
#  Selección sin usar which()
normal2 <- normal[normal>0]
print(normal2, digits=5)
# Selección usando which()
# Se buscan los índices cuyos valores en “normal” son positivos
ind <- which(normal > 0)
print(ind)
normal3 <- normal[ind]
print(normal3, digits=5)
```

Consola de salida de R

```
> #  Generando 12 números aleatorios con distribución normal (0,1)
> normal <- rnorm(12,0,1)
> print(normal, digits=5)
 [1] -0.26103 -1.52165  0.47596  0.28019  0.24544  1.07722 -1.44837
 [8]  1.26760  1.00523 -0.46886 -0.87815  0.12385
> # Selección sin usar which()
> normal2 <- normal[normal>0]
> print(normal2, digits=5)
[1] 0.47596 0.28019 0.24544 1.07722 1.26760 1.00523 0.12385
> # Selección usando which()
> # Se buscan los índices cuyos valores en “normal” son positivos
> ind <- which(normal > 0)
> print(ind)
[1]  3  4  5  6  8  9 12
> normal3 <- normal[ind]
> print(normal3, digits=5)
[1] 0.47596 0.28019 0.24544 1.07722 1.26760 1.00523 0.12385
```

Las funciones **which.min(x)** y **which.max(x)** retornan respectivamente la posición del (primer) mínimo y del (primer) máximo de un vector numérico o lógico **x**.

La función **which()** permite remplazar ciertos elementos del vector por otro, por ejemplo,

```
x[which(x == 0)] <- 10
```

permite remplazar cada componente del vector x igual a cero por el valor 10.

En la tabla 6 se exponen otras funciones con vectores, adicionales a las aprendidas en el epígrafe 2.4. Se ejemplificarán usando los vectores:

```
x <- c(6, 4, 3.5, 2, 5)

y <- c(1, 3, 3, 1, 5)

z <- c(8, 3, 3, 2, 9)
```

El parámetro **`method`** que aparece en algunas de las funciones especifica el tipo de coeficiente que se desea calcular: "pearson", "kendall" o "spearman". Por defecto asume el de Pearson.

Tabla 6. Funciones con vectores, significado y ejemplos

Función	**Significado**	**Resultado**
cor(x,y, method)	Coeficiente de correlación entre los vectores x e y.	`> cor(x,y)` `[1] 0.2167304`
cov(x,y, method)	Covarianza entre los vectores x e y.	`> cov(x,y)` `[1] 0.55`
cummax(x)	Vector formado por los máximos acumulados de $x = (x_1, x_2, \ldots, x_n)$, o sea, $(x_1, max(x_1, x_2), \ldots, max(x_1, \ldots, x_n))$.	`> cummax(x)` `[1] 6 6 6 6 6`
cummin(x)	Vector formado por los mínimos acumulados de x.	`> cummin(x)` `[1] 6.0 4.0 3.5 2.0 2.0`
cumprod(x)	Vector formado por el producto acumulado del vector $x = (x_1, x_2, \ldots, x_n)$, o sea, $(x_1, x_1 * x_2, \ldots, x_1 * x_2 * \ldots * x_n)$.	`> cumprod(x)` `[1] 6  24  84 168 840`
cumsum(x)	Vector formado por la suma acumulada del vector $x = (x_1, x_2, \ldots, x_n)$, o sea, $(x_1, x_1 + x_2, \ldots, x_1 + x_2 + \cdots + x_n)$.	`> cumsum(x)` `[1]  6.0  10.0  13.5  15.5 20.5`
length(x)	Cantidad de elementos del vector x	`> length(x)` `[1] 5`

mean(x)	Media aritmética de los valores del vector x.	`> mean(x)` `[1] 4.1`
median(x)	Mediana de los valores del vector x.	`> median(x)` `[1] 4`
order(x)	Indica la posición que tendrían los valores de x ordenados ascendentemente.	`> order(x)` `[1] 4 3 2 5 1`
pmax(x,y,z)	Vector de longitud igual al más largo de x, y o z, formado por los valores máximos de cada posición.	`> pmax(x,y,z)` `[1] 8.0 4.0 3.5 2.0 9.0`
pmin(x,y,z)	Vector de longitud igual al más largo de x, y o z, formado por los valores mínimos de cada posición.	`> pmin(x,y,z)` `[1] 1 3 3 1 5`

quantile(x)	Retorna el valor mínimo, el primer cuartil, la mediana, el tercer cuartil y el valor máximo del vector x.	`> quantile(x)`
		`> quantile(x)`
		`  0%  25%  50%  75% 100%`
		`2.0  3.5  4.0  5.0  6.0`

rank(x)	Le asigna rangos a los elementos de x.	`> rank(x)` `[1] 5 3 2 1 4`
rev(x)	Invierte la colocación de los elementos del vector x	`> rev(x)` `[1] 5.0 2.0 3.5 4.0 6.0`
sort(x)	Ordena los elementos del vector x, por defecto de menor a mayor. `sort(x, decreasing=TRUE)` ordena de mayor a menor.	`> sort(x)` `[1] 2.0 3.5 4.0 5.0 6.0`

unique(x)	Elimina la aparición repetida de elementos de un vector	`> unique(x)`
		`[1] 6.0 4.0 3.5 2.0 5.0`
		`> unique(y)`
		`[1] 1 3 5`

var(x)	Devuelve la cuasi varianza de los valores del vector x.	`> var(x)` `[1] 2.3`

4.1.5 Vectores de caracteres

Las cadenas de caracteres se utilizan para nombrar cosas u objetos del mundo. Igual que en el caso de los números, en R la clase **character** no se refiere a una cadena de caracteres aislada, sino a un vector que contiene cero o más caracteres. En el siguiente ejemplo se crean dos vectores de caracteres, uno con los nombres de un conjunto de personas y otro con los meses en que nacieron, respectivamente.

Script de entrada en R

```
nombre <- c("Juana", "Juan", "Pedro", "Luis", "Maria", "Gloria",
            "Bertha", "Rosa", "Rafael", "Ernesto", "Elsa")
mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio", "Julio",
             "Febrero", "Abril", "Mayo", "Abril", "Mayo")
print(nombre); print(mes.nac)
# Formas de mostrar nombre y mes de nacimiento de la primera persona
print(nombre[1]); print(mes.nac[1])
print(c(nombre[1], mes.nac[1]))
```

Consola de salida de R

```
> nombre <- c("Juana", "Juan", "Pedro", "Luis", "Maria", "Gloria",
+             "Bertha", "Rosa", "Rafael", "Ernesto", "Elsa")
> mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio", "Julio",
+              "Febrero", "Abril", "Mayo", "Abril", "Mayo")
> print (nombre); print(mes.nac)
 [1] "Juana"   "Juan"   "Pedro"   "Luis"   "Maria"   "Gloria"   "Bertha"
 [8] "Rosa"   "Rafael"   "Ernesto"   "Elsa"
 [1] "Enero"  "Febrero"  "Abril"  "Febrero"  "Julio"  "Julio"  "Febrero"
 [8] "Abril"  "Mayo"  "Abril"  "Mayo"
> # Formas de mostrar nombre y mes de nacimiento de la primera persona
> print(nombre[1]); print(mes.nac[1])
[1] "Juana"
[1] "Enero"
> print(c(nombre[1], mes.nac[1]))
[1] "Juana" "Enero"
```

En el epígrafe 2.1 se mostraron algunos vectores de caracteres predefinidos que posee el lenguaje R: **LETTERS**, **letters**, **month.name** y **month.abb**.

Ejemplo:

Script de entrada en R

```
LETTERS
month.abb
```

Consola de salida de R

```
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P"
[17] "Q" "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
> month.abb
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct"
[11] "Nov" "Dec"
```

Para chequear si un valor es un carácter se usa la función **`is.character()`**, en tanto para forzar un valor al tipo carácter se emplea la función **`as.character()`**.

```
> x <- "Aprendizaje del lenguaje R"

> class(x)

[1] "character"

> is.character(x)

[1] TRUE
```

Una cadena que representa un valor numérico puede ser forzada a número, en caso contrario resulta en NA.

```
> as.numeric("2.5")

[1] 2.5

> as.numeric("lenguaje")

[1] NA

Warning message:

NAs introduced by coercion
```

Una operación frecuente con cadenas de caracteres es su **concatenación**, la cual ya se introdujo en el epígrafe 4.1.2, pero para concatenar vectores mediante el empleo de la función

`c()`. Para la concatenación de cadenas de caracteres, que consiste en unir varias cadenas para formar una sola, donde se puedan introducir variaciones sintácticas en su estructura, se puede emplear la función **`paste()`:**

`paste (`s_1`,...,`s_n`, sep = " ", others...)`

donde

$s_1,...,s_n$: objetos de R que son convertidos a vectores de caracteres.

`sep`: separador entre las cadenas, por defecto un espacio en blanco.

`others...`: otros parámetros. (Para más información consultar el manual o la ayuda del R).

El valor de la concatenación puede asignarse a una variable.

Ejemplo:

Script de entrada en R

```
nombre <- c("Juana", "Juan", "Pedro", "Luis", "Maria", "Gloria")
mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio", "Julio")
# La función paste() sin separador (asume espacio)
paste(nombre[1],"nació en el mes de",mes.nac[1])
# Usando como separador el asterisco "*"
paste(nombre[1],"nació en el mes de",mes.nac[1], sep="*")
# Usando como separador el punto "."
x <- paste("El número", 1 ,"de la lista nació en",mes.nac[1],sep=".")
print(x)
```

Consola de salida de R

```
> nombre <- c("Juana", "Juan", "Pedro", "Luis", "Maria", "Gloria")
> mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio", "Julio")
> # La función paste() sin separador (asume espacio)
> paste(nombre[1],"nació en el mes de",mes.nac[1])
[1] "Juana nació en el mes de Enero"
> # Usando como separador el asterisco "*"
> paste(nombre[1],"nació en el mes de",mes.nac[1], sep="*")
[1] "Juana*nació en el mes de*Enero"
> # Usando como separador el punto "."
> x <- paste("El número",1,"de la lista nació en",mes.nac[1],sep=".")
> print(x)
[1] "El número.1.de la lista nació en.Enero"
```

Por otra parte, la función **`paste0()`** no deja separador entre las cadenas y es algo más eficiente. En el siguiente ejemplo se muestra el uso de la función `paste()` y `paste0()` para generar nombres de objetos con un prefijo fijo.

Script de entrada en R

```
labs1 <- paste(c("X"),1:5)
labs1
labs2 <- paste0(c("X"),1:5)
labs2
labs <- paste(c("X","Y"),1:5,sep=",")
labs
```

Consola de salida de R

```
> labs1 <- paste(c("X"),1:5)
> labs1
[1] "X 1" "X 2" "X 3" "X 4" "X 5"
> labs2 <- paste0(c("X"),1:5)
> labs2
[1] "X1" "X2" "X3" "X4" "X5"
> labs <- paste(c("X","Y"),1:5,sep=",")
> labs
[1] "X,1" "Y,2" "X,3" "Y,4" "X,5"
```

Otras operaciones con cadenas son:

- **`nchar(x)`** – devuelve la longitud de cada cadena contenida en x.

  Script de entrada en R

  ```
  s <- c("programa", "R", "cadena")
  # Cantidad de caracteres de cada cadena en s
  nchar(s)
  nchar("RStudio")
  ```

  Consola de salida de R

  ```
  > s <- c("programa", "R", "cadena")
  > # Cantidad de caracteres de cada cadena en s
  > nchar(s)
  [1] 8 1 6
  > nchar("RStudio")
  [1] 7
  ```

- **`substr(x, ini, fin)`** – devuelve la subcadena de **`x`** comenzando en el carácter ubicado en la posición **`ini`** hasta el carácter ubicado en la posición **`fin`**.

Ejemplo:

`substr("abcdef", 2, 4)` devuelve la subcadena **`"bcd"`**.

Una variante es la función **`substring()`**, que permite extraer más de una subcadena.

Ejemplo:

`substring("abcdef",first=1:3,last=2:4)` devuelve las subcadenas "ab", "bc" y "cd" que corresponden a aquellas subcadenas que comienzan por el carácter de posición 1, 2 y 3, y terminan por los de posición 2, 3 y 4, respectivamente.

- **`sub(viejo, nuevo, x)`** – sustituye la primera ocurrencia de la subcadena **`viejo`** por **`nuevo`** dentro de la cadena **`x`**. La variante **`gsub(viejo, nuevo, x)`** sustituye todas las ocurrencias.

  Script de entrada en R

```
# Remplaza primera aparición de "pro" con "penta"
sub("pro", "penta", "programapro")
# Remplaza toda aparición de "pro" con "penta"
gsub("pro", "penta", "programapro")
gsub("pro", "penta", c("programa", "proyecto"))
```

  Consola de salida de R

```
> # Remplaza primera aparición de "pro" con "penta"
> sub("pro", "penta", "programapro")
[1] "pentagramapro"
> # Remplaza toda aparición de "pro" con "penta"
> gsub("pro", "penta", "programapro")
[1] "pentagramapenta"
> gsub("pro", "penta", c("programa", "proyecto"))
[1] "pentagrama" "pentayecto"
```

- **`grep(x,y...)`** – localiza una subcadena **`x`** dentro de un vector de cadenas **`y`**, devolviendo los índices de las cadenas de **`y`** que contienen a la subcadena **`x`**.

  Por ejemplo, **`grep("i",c("Santiago", "Habana", "Artemisa"))`** devuelve 1 y 3.

- **`toupper(x)`** – lleva todos los objetos de caracteres en **`x`** a mayúscula.

- **`tolower(x)`** – lleva todos los objetos de caracteres en **`x`** a minúscula.

4.1.6 Operaciones con conjuntos

Un conjunto es representable en R con un vector donde no hay repetición de ninguno de sus elementos. En el lenguaje R un vector puede tener elementos repetidos, en ese caso un conjunto es lograble aplicando la función **`unique()`** a dicho vector. Algunas operaciones conjuntuales de R se muestran a continuación, acompañadas de algunos ejemplos:

a) Pertenencia de un valor a un conjunto usando la operación **`%in%`** vista en el epígrafe 4.1.2.

b) Comparación en igualdad: **`setequal(c1, c2)`**, donde **`c1`** y **`c2`** son conjuntos, devuelve TRUE si ambos conjuntos contienen los mismos elementos.

Script de entrada en R

```
v <- c("B", "A")   # Un conjunto
w <- c("A", "B")   # Otro conjunto
setequal(v, w)     # Se comparan en igualdad
```

Consola de salida de R

```
> v <- c("B", "A")   # Un conjunto
> w <- c("A", "B")   # Otro conjunto
> setequal(v, w)     # Se comparan en igualdad
[1] TRUE
```

c) Inclusión de conjuntos ($\subseteq$): **`all(c1 %in% c2)`**, devuelve TRUE si $c1 \subseteq c2$ y FALSE en otro caso.

d) Unión de dos conjuntos: **`union(c1,c2)`**, devuelve el conjunto formado por los elementos que están en **`c1`** o en **`c2`**.

e) Intersección de dos conjuntos: **`intersect(c1,c2)`**, devuelve el conjunto formado por los elementos que están tanto **`c1`** como en **`c2`**.

f) Diferencia de dos conjuntos: **`setdiff(c1,c2)`**, devuelve el conjunto formado por los elementos que están en **`c1`** y no están en **`c2`**.

Script de entrada en R

```
x = c(1, 2, 3)
y = c(2, 4)
```

```
union(x, y)       # 1 2 3 4
intersect(x, y)   # 2
setdiff(x, y)     # 1 3
all(y %in% x)     # FALSE
```

Consola de salida de R

```
> x = c(1, 2, 3)
> y = c(2, 4)
> union(x, y)       # 1 2 3 4
[1] 1 2 3 4
> intersect(x, y)   # 2
[1] 2
> setdiff(x, y)     # 1 3
[1] 1 3
> all(y %in% x)     # FALSE
[1] FALSE
```

Las operaciones conjuntuales pueden aplicarse a vectores con repetición de elementos (similar a **multiconjuntos**), como se muestra a continuación:

```
x <- c(1,2,3,3)  # Se repite 3

y <- c(2,4,4)    # Se repite 4

union(x,y)       # 1 2 3 4

intersect(x,y)   # 2

setdiff(x,y)     # 1 3

z <- c(2,2,4)    # Se repite 2

setequal(y,z)    # TRUE
```

Pueden comprobarse las respuestas a las operaciones efectuadas, que aparecen al lado de las instrucciones como comentarios. Se observa que la repetición de elementos no afecta la ejecución de la operación.

El conjunto vacío se representa por `c()` que equivale a `NULL`. Hay una diferencia entre la interpretación que da R al conjunto vacío a la que da las matemáticas, pues:

- `c(c(), x) = c(x) = x`

- `c(x, c()) = x`

Entonces el conjunto vacío no puede ser elemento de un vector que representa a un conjunto y no puede ser probado para membresía en un conjunto. Por ello:

```
c() %in% c(3,4,c())
```

devuelve el resultado `logical(0)`[6], o sea, `FALSE`.

4.1.7 Otras operaciones con vectores

La función **`duplicated()`** marca con un valor lógico si un elemento dentro de un vector ya ha aparecido antes. Empleando el vector x = (1, 2, 3, 3), **`duplicated(x)`** retorna:

```
> FALSE FALSE FALSE TRUE
```

La función **`anyDuplicated(x)`** devuelve la posición del primer elemento del vector **`x`** que ya han aparecido antes dentro de su composición. Por ello **`anyDuplicated(x)`** devolvería 4. Comprobar **`anyDuplicated(x)> 0`** es una manera rápida de chequear si un vector contiene componentes repetidas.

4.2 Matrices

Una matriz puede verse como un vector de dos dimensiones con un atributo adicional **dim**, el cual en el caso de las matrices es un vector entero de dos elementos, el número de filas y el número de columnas que componen la matriz. A continuación, se muestra un esquema de matriz de nombre **tabla**, con **2** filas y **3** columnas, y como se nombra cada elemento de la misma.

	Columna 1	Columna 2	Columna 3
Fila 1	**tabla[1,1]**	**tabla[1,2]**	**tabla[1,3]**
Fila 2	**tabla[2,1]**	**tabla[2,2]**	**tabla[2,3]**

[6] `logical(n)` para n $\neq 0$ es TRUE.

4.2.1 Creación de matrices

En la construcción de matrices se emplean ciertas funciones, las cuales se muestran a continuación a través de ejemplos:

a) A partir de un vector, usando la función **`dim()`**.

Esta función permite distribuir un vector en dos dimensiones dadas. También se usa para retornar las dimensiones (filas, columnas) de una matriz.

Script de entrada en R

```
# Se generan 20 números aleatorios N(1.5,4)
set.seed(0)  # Semilla de generación
z <- rnorm(20, 1.5, 2)
print(z, digits = 4)
is.vector(z)
# Se redimensiona el vector z a una matriz de 4x5
dim(z) <- c(4,5)
class(z)
print(z, digits = 4)
dim(z)
```

Consola de salida de R

```
> # Se generan 20 números aleatorios N(1.5,4)
> set.seed(0)  # Semilla de generación
> z <- rnorm(20, 1.5, 2)
> print(z, digits = 4)
 [1]  1.0515  2.2548  1.7667  3.1084  1.3858  2.5072  3.6715
 [8]  0.1181 -1.0692  1.5935  1.0286  0.4142  0.6334  0.2011
 [15]  2.9535  3.8038  3.4843  0.6410  3.9766  0.9413
> is.vector(z)
[1] TRUE
> # Se redimensiona el vector z a una matriz de 4x5
> dim(z) <- c(4,5)
> class(z)
[1] "matrix"
> print(z, digits = 4)
       [,1]    [,2]     [,3]    [,4]    [,5]
[1,] 4.0259  2.3293  1.48847 -0.7953  2.0044
[2,] 0.8475 -1.5799  6.30931  0.9211 -0.2838
[3,] 4.1596 -0.3571  3.02719  0.9016  2.3714
[4,] 4.0449  0.9106 -0.09802  0.6770 -0.9751
> dim(z)
[1] 4 5
```

b) Especificando sus elementos individualmente.

Se muestra con el siguiente script para crear la matriz $\begin{pmatrix} \mathbf{1} & \mathbf{3} \\ \mathbf{2} & \mathbf{4} \end{pmatrix}$. La función **`matrix()`** es empleada en este caso para crear un objeto de la clase matrix con dos filas y dos columnas.

```
y <- matrix(nrow = 2, ncol = 2)
y[1,1] <- 1
y[2,1] <- 2
y[1,2] <- 3
y[2,2] <- 4
print(y)
```

c) A partir de un vector, usando la función **`matrix`**.

Las matrices también se pueden crear de manera flexible por medio de la función primitiva **`matrix(data = NA, nrow = 1, ncol = 1, byrow = FALSE, dimnames = NULL),`** que da la posibilidad de construir la matriz por filas o por columnas. Sus parámetros son:

- **`data`**: vector con los datos de la matriz,
- **`nrow`**: cantidad de filas (por defecto una),
- **`ncol`**: cantidad de columnas (por defecto una),
- **`byrow`**: TRUE si se forma por filas, FALSE (por defecto) si se forma por columnas,
- **`dimnames`**: lista con los nombres de las filas y columnas de la matriz.

En los ejemplos que siguen la matriz se forma a partir de un vector completamente especificado:

Script de entrada en R

```
# matriz 3x3 formada por columnas
```

```
mat3 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)
print(mat3)
```

Consola de salida de R

```
> # matriz 3x3 formada por columnas
> mat3 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)
> print(mat3)
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
```

Al especificar `ncol=3`, el vector suministrado se considera de 3 columnas (y automáticamente 3 filas, por tener 9 elementos). Los elementos se disponen por columnas, pues **`byrow`** no se especifica y por defecto es FALSE. El mismo resultado se hubiera obtenido al ejecutar el script:

```
mat3col <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3)

print(mat3col)
```

Para realizar la construcción por filas, se especifica **`byrow`** con valor TRUE.

```
mat3fil <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 3,

           byrow = TRUE)

print(mat3fil)

        [,1] [,2] [,3]

 [1,]     1    2    3

 [2,]     4    5    6

 [3,]     7    8    9
```

Si se trata de crear una matriz con mayor capacidad que la cantidad de elementos proporcionados, el sistema R da un mensaje de advertencia y se recicla con esos elementos hasta que se llene la matriz:

```
> matExced <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 2)
Warning message:
   In matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), nrow = 2) :
   la longitud de los datos [9] no es un submúltiplo o múltiplo
   del número de filas [2] en la matriz
> print(matExced)
     [,1] [,2] [,3] [,4] [,5]
[1,]    1    3    5    7    9
[2,]    2    4    6    8    1
```

El valor de fila 2 y columna 5 se creó reciclando el vector de creación.

d) Creación de una **matriz diagonal**

De forma muy intuitiva se puede emplear la función **`diag()`**. Por ejemplo, el siguiente comando crea una matriz de 3x3 con los elementos de la diagonal en 1, o sea, la matriz identidad de orden 3:

```
matdiagonal <- diag(1, nrow = 3)
```

Si se agrega ahora el comando:

```
diag(matdiagonal) <- c(1,2,3)
```

y se ejecuta, entonces matdiagonal se convierte en la matriz:

```
1    0    0
0    2    0
0    0    3
```

Esto también se puede realizar a través de la instrucción:

```
matdiagonal <- diag(c(1,2,3), nrow = 3)
```

e) Construcción de una matriz con las funciones **rbind()** y **cbind()**

Otra forma de construir una matriz es a través de las funciones **rbind()** y **cbind()**[7], dándoles como argumentos las filas o las columnas individuales que la componen, respectivamente. A modo de ejemplo, el siguiente programa crea matrices por filas (m1) y columnas (m2) a partir de dos vectores y las imprime.

```
# Se forma m1 con las dos filas dadas como argumentos
m1 <- rbind(c(1.5, 3.2, -5.5), c(0, -1.1, 6))
print(m1)
# Se forma m2 con las tres columnas dadas como argumentos
m2 <- cbind(c(1.5, 3.2), c(-5.5, 6), c(0, -1.1))
print(m2)
```

f) Nombres para filas y columnas

A las filas y las columnas de una matriz se les pueden asignar nombres, al igual que a los vectores, los que pueden ser después consultados o usados como índices, mediante las funciones **`rownames()`** y **`colnames()`**, como muestra el script siguiente:

Script de entrada en R

```
# Formando una matriz por filas de 5x4 con elementos del 11 al 30
mf <- matrix(11:30, nrow=5, ncol=4, byrow=TRUE)
print(mf)
# Dando nombre f1, f2, f3, f4 y f5 a las filas de mf
rownames(mf) <- paste0(" f",1:5)
# Dando nombre c1, c2, c3 y c4 a las columnas de mf
colnames(mf) <- paste0(" c",1:4)
print(mf)
# Consultando nombres de columnas
colnames(mf)
# Consultando nombres de filas
rownames(mf)
```

[7] Bind: ligar, unir. Estas funciones también se pueden usar para unir por filas o columnas dos o más matrices.

Consola de salida de R

```
> # Formando matriz por filas de 5x4 con elementos del 11 al 30
> mf <- matrix(11:30, nrow=5, ncol=4, byrow=TRUE)
> print(mf)
     [,1] [,2] [,3] [,4]
[1,]   11   12   13   14
[2,]   15   16   17   18
[3,]   19   20   21   22
[4,]   23   24   25   26
[5,]   27   28   29   30
> # Dando nombre f1, f2, f3, f4 y f5 a las filas de mf
> rownames(mf) <- paste0(" f",1:5)
> # Dando nombre c1, c2, c3 y c4 a las columnas de mf
> colnames(mf) <- paste0(" c",1:4)
> print(mf)
       c1   c2   c3   c4
 f1    11   12   13   14
 f2    15   16   17   18
 f3    19   20   21   22
 f4    23   24   25   26
 f5    27   28   29   30
> # Consultando nombres de columnas
> colnames(mf)
[1] " c1 " " c2 " " c3 " " c4 "
> # Consultando nombres de filas
> rownames(mf)
[1] " f1 " " f2 " " f3 " " f4 " " f5 "
```

Otra forma de asignarle nombres a las filas y columnas de una matriz es a través del argumento **`dimnames`** de la función **`matrix`**, el cual representa una lista[8] de dos componentes, cada componente es un vector con los nombres de las filas y columnas, respectivamente. En el ejemplo anterior se pudo haber usado:

```
matrix(11:30, nrow=5, ncol=4, byrow=TRUE, dimnames=list(c("f1",
"f2", "f3", "f4", "f5"), c("c1", "c2", "c3", "c4")))
```

g) Cambios en las dimensiones de la matriz

[8] Las listas se estudiarán en la unidad 5

Las dimensiones de una matriz pueden ser cambiadas, como se muestra a continuación, si al programa del ejemplo anterior se le agregan los comandos:

```
dim(mf) <- c(4,5)
print(mf)
```

Si se ejecutan esas líneas, daría como resultado:

```
> dim(mf) <- c(4,5)
> print(mf)
     [,1] [,2] [,3] [,4] [,5]
[1,]   11   27   24   21   18
[2,]   15   12   28   25   22
[3,]   19   16   13   29   26
[4,]   23   20   17   14   30
```

Debe asegurarse que la cantidad de elementos de la matriz redimensionada sea igual al de la matriz original.

4.2.2 Acceso a un elemento particular de una matriz (indización)

Para tener acceso a un elemento particular de una matriz, se usa el operador **[]** y dentro de este y separados por coma, la fila y columna del elemento deseado. Si el valor de la fila o columna se omite, se muestran todos los elementos de la columna o fila solicitada, respectivamente.

Ejemplo:

Script de entrada en R

```
z <- matrix(c(10:21), ncol = 4)
print(z)
# Accediendo a un elemento individual (fila 3, columna 2)
a <- z[3,2]
print(a)
```

```
class(a)
# 8vo elemento considerando a z un vector
b <- z[8]
print(b)
class(b)
# Accediendo a una fila y a una columna completa respectivamente
fila2 <- z[2, ]      # Fila 2 completa
columna3 <- z[, 3]   # Columna 3 completa
print(fila2)
print(columna3)
```

Consola de salida de R

```
> z <- matrix(c(10:21), ncol = 4)
> print(z)
     [,1] [,2] [,3] [,4]
[1,]   10   13   16   19
[2,]   11   14   17   20
[3,]   12   15   18   21
> # Accediendo a un elemento particular (fila 3, columna 2)
> a <- z[3,2]
> print(a)
[1] 15
> class(a)
[1] "integer"
> # 8vo elemento considerando a z un vector
> b <- z[8]
> print(b)
[1] 17
> class(b)
[1] "integer"
> # Accediendo a una fila y a una columna completa, respectivamente
> fila2 <- z[2, ]      # Fila 2 completa
> columna3 <- z[, 3]   # Columna 3 completa
> print(fila2)
[1] 11 14 17 20
> print(columna3)
[1] 16 17 18
```

También se puede tener acceso a una fila o a una columna de una matriz empleando su nombre. Observe el siguiente script y su ejecución:

Script de entrada en R

```
z <- matrix(c(10:21), ncol = 4)
rownames(z) <- c("f1", "f2", "f3")
colnames(z) <- c("c1", "c2", "c3", "c4")
```

```
print(z)
#  Fila 1
Fila_f1 <- z["f1",]
print(Fila_f1)
#  Columna 1
Col_c1 <- z[,"c1"]
print(Col_c1)
```

Consola de salida de R

```
> z <- matrix(c(10:21), ncol = 4)
> rownames(z) <- c("f1", "f2", "f3")
> colnames(z) <- c("c1", "c2", "c3", "c4")
> print(z)
   c1 c2 c3 c4
f1 10 13 16 19
f2 11 14 17 20
f3 12 15 18 21
> #  Fila 1
> Fila_f1 <- z["f1",]
> print(Fila_f1)
c1 c2 c3 c4
10 13 16 19
> #  Columna 1
> Col_c1 <- z[,"c1"]
> print(Col_c1)
f1 f2 f3
10 11 12
```

Observar que los nombres de las filas y columnas deben escribirse entre comillas dobles.

4.2.3 Operaciones algebraicas con matrices

Todas las operaciones aritméticas aplicables a vectores (**+**, **-**, *****, **/**, etc) son válidas para las matrices, siempre que las matrices operandos tengan las mismas dimensiones[9]. Las operaciones se aplican elemento a elemento de iguales posiciones dentro de la matriz. Por ejemplo:

Script de entrada en R

```
m1 <- matrix(1:6, nrow = 2, byrow = TRUE)
m2 <- rbind(c(3,0,1), c(-1,3, 0))
```

[9] Con las matrices no es aplicable el reciclaje para esas operaciones como con los vectores.

```
print(m1)
print(m2)
Suma <- m1 + m2   # Suma de matrices
print(Suma)
ProductoP <- m1 * m2   # Producto posicional
print(ProductoP)
DivisionP <- m2 / m1   # División posicional
print(DivisionP, digits =4)
```

Consola de salida de R

```
> m1 <- matrix(1:6, nrow = 2, byrow = TRUE)
> m2 <- rbind(c(3,0,1), c(-1,3, 0))
> print(m1)
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
> print(m2)
     [,1] [,2] [,3]
[1,]    3    0    1
[2,]   -1    3    0
> Suma <- m1 + m2   # Suma de matrices
> print(Suma)
     [,1] [,2] [,3]
[1,]    4    2    4
[2,]    3    8    6
> ProductoP <- m1 * m2   # Producto posicional
> print(ProductoP)
     [,1] [,2] [,3]
[1,]    3    0    3
[2,]   -4   15    0
> DivisionP <- m2 / m1   # División posicional
> print(DivisionP, digits =4)
      [,1]  [,2]     [,3]
[1,]  3.00   0.0   0.3333
[2,] -0.25   0.6   0.0000
```

Algunas operaciones matriciales que se pueden realizar en R son:

a) **Multiplicación matricial** con el operador **%*%**.

Por ejemplo, sean:

En R: **A %*% B**

$A = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$, $B = \begin{pmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{pmatrix}$, entonces $A * B = \begin{pmatrix} 47 & 52 & 57 \\ 64 & 71 & 78 \\ 81 & 90 & 99 \end{pmatrix}$.

Se muestra el programa y su ejecución.

Script de entrada en R

```
A <- matrix(1:6, 3, 2)
print(A)
B <- rbind(7:9, 10:12)
print(B)
P <- A %*% B   # P es el producto matricial de A y B
print(P)
```

Consola de salida de R

```
> A <- matrix(1:6, 3, 2)
> print(A)
     [,1] [,2]
[1,]    1    4
[2,]    2    5
[3,]    3    6
> B <- rbind(7:9, 10:12)
> print(B)
     [,1] [,2] [,3]
[1,]    7    8    9
[2,]   10   11   12
> P <- A %*% B   # P es el producto matricial de A y B
> print(P)
     [,1] [,2] [,3]
[1,]   47   52   57
[2,]   64   71   78
[3,]   81   90   99
```

b) **Transpuesta de una matriz:** Se emplea la función **`t()`**.

A modo de ejemplo, si agrega al programa anterior los comandos:

```
transp_a <- t(A)

print (transp_a)
```

al ejecutarlos se obtendrá:

```
     [,1] [,2] [,3]

[1,]    1    2    3

[2,]    4    5    6
```

c) **Determinante de una matriz:** Se emplea la función **`det()`.**

Script de entrada en R

```
X <- matrix(c(1,3,-2,6), nrow = 2, ncol = 2, byrow = T)
print(X)
det(X)
```

Consola de salida de R

```
> X <- matrix(c(1,3,-2,6), nrow = 2, ncol = 2, byrow = T)
> print(X)
     [,1] [,2]
[1,]    1    3
[2,]   -2    6
> det(X)
[1] 12
```

d) **Inversa de una matriz y resolución de sistemas de ecuaciones lineales: `solve()`**

Para hallar la inversa de una matriz cuadrada A se emplea la función **`solve(A)`**.

Script de entrada en R

```
X <- matrix(c(1, 3, -2, 6), nrow = 2, ncol = 2, byrow = T)
Inversa_X <- solve(X)
print(Inversa_X)
print(X %*% Inversa_X)  # Comprobando la inversa
```

Consola de salida de R

```
> X <- matrix(c(1, 3, -2, 6),nrow = 2, ncol = 2, byrow = T)
> Inversa_X <- solve(X)
> print(Inversa_X)
          [,1]         [,2]
[1,] 0.5000000  -0.25000000
[2,] 0.1666667   0.08333333
> print(X %*% Inversa_X)  # Comprobando la inversa
     [,1] [,2]
[1,]    1    0
[2,]    0    1
```

En la solución de sistemas de ecuaciones lineales, la invocación de la función **`solve()`** tiene la forma **`x <- solve(A,b)`**, donde **`A`** es la matriz de coeficientes del sistema, **`b`** es el vector de los términos independientes y **`x`** es el vector solución. A continuación, se muestra el programa para resolver el sistema de ecuaciones:

$$\begin{cases} x_1 + 2x_2 = 7 \\ x_1 + x_2 = 4 \end{cases}$$

Script de entrada en R

```
# Resolución de un sistema de ecuaciones lineales
A <- matrix(c(1,1,2,1), 2, 2)
print(A)
b <- c(7,4)      # Términos independientes
x <- solve(A,b)  # Solución
print(x)
# Comprobando la solución
all.equal(as.vector(A %*% x), b, tolerance = 1e-16)
```

Consola de salida de R

```
> # Resolución de un sistema de ecuaciones lineales
> A <- matrix(c(1,1,2,1), 2, 2)
> print(A)
     [,1] [,2]
[1,]    1    2
[2,]    1    1
> b <- c(7,4)      # Términos independientes
> x <- solve(A,b)  # Solución
> print(x)
[1] 1 3
> # Comprobando la solución
> all.equal(as.vector(A %*% x), b, tolerance = 1e-16)
[1] TRUE
```

e) **Valores y vectores propios**

La función **eigen(A)** calcula y devuelve los valores y vectores propios de la matriz A.

Script de entrada en R

```
X <- matrix(c(1,3,-2,6), nrow = 2, ncol = 2, byrow = T)
print(X)
# Cálculo de los valores y vectores propios de X
eigen(X)
```

Consola de salida de R

```
> X <- matrix(c(1,3,-2,6), nrow = 2, ncol = 2, byrow = T)
> print(X)
     [,1] [,2]
[1,]    1    3
[2,]   -2    6
> # Cálculo de los valores y vectores propios de X
> eigen(X)
```

```
eigen() decomposition
$values
[1] 4 3

$vectors
           [,1]        [,2]
[1,] -0.7071068  -0.8320503
[2,] -0.7071068  -0.5547002
```

f) **Extraer una submatriz de una matriz**

Pueden usarse índices negativos, como en el caso de los vectores, para excluir elementos particulares, filas o columnas completas, como se muestra en el siguiente script. La matriz original queda inalterada.

Script de entrada en R

```
m4 <- matrix(10:18, nrow = 3, byrow = T)
print(m4)
m4sf2 <- m4[-2,]   # m4sf2 es m4 sin fila 2
print(m4sf2)
m4sc2 <- m4[,-2]  # m4sc2 es m4 sin columna 2
print(m4sc2)
m4sfc2 <- m4[-2,-2]   # m4sfc2 es m4 sin fila 2 ni columna 2
print(m4sfc2)
m4sf12 <- m4[-(1:2),] # m4sf12 es m4 sin las dos primeras filas
print(m4sf12)
```

Consola de salida de R

```
> m4 <- matrix(10:18, nrow = 3, byrow = T)
> print(m4)
     [,1] [,2] [,3]
[1,]   10   11   12
[2,]   13   14   15
[3,]   16   17   18
> m4sf2 <- m4[-2,]   # m4sf2 es m4 sin fila 2
> print(m4sf2)
     [,1] [,2] [,3]
[1,]   10   11   12
[2,]   16   17   18
> m4sc2 <- m4[,-2]  # m4sc2 es m4 sin columna 2
> print(m4sc2)
     [,1] [,2]
[1,]   10   12
[2,]   13   15
```

```
[3,]   16   18
> m4sfc2 <- m4[-2,-2]   # m4sfc2 es m4 sin fila 2 ni columna 2
> print(m4sfc2)
     [,1] [,2]
[1,]   10   12
[2,]   16   18
> m4sf12 <- m4[-(1:2),] # m4sf12 es m4 sin las dos primeras filas
> print(m4sf12)
[1] 16 17 18
```

La acción `m4sfc2 <- m4[-2,-2]` es equivalente a extraer los elementos de posición [1,1], [1,3], [3,1], [3,3] de la matriz m4 para formar la submatriz deseada, lo cual puede hacerse también como se muestra a continuación. Los comentarios indican las operaciones a realizar.

Script de entrada en R

```
m4 <- matrix(10:18, nrow = 3, byrow = T)
print(m4)
# Se forma una matriz de 2 columnas con los subíndices de los
# elementos a extraer de la matriz m4
i <- matrix(c(1,1,1,3,3,1,3,3), ncol = 2, byrow = T)
print(i)
# Se conforma una matriz con los elementos de m4 ubicados en
# las posiciones seleccionadas en i
NuevaMz <- m4[i]
# Se dan las dimensiones a la nueva matriz, en este ejemplo 2x2
dim(NuevaMz) <- c(2,2)
# Como se construyó por columnas, se debe trasponer
NuevaMz <- t(NuevaMz)
print(NuevaMz)
```

Consola de salida de R

```
> m4 <- matrix(10:18, nrow = 3, byrow = T)
> print(m4)
     [,1] [,2] [,3]
[1,]   10   11   12
[2,]   13   14   15
[3,]   16   17   18
> # Se forma una matriz de 2 columnas con los subíndices de los
> # elementos a extraer de la matriz m4
> i <- matrix(c(1,1,1,3,3,1,3,3), ncol = 2, byrow = T)
> print(i)
     [,1] [,2]
```

```
[1,]    1    1
[2,]    1    3
[3,]    3    1
[4,]    3    3
> # Se conforma una matriz con los elementos de m4 ubicados en
> # las posiciones seleccionadas en i
> NuevaMz <- m4[i]
> # Se dan las dimensiones a la nueva matriz, en este ejemplo 2x2
> dim(NuevaMz) <- c(2,2)
> # Como se construyó por columnas, se debe trasponer
> NuevaMz <- t(NuevaMz)
> print(NuevaMz)
     [,1] [,2]
[1,]   10   12
[2,]   16   18
```

Extraer (o seleccionar) filas, columnas o elementos de una matriz puede conllevar a obtener vectores, que no tendrán atributo dim (su valor es NULL), como en el ejemplo siguiente:

Script de entrada en R

```
P <- matrix(10:13, nrow = 2, byrow = T)  # matriz de 2x2
print(P)
# Seleccionando la primera fila de la matriz P
PFila <- P[1, ]  # Ahora PFila es un vector
print(PFila)
dim(PFila)
# Seleccionando la segunda columna de P
PCol <- P[ ,2]   # Ahora PCol es un vector
print(PCol)
dim(PCol)
```

Consola de salida de R

```
> P <- matrix(10:13, nrow = 2, byrow = T)  # matriz de 2x2
> print(P)
     [,1] [,2]
[1,]   10   11
[2,]   12   13
> # Seleccionando la primera fila de la matriz P
> PFila <- P[1, ]  # Ahora PFila es un vector
> print(PFila)
[1] 10 11
> dim(PFila)
NULL
```

```
> # Seleccionando la segunda columna de P
> PCol <- P[ ,2]   # Ahora PCol es un vector
> print(PCol)
[1] 11 13
> dim(PCol)
NULL
```

Si resulta de interés que el resultado de una selección sea una submatriz en vez de un vector, se debe agregar en la lista de índices de la selección, un tercer elemento, el parámetro **drop = FALSE.**

Script de entrada en R

```
P <- matrix(10:13, nrow = 2, byrow = T)  # matriz de 2x2
print(P)
# Seleccionando la primera fila de P y conservando su clase
PFila <- P[1, , drop = F]  # Ahora PFila es una matriz 1x2
print(PFila)
```

Consola de salida de R

```
> P <- matrix(10:13, nrow = 2, byrow = T)  # matriz de 2x2
> print(P)
     [,1] [,2]
[1,]   10   11
[2,]   12   13
> # Seleccionando la primera fila de P y conservando su clase
> PFila <- P[1, , drop = F]  # Ahora PFila es una matriz 1x2
> print(PFila)
     [,1] [,2]
[1,]   10   11
```

El procedimiento empleado para seleccionar ciertos elementos de una matriz puede emplearse también para seleccionar elementos que satisfagan cierta condición, por ejemplo, para extraer los elementos de P que son mayores que 11, debe hacerse lo siguiente (se procede igual que cuando se trabajó con los vectores):

Script de entrada en R

```
P <- matrix(10:13, nrow = 2, byrow = T)
print(P)
# Se forma el vector NewP con los elementos de P mayores que 11
NewP <- P[P>11]
print(NewP)
print(NewP[2])
```

Consola de salida de R

```
> P <- matrix(10:13, nrow = 2, byrow = T)
> print(P)
     [,1] [,2]
[1,]   10   11
[2,]   12   13
# Se forma el vector NewP con los elementos de P mayores que 11
> NewP <- P[P>11]
> print(NewP)
[1] 12 13
> print(NewP[2])
[1] 13
```

Si se quiere reemplazar ciertos elementos de una matriz por un valor dado, se procede como se muestra a continuación, donde los elementos que ocupan las posiciones 2 y 3 (considerando la matriz como un vector) son sustituidos por 0. Tener presente que cuando una matriz se considera como vector, el ordenamiento de las posiciones de sus elementos lo realiza por columnas, esto es, las posiciones 2 y 3 de la matriz de 2x2 considerada como un vector, se corresponden con los elementos (2,1) y (1,2) de la matriz, respectivamente.

Script de entrada en R

```
P <- matrix(10:13, nrow = 2, byrow = T)
print(P)
i <- c(2,3)
P[i] <- 0
print(P)
```

Consola de salida de R

```
> P <- matrix(10:13, nrow = 2, byrow = T)
> print(P)
     [,1] [,2]
[1,]   10   11
[2,]   12   13
> i <- c(2,3)
> P[i] <- 0
> print(P)
     [,1] [,2]
[1,]   10    0
[2,]    0   13
```

4.3 Arreglos

Un arreglo es una extensión natural de una matriz, ampliando sus dimensiones. También se puede decir que un arreglo es un vector que es representable y accesible a través de un cierto número de dimensiones (vector numérico de dimensiones), generalmente más de dos. Todos los elementos del arreglo son del mismo tipo. Los vectores y las matrices son casos particulares de arreglos.

4.3.1 Creación de arreglos

Los elementos del vector de dimensiones indican los límites superiores de los índices. Los límites inferiores siempre valen 1. Existen dos maneras de crear un arreglo:

a) A partir de un vector.

Un vector puede transformarse en un arreglo cuando se asigna un vector de dimensiones al atributo **dim**, al igual que se hizo con matrices. Por ejemplo, para crear un arreglo de 3 dimensiones se emplea el vector de dimensiones `c(4, 2, 3)` que está diciendo que el arreglo tendrá 4 elementos en la primera dimensión, 2 en la segunda y 3 en la tercera, o sea, 3 matrices de 4 filas y 2 columnas.

Script de entrada en R

```
# Generando 24 números aleatorios N(0,1)
set.seed(0)
z<-rnorm(24)
# Creando un arreglo de 4x2x3
dim(z)<-c(4,2,3)
z
```

Consola de salida de R

```
> Generando 24 números aleatorios N(0,1)
> set.seed(0)
> z<-rnorm(24)
> # Creando un arreglo de 4x2x3
> dim(z)<-c(4,2,3)
> z
, , 1
```

```
           [,1]       [,2]
[1,]  1.2629543  0.4146414
[2,] -0.3262334 -1.5399500
[3,]  1.3297993 -0.9285670
[4,]  1.2724293 -0.2947204

, , 2
             [,1]       [,2]
[1,] -0.005767173 -1.1476570
[2,]  2.404653389 -0.2894616
[3,]  0.763593461 -0.2992151
[4,] -0.799009249 -0.4115108

, , 3
           [,1]       [,2]
[1,]  0.2522234 -0.2242679
[2,] -0.8919211  0.3773956
[3,]  0.4356833  0.1333364
[4,] -1.2375384  0.8041895
```

Usando la función array(vector de datos, vector de dimensiones).

Por ejemplo, para crear un arreglo formado por dos matrices de 4x2 con los elementos del 1 al 8 la primera y del 9 al 16 la segunda se escribiría el comando:

```
array1 <- array(1:16, dim = c(4, 2, 2))
```

Con ello se ha creado un arreglo de dimensiones 4x2x2, o sea, dos matrices de 4x2. El primer número (4) es la cantidad de filas, el segundo (2) la cantidad de columnas y el tercero (2) el nivel de profundidad. Al ejecutar este comando y luego imprimir array1 saldrían dos matrices, según se muestra con la siguiente corrida:

Script de entrada en R

```
array1 <- array(1:16, dim = c(4, 2, 2))
print(array1)
class(array1)
```

Consola de salida de R

```
> array1 <- array(1:16, dim = c(4, 2, 2))
> print(array1)
, , 1
```

```
     [,1] [,2]
[1,]    1    5
[2,]    2    6
[3,]    3    7
[4,]    4    8

, , 2
     [,1] [,2]
[1,]    9   13
[2,]   10   14
[3,]   11   15
[4,]   12   16
> class(array1)
[1] "array"
```

En R, al igual que en otros lenguajes, los índices del arreglo siguen la regla de que el primer índice es el que se mueve primero y el último es el que se mueve último. Observar además que, al igual que con los vectores, por defecto el vector se transforma en una matriz completando por columnas.

Ejemplo: Si se define una variable indexada "vi" con vector de dimensiones c(3,4,2), la variable indexada tendrá 3x4x2=24 elementos que se formarán a partir de los elementos originales en el orden (moviéndonos por la fila). En la figura 13 se muestra esta estructura:

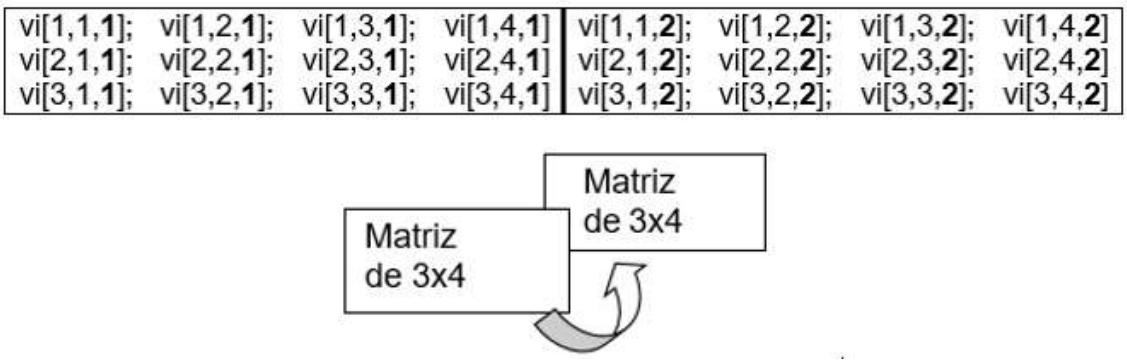


Figura 13. Estructura de un arreglo de dimensión 3x4x2.

4.3.2 ¿Cómo acceder a partes o a elementos de un arreglo?

Puede referenciarse una parte de un arreglo mediante una sucesión de índices, teniendo en cuenta que, si un índice es vacío, equivale a utilizar todo el rango de valores para dicho índice.

Script de entrada en R

```
Array2 <- array(11:26, dim = c(4, 2, 2))
print(Array2)
#Accediendo a los elementos de un arreglo
Array2[3,1,1] # Acceso al elemento de fila 3, columna 1, nivel 1
Array2[3,1,2] # Acceso al elemento de fila 3, columna 1, nivel 2
Array2[ , ,1] # Matriz de 4x2 correspondiente al nivel 1 de array1
# Matriz de 2x2 con elementos de la fila 2 de cada matriz,
# colocados por columnas
Array2[2, , ]
Array2[2]  # Segundo elemento de array1 en recorrido por columnas
Array2[11] # 11-avo elemento de array1 en recorrido por columnas
```

Consola de salida de R

```
> array2 <- array(11:26, dim = c(4, 2, 2))
> print(Array2)
, , 1
     [,1] [,2]
[1,]   11   15
[2,]   12   16
[3,]   13   17
[4,]   14   18
, , 2
     [,1] [,2]
[1,]   19   23
[2,]   20   24
[3,]   21   25
[4,]   22   26

> #Accediendo a los elementos de un arreglo
> Array2[3,1,1] # Acceso al elemento de fila 3, columna 1, nivel 1
[1] 13
> Array2[3,1,2] # Acceso al elemento de fila 3, columna 1, nivel 2
[1] 21
> Array2[ , ,1] # Matriz de 4x2 correspondiente al nivel 1 de array1
     [,1] [,2]
[1,]   11   15
[2,]   12   16
[3,]   13   17
[4,]   14   18
```

```
> # Matriz de 2x2 con elementos de la fila 2 de cada matriz,
> # colocados por columnas
> Array2[2, , ]
     [,1] [,2]
[1,]   12   20
[2,]   16   24
> Array2[2]  # Segundo elemento de array1 en recorrido por columnas
[1] 12
> Array2[11] # 11-avo elemento de array1 en recorrido por columnas
[1] 21
```

Note que **`array2[11]`** corresponde al onceavo elemento del arreglo, contado por columna y comenzando en el nivel 1, considerando este como un vector de longitud 16.

4.3.3 Dando nombre a filas, columnas y matrices en un arreglo

Se puede dar nombres a las filas, columnas y matrices de un arreglo usando el parámetro **`dimnames`**.

Script de entrada en R

```
vector <- c(2,9,6,10,15,13,16,11,12)
column.names <- c("COL1","COL2","COL3")
row.names <- c("ROW1","ROW2","ROW3")
matrix.names <- c("Matrix1","Matrix2")
result <- array(vector, dim = c(3,3,2),
          dimnames = list(row.names, column.names, matrix.names))
print(result)
```

Consola de salida de R

```
> vector <- c(2,9,6,10,15,13,16,11,12)
> column.names <- c("COL1","COL2","COL3")
> row.names <- c("ROW1","ROW2","ROW3")
> matrix.names <- c("Matrix1","Matrix2")
> result <- array(vector, dim = c(3,3,2),
+           dimnames = list(row.names, column.names, matrix.names))
> print(result)
, , Matrix1
     COL1 COL2 COL3
ROW1    2   10   16
ROW2    9   15   11
ROW3    6   13   12
, , Matrix2
     COL1 COL2 COL3
ROW1    2   10   16
```

```
ROW2    9   15   11
ROW3    6   13   12
```

Si el arreglo tiene más de 3 dimensiones se procede de forma similar.

4.4 Factores

Un factor es usado para especificar los elementos que no se repiten en un vector y pueden ser ordenados o desordenados. Se puede pensar en los factores como un vector de enteros en el que cada entero representa una etiqueta. Los factores son importantes para trabajar con modelos estadísticos, por ejemplo, el Análisis de Varianza. El lenguaje R proporciona la clase **factor** para trabajar con los factores.

Sean los vectores de caracteres empleados en el epígrafe 4.1.5:

```
nombre <- c("Juana", "Juan", "Pedro", "Luis", "Maria",
            "Gloria", "Bertha", "Rosa", "Rafael", "Ernesto", "Elsa")
mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio",
             "Julio", "Febrero", "Abril", "Mayo", "Abril", "Mayo")
```

Estos vectores (nombre y mes.nac) pueden considerarse como una estructura de información, la cual se puede someter a algún tipo de procesamiento estadístico.

4.4.1 Creación de un factor

Los objetos de tipo "factor" pueden ser creados con la función **`factor()`**.

Script de entrada en R

```
nombre <- c("Juana", "Juan", "Pedro", "Luis", "Maria", "Gloria",
            "Bertha", "Rosa", "Rafael", "Ernesto", "Elsa")
mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio",
             "Julio", "Febrero", "Abril", "Mayo", "Abril", "Mayo")
Fmes.nac <- factor(mes.nac) # Creación del factor
print(Fmes.nac)
```

Consola de salida de R

```
> nombre <- c("Juana", "Juan", "Pedro", "Luis", "Maria", "Gloria",
+             "Bertha", "Rosa", "Rafael", "Ernesto", "Elsa")
```

```
> mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio",
+              "Julio", "Febrero", "Abril", "Mayo", "Abril", "Mayo")
> Fmes.nac <- factor(mes.nac) # Creación del factor
> print(Fmes.nac)
 [1] Enero   Febrero Abril   Febrero Julio   Julio   Febrero Abril   Mayo
[10] Abril   Mayo
Levels: Abril Enero Febrero Julio Mayo
```

Niveles o categorías (**levels**) son los valores diferentes presentes en el factor. En el ejemplo anterior los niveles son: `Abril, Enero, Febrero, Julio y Mayo.` Aquí aparecen todos los niveles en orden alfabético, si se requieren solo algunos niveles o establecer un orden de los niveles de un factor se puede usar el argumento **`levels`** de la función **`factor()`**. Si la penúltima línea del programa anterior la sustituimos por:

```
Fmes.nac <- factor(mes.nac, levels = c("Abril", "Julio"))
```

sale entonces:

```
[1] <NA>  <NA>  Abril <NA>  Julio Julio <NA>  Abril <NA>  Abril <NA>
Levels: Abril Julio
```

La clase *factor* asigna un valor entero a cada uno de los elementos que no se repiten, tomados estos alfabéticamente como se muestra a continuación:

Elementos del vector sin repetición:	Abril	Enero	Febrero	Julio	Mayo
Número que le asigna:	1	2	3	4	5

Por ello al vector original (VO) le hace corresponder el vector codificado (VCod):

VO	Enero	Febrero	Abril	Febrero	Julio	Julio	Febrero	Abril	Mayo	Abril	Mayo
VCod	2	3	1	3	4	4	3	1	5	1	5

Esta estructura interna de la clase se puede descubrir con la función **`unclass()`**. Agregando `unclass(Fmes.nac)` al programa antes desarrollado, se agregaría la salida:

```
> unclass(Fmes.nac)
```

```
 [1] 2 3 1 3 4 4 3 1 5 1 5

attr(,"levels")

[1] "Abril"   "Enero"   "Febrero"   "Julio"   "Mayo"
```

Para obtener la frecuencia de cada factor se usa la función **table()**[10], la cual toma típicamente como argumento un factor y regresa como resultado la frecuencia de aparición de los niveles, como se muestra en el siguiente script:

Script de entrada en R

```
mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio",
             "Julio", "Febrero", "Abril", "Mayo", "Abril", "Mayo")
Fmes.nac <- factor(mes.nac)
table(Fmes.nac)
```

Consola de salida de R

```
> mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio",
+              "Julio", "Febrero", "Abril", "Mayo", "Abril", "Mayo")
> Fmes.nac <- factor(mes.nac)
> table(Fmes.nac)
Fmes.nac
  Abril   Enero   Febrero   Julio   Mayo
      3       1         3       2      2
```

La interpretación de estos resultados en el contexto de la estructura de información original, es que, por ejemplo, tres personas del vector *nombre*, nacieron en el mes de Abril, una en Enero, etc.

4.4.2 Acceso a los elementos de un factor

La estructura de un factor está compuesta por dos vectores, observe en la tabla 7 cómo se accede a/o se modifica un elemento de ese factor. En los ejemplos se usa el factor Fmes.nac.

[10] Devuelve un objeto de la clase **table**, que no es más que un arreglo de valores enteros.

Tabla 7. Acceso o modificación de los elementos de un factor

Operación	**Forma general**	**Ejemplo**
Acceso a un elemento particular del factor	`nombre_factor[k]`, $1 \leq k \leq m$ con m: cantidad de elementos del factor	`Fmes.nac[5]`
Acceso a un nivel individual	`levels(nombre_factor)[l]` $1 \leq l \leq n$ n: cantidad de niveles del factor	`levels(Fmes.nac)[3]`
Modificar un nivel	`levels(nombre_factor)[l] <- nuevo_nivel`	`levels(Fmes.nac)[3] <- "Junio"`
Mostrar factor como vector de índices	`as.integer(nombre_factor)`	`as.integer(Fmes.nac)`

Todo esto se muestra en el siguiente script:

Script de entrada en R

```
mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio",
  "Julio", "Febrero", "Abril", "Mayo", "Abril", "Mayo")
Fmes.nac <- factor(mes.nac) # Creación del factor
print(Fmes.nac)
# Acceso a un elemento individual del factor
Fmes.nac[5]
# Un elemento individual de los niveles
levels(Fmes.nac)[3]
# Cambiando nivel Febrero por Junio en Fmes.Nac
levels(Fmes.nac)[3] <- "Junio"
Fmes.nac
# Mostrando el factor como un vector de índices
as.integer(Fmes.nac)
```

Consola de salida de R

```
> mes.nac <- c("Enero", "Febrero", "Abril", "Febrero", "Julio",
+   "Julio", "Febrero", "Abril", "Mayo", "Abril", "Mayo")
> Fmes.nac <- factor(mes.nac) # Creación del factor
> print(Fmes.nac)
 [1] Enero   Febrero Abril   Febrero Julio   Julio   Febrero Abril
 [9] Mayo    Abril   Mayo
Levels: Abril Enero Febrero Julio Mayo
> # Acceso a un elemento individual del factor
> Fmes.nac[5]
[1] Julio
```

```
Levels: Abril Enero Febrero Julio Mayo
> # Un elemento individual de los niveles
> levels(Fmes.nac)[3]
[1] "Febrero"
> # Cambiando nivel Febrero por Junio en Fmes.Nac
> levels(Fmes.nac)[3] <- "Junio"
> Fmes.nac
 [1] Enero Junio Abril Junio Julio Julio Junio Abril Mayo  Abril Mayo
Levels: Abril Enero Junio Julio Mayo
> # Acceso al factor como un vector de índices
> as.integer(Fmes.nac)
 [1] 2 3 1 3 4 4 3 1 5 1 5
```

¿Cómo generar un factor?

Script de entrada en R

```
# Factor con 3 categorías
f <- as.factor(c(1,2,3,1,2,1,1,3,2))
f
# Asignando nombres a las categorías
levels(f) <- c("Bajo",  "Medio",  "Alto")
f
# Ordenando los factores
f_Ord <- as.ordered(f)
f_Ord
```

Consola de salida de R

```
# Factor con 3 categorías
> f <- as.factor(c(1,2,3,1,2,1,1,3,2))
> f
[1] 1 2 3 1 2 1 1 3 2
Levels: 1 2 3
# Asignando nombres a las categorías
> levels(f)<-c("Bajo", "Medio", "Alto")
> f
[1] Bajo Medio Alto Bajo Medio Bajo Bajo Alto Medio
Levels: Bajo Medio Alto
# Ordenando los factores
> f_Ord<-as.ordered(f)
> f_Ord
[1] Bajo Medio Alto Bajo Medio Bajo Bajo Alto Medio
Levels: Bajo < Medio < Alto
```

Ejercicios

1. Probar las propiedades de la concatenación para vectores.
2. Escriba un programa en R para crear e imprimir un vector con 10 enteros aleatorios entre -50 y 50.
3. Crear una secuencia llamada `seq_1`, con valores entre -10 y 10, con separación entre valores igual a 2.
4. Defina otra secuencia llamada `seq_2,` que contenga 1000 valores equidistantes, entre 0 y 100.
5. Calcule la suma de `seq_1` y `seq_2` obtenidas en los ejercicios 3 y 4. ¿Es válida esta operación a pesar de que ambas secuencias tienen diferentes longitudes? Explique.
6. Escriba un script que genere una secuencia de cubos y cuadrados de los números del 20 al 40.
7. Escriba un script que cree un vector de senos de ángulos entre 0º y 90º con paso 5º. Debe recordar que la función sin() procesa ángulos dados en radianes (360° = 2π radianes).
8. Escriba un programa en R para crear una matriz en cuya diagonal principal aparecerán los elementos de un vector dado, por ejemplo, (4,8,3,7) y el resto de los elementos de la matriz son ceros.
9. Escriba un programa en R para crear una matriz numérica de 3x4 e imprima el elemento de fila 2 y columna 3, la fila 3 completa y la columna 4 completa.
10. Escriba un programa en R para crear tres vectores a, b y c, cada uno con 3 valores enteros. Combine los tres vectores para formar una matriz de 3x3, donde cada columna representa un vector. Imprima el contenido de la matriz.

11. A partir de los vectores siguientes conforme una matriz llamada M, cuyas filas sean dichos vectores:

    f1 = (5, 2, 5); f2 = (3, 4, 2) ; f3 = (3, 6, 15)

12. Para la matriz obtenida en el ejercicio 11, halle:

    a) Su transpuesta.

    b) Su determinante.

    c) Sus dimensiones.

13. Genere 9 números aleatorios con distribución Poisson de parametro λ=2 y conforme con ellos una matriz de 3x3, llamada Pois. Póngale nombres a las filas y a las columnas.

    - Con las matrices M (del ejercicio 11) y Pois, obtenga su suma.
    - De la matriz Pois obtenga una submatriz conformada por los elementos de las 2 primeras filas y las dos primeras columnas.
    - Halle el producto matricial entre M y Pois.

14. La siguiente secuencia de valores contiene información referente al peso de 47 estudiantes. Se desea determinar de estos, cuales podrán participar en una tabla gimnastica que se está conformando. Para poder integrar la tabla se exige que el peso sea como máximo igual a 75 libras. Si los estudiantes están codificados del 1 al 47, diga qué estudiantes podrán integrar dicha secuencia.

75	54	96	75	75	70	83	77	52	65	78	67
72	64	56	78	82	80	70	95	75	71	67	87
60	79	75	50	56	80	53	60	80	66	90	70
56	74	100	85	49	72	90	58	92	58	82	

15. A partir de los datos que se muestran en la tabla siguiente, que corresponden al rendimiento de 3 variedades de caña A, B y C, ubique estos datos en un vector nombrado

Rend y genere un factor (Fact) que especifique a qué variedad pertenece cada dato, según su posición en el vector Rend. Halle la media de cada una de las variedades y su varianza.

A	**B**	**C**
20.4	10.1	15.3
33.6	12.9	15.4
28.0	15.8	12.8
33.1	18.0	12.6
34.0	20.4	15.2
29.2		

16. La rotación en dos dimensiones es una transformación lineal: si se quiere rotar el punto (x_1, x_2) por un ángulo α, la operación está dada por:

$$\begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} \cos\alpha & -\operatorname{sen}\alpha \\ \operatorname{sen}\alpha & \cos\alpha \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \end{pmatrix}$$

Supóngase ahora que se tiene un triángulo cuyos vértices son (1, 0); (2, 1) y (1, 1) y se quieren encontrar los vértices del triángulo resultante de una rotación de 32º. Calcule estos nuevos vértices.

17. Investigue en qué consiste la descomposición en valores singulares de una matriz y su relación con el determinante y el rango de la matriz.
18. A través de la ayuda del R, determine cómo se halla la descomposición en valores singulares de una matriz.
19. Dada la matriz $M = \begin{pmatrix} 2 & 3 & 2 \\ 1 & 10 & 5 \\ 3 & 4 & 2 \end{pmatrix}$, halle su descomposición en valores singulares y calcule su rango y su determinante.
20. Calcule los valores y vectores propios de la matriz M del ejercicio 19.
21. Calcule los valores singulares de M a partir de los valores propios de M^tM y compruebe que coinciden con los obtenidos en el ejercicio 20.

Unidad 5. Estructuras de datos heterogéneas: listas y data frames

5.1 Listas

Una lista (objeto de la clase **list**) es un objeto de datos que puede contener cero o más elementos, cada uno de los cuales puede pertenecer a una clase distinta, incluso otras listas.

5.1.1 Creación de una lista

Para crear una lista se emplea la función **`list()`**. Por ejemplo, para crear una lista de un vector numérico de un solo elemento, de un vector lógico de dos valores y de otro vector de 3 caracteres se escribiría `list1 <- list(1, c(TRUE, FALSE), c("a", "b", "c"))`.

También puede crearse una lista vacía con una longitud preestablecida mediante la función `vector()`: `x <- vector("list", length = 5)`.

A continuación, se propone un script para crear una lista para representar los integrantes de un grupo, tomando de cada uno de ellos su *nombre*, *sexo*, si es *becado* o no y sus *notas* en Probabilidades:

Script de entrada en R

```
# Creación de una lista
estudiante <-c ("Juan","María","Pedro")
sexo <- c("M","F","M")
becado <- c("No","Si","Si")
notas <- c(3,2,3)
grupo <- list(estudiante,sexo,becado,notas)
grupo
```

Consola de salida de R

```
> # Creación de una lista
> estudiante <- c("Juan","María","Pedro")
> sexo <- c("M","F","M")
> becado <- c("No","Si","Si")
> notas <- c(3,2,3)
> grupo <- list(estudiante,sexo,becado,notas)
> grupo
[[1]]
[1] "Juan" "María" "Pedro"
```

```
[[2]]
[1] "M" "F" "M"

[[3]]
[1] "No" "Si" "Si"

[[4]]
[1] 3 2 3
```

Como se puede observar esta lista está formada por 4 elementos, todos de la misma longitud, 3. ¿Podría una lista tener elementos de longitudes diferentes? Perfectamente. El ejemplo siguiente muestra la integración de una familia: *nombre de la madre*, *nombre del padre*, *años de casados*, *nombres de los hijos y edad de los hijos.*

Script de entrada en R

```
familia <- list("Juana","Manuel",10,
                c("Juanita","Luisito"), c(8, 6))
familia
```

Consola de salida de R

```
> familia <- list("Juana","Manuel",10,
+                 c("Juanita","Luisito"), c(8, 6))
> familia
[[1]]
[1] "Juana"

[[2]]
[1] "Manuel"

[[3]]
[1] 10

[[4]]
[1] "Juanita" "Luisito"

[[5]]
[1] 8 6
```

Esta lista tiene 5 elementos, los tres primeros de longitud 1 y los otros dos de longitud 2. Observar que todos los elementos de la lista no son de la misma clase, los dos primeros son de

la clase **character** de longitud 1, el tercero es **numeric**, el cuarto es **character** de longitud 2 y el último es **numeric** de longitud 2.

Al igual que en el caso de los vectores, los elementos de las listas pueden ser nombrados, lo que añade mayor claridad a su significado dentro de la lista. La forma de hacerlo es agregando el nombre deseado y el símbolo **=** delante del elemento de lista a nombrar, como se muestra a continuación:

Script de entrada en R

```
familia <- list(Madre="Juana", Padre="Manuel", Casados=10,
           Hijos=c("Juanita","Luisito"), Edades=c(8,6))
familia
```

Consola de salida de R

```
> familia <- list(Madre="Juana", Padre="Manuel", Casados=10,
+            Hijos=c("Juanita","Luisito"), Edades=c(8,6))
> familia
$`Madre`
[1] "Juana"

$Padre
[1] "Manuel"

$Casados
[1] 10

$Hijos
[1] "Juanita" "Luisito"

$Edades
[1] 8 6
```

5.1.2 Acceso a los elementos de una lista

Existen varias formas de acceder a los elementos de una lista, usando los operadores **[]** (corchete simple), **[[]]** (doble corchete) y **$**.

Si, formalmente, una lista x con n elementos es denotada como $list(x_1, x_2, \ldots, x_n) \equiv \langle x_1, x_2, \ldots, x_n \rangle$, se tiene que:

a) Los elementos de la lista están indexados por los valores 1, 2, ..., n.

b) Operador [[ ]]: Para una lista x, $x[[k]] \equiv x_k$, o sea, el k-ésimo elemento de la lista.

c) Operador $: Es equivalente a la operación [[ ]].

d) Operación [ ]: Para una lista x, $x[k] \equiv \langle x_k \rangle$, o sea, la lista que contiene como único elemento a x_k. Esta operación permite obtener una sublista.

e) La longitud de la lista es *n*.

Algunos ejemplos que permitirán precisar en la práctica diferencias entre esos operadores se muestran en el siguiente script:

Script de entrada en R

```
familia <- list(Madre="Juana", Padre="Manuel", Casados=10,
                Hijos=c("Juanita","Luisito"), Edades=c(8,6))
# Determinación de cada elemento de la lista familia
familia$Madre     # madre de la familia
familia$Padre     # padre de la familia
familia$Casados   # años de casados
familia$Hijos     # nombres de los hijos
familia$Edades    # edades de los hijos
# Determinación de cada elemento de la lista familia
# Incluso integrantes de cada elemento
familia[["Madre"]]  # madre de la familia
familia[["Hijos"]]  # hijos de la familia
familia[["Hijos"]][1]  # primer hijo de la familia
# Lista formada por los elementos que se especifican de familia
familia[c("Madre","Padre")]
```

Consola de salida de R

```
> familia <- list(Madre="Juana", Padre="Manuel", Casados=10,
+                 Hijos=c("Juanita","Luisito"), Edades=c(8,6))
> # Determinación de cada elemento de la lista familia
> familia$Madre     # madre de la familia
[1] "Juana"
> familia$Padre     # padre de la familia
[1] "Manuel"
> familia$Casados   # años de casados
[1] 10
> familia$Hijos     # nombres de los hijos
[1] "Juanita" "Luisito"
```

```
> familia$Edades    # edades de los hijos
[1] 8 6
> # Determinación de cada elemento de la lista familia
> # Incluso integrantes de cada elemento
> familia[["Madre"]]  # madre de la familia
[1] "Juana"
> familia[["Hijos"]]  # hijos de la familia
[1] "Juanita" "Luisito"
> familia[["Hijos"]][1]  # primer hijo de la familia
[1] "Juanita"
> # Lista formada por los elementos que se especifican de familia
> familia[c("Madre","Padre")]
$Madre
[1] "Juana"

$Padre
[1] "Manuel"
```

Observe el uso de nombres entre comillas con los operadores `[]` y `[[]],` en el script anterior.

Las listas tienen dos usos importantes:

a) Las funciones solo retornan un valor simple, por lo que es común retornar datos más estructurados empleando una lista. Sea:

```
f <- function(x) list(x_mas10 = x + 10, x_cuad = x^2)
```

Con la llamada **`f(7)`** sale:

```
$x_mas10

[1] 17

$x_cuad

[1] 49
```

b) Las listas constituyen un tipo fundamental para los data frames. Puede considerarse un data frame como una lista de vectores, todos con igual longitud, lo que se comprueba en el siguiente epígrafe.

c) Ayudan a personalizar nombres de filas y columnas de una matriz o data frame.

Se desea crear una matriz de valores aleatorios de 4 filas y 5 columnas. Las filas se numeran del 1 al 4 y las columnas con los nombres C1 a C5 y cada columna separada por un espacio. Esto se logra con:

```
miMA <- matrix(rnorm(20),4,5,
               dimnames=list(1:4,paste("C",1:5,sep="")))
```

Al ejecutar `print(miMA)` una salida puede ser:

```
           C1         C2         C3         C4          C5
1 -1.27610308  0.3414477 -0.9556249  0.6511444 -0.02741414
2 -0.18238174  0.7270017  1.4095437  1.5226482 -0.24127984
3  0.29870378  1.2522645 -0.4450350  0.1250311 -2.32112778
4 -0.01725328 -0.1739350 -0.8649551 -0.8065176  1.47892948
```

d) Un vector es un caso particular de una lista donde todos los elementos son del mismo tipo. De hecho existe la función **`unlist()`** que permite convertir una lista en un vector. Solo se considera aquí un ejemplo en que todos los elementos de la lista son del mismo tipo atómico, pues la función tiene diversos parámetros e interpretaciones.

Script de entrada en R

```
# Creando una lista
list1 <- list(2:6)
print(list1)
# Convirtiendo la lista en vector
v1 <- unlist(list1)
print(v1)
class(v1)
```

Consola de salida de R

```
> # Creando una lista
> list1 <- list(2:6)
> print(list1)
[[1]]
[1] 2 3 4 5 6
> # Convirtiendo la lista en vector
```

```
> v1 <- unlist(list1)
> print(v1)
[1] 2 3 4 5 6
> class(v1)
[1] "integer"
```

Pueden insertarse y eliminarse (o modificarse) elementos de una lista mediante los operadores de acceso a sus elementos. A modo de ejemplo, observe la salida en consola de la ejecución de un script que contempla estas operaciones:

```
> z <- list(a="abc", b=12)

> z

$a

[1] "abc"

$b

[1] 12

> # Insertar nuevo elemento (1) en la posición de nombre c

> z$c <- 1

> # Lo anterior es equivalente a z[3] <- 1

> z

$a

[1] "abc"

$b

[1] 12

$c

[1] 1

> # Para eliminar el primer elemento
```

```
> z[1] <- NULL
> z
$b
[1] 12
$c
[1] 1
> # Cambiando valor del elemento de posición 2 en la actual z
> z[[2]] <- 8
> z
$b
[1] 12
$c
[1] 8
```

5.2 Tabla de datos o data frame

Los ***data frames*** son utilizados en R para almacenar datos de tipo tabla. Es un tipo de objeto muy importante en R y utilizados ampliamente en modelado estadístico.

Un data frame puede considerarse como una lista de vectores de igual longitud, en que cada vector (columna del data frame) puede corresponder a una variable en un experimento y cada fila a una observación simple o unidad experimental. El modo de un data frame es *list*.

Además del nombre de las columnas, que indica el nombre de las variables, los data frames tienen un atributo especial denominado **`row.names`** que indica la información de cada fila en el data frame.

5.2.1 Creación de un data frame

Los data frames pueden ser creados explícitamente con la función **`data.frame()`**. Por ejemplo, para crear un data frame con datos de personas que recoge nombre, estatura, peso y sexo situados en diferentes vectores:

```
nombre <- c("Alberto","Bárbara","Carlos","Juana","Elena")
estatura <- c(172,164,173,165,166)
peso <- c(95,67,75,62,55)
sexo <- c("M","F","M","F","F")
```

basta escribir:

```
d <- data.frame(nombre,sexo,estatura,peso)
```

y se forma el data frame:

```
   nombre sexo estatura peso
1 Alberto    M      172   95
2 Barbara    F      164   67
3  Carlos    M      173   75
4   Juana    F      165   62
5   Elena    F      166   55
```

Este data frame tiene 5 filas, una para cada persona y 4 columnas, una por cada característica que se ha querido recoger.

El acceso a un elemento o partes de un data frame es muy parecido a como se hace con listas y matrices. Por ejemplo, el acceso a la estatura de Carlos (la tercera persona), se hace usando el operador **`[]`** de matrices: **`d[3,3]`**. Incluso se puede modificar un valor particular en el data frame: **`d[3,3] <- 188`**.

Una columna en particular se puede acceder con el operador **$**:

```
nombre_dataframe$nombre_columna
```

Por ello **d$estatura** es un vector con los valores de la columna de estatura del data frame **d**, similar a **d[[3]]**. En cambio **d[3]** es un data frame con una sola columna, que es la columna 3 (estatura) del data frame **d**.

Otra característica importante de los data frames es que, las columnas de tipo **character** se pueden convertir a tipo factor, con el parámetro `stringsAsFactors`, el cual por defecto, a partir de la versión 4.0.0 de R, toma el valor FALSE.

Si se desea seleccionar elementos de la tabla que cumplen cierta condición, por ejemplo, calcular el promedio de la estatura de las mujeres, escribimos:

```
mean(d$estatura[d$sexo == "F"])
```

La cantidad de filas y columnas de un data frame x se obtiene con las funciones **nrow(x)** y **ncol(x)**, respectivamente.

A continuación, se muestran todos los elementos estudiados en un solo script:

Script de entrada en R

```
# Formación de un data frame
nombre <- c("Alberto","Bárbara","Carlos","Juana","Elena")
estatura <- c(172,164,173,165,166)
peso <- c(95,67,75,62,55)
sexo <- c("M","F","M","F","F")
d <- data.frame(nombre, sexo, estatura, peso)
print(d)
# estatura de la tercera persona (Carlos)
d[3,3]
# Cambiando estatura de la tercera persona
d[3,3] <- 188
# Uso del símbolo $ para identificar una columna del data frame
d$estatura
# Otra forma de obtener el mismo resultado
d[[3]]
class(d[[3]])
# Data frame formado por la tercera columna
```

```
d[3]
class(d[3])
# Cálculo de promedio de estatura
mean(d$estatura)
# Promedio de estatura de las mujeres
mean(d$estatura[d$sexo == "F"])
# Cantidad de filas del data frame
nrow(d)
# Cantidad de columnas
ncol(d)
# Se añade columna al data frame usando cbind().
# Se crea el vector con los valores
aptitud <- c(35,20,32,22,18)
# Se añade el vector como una columna
d <- cbind(d, aptitud)
# Es lo mismo: d <- cbind(d, aptitud = c(35,20,32,22,18))
print(d)
# Se añade una fila al data frame usando rbind()
d <- rbind(d, data.frame(nombre = "Jose", sexo = "M",
     estatura = 175, peso = 77, aptitud = 20))
print(d)
# Conversión automática de columnas character a factores
x<-d[["nombre"]]
class(x)
print(x)
d2 <- data.frame(nombre, sexo, estatura, peso,
      stringsAsFactors=TRUE)
print(d2)
x<-d2[["nombre"]]
class(x)
print(x)
```

Consola de salida de R

```
> # Formación de un data frame
> nombre <- c("Alberto","Bárbara","Carlos","Juana","Elena")
> estatura <- c(172,164,173,165,166)
> peso <- c(95,67,75,62,55)
> sexo <- c("M","F","M","F","F")
> d <- data.frame(nombre, sexo, estatura, peso)
> print(d)
   nombre  sexo  estatura  peso
1 Alberto     M       172    95
2 Bárbara     F       164    67
3  Carlos     M       173    75
4   Juana     F       165    62
5   Elena     F       166    55
```

```
> # estatura de la tercera persona (Carlos)
> d[3,3]
[1] 173
> # Cambiando estatura de la tercera persona
> d[3,3] <- 188
> # Uso del símbolo $ para identificar columna
> d$estatura
[1] 172 164 188 165 166
> # Lo mismo
> d[[3]]
[1] 172 164 188 165 166
> class(d[[3]])
[1] "numeric"
> # Data frame formado por la tercera columna
> d[3]
  estatura
1      172
2      164
3      188
4      165
5      166
> class(d[3])
[1] "data.frame"
> # Cálculo de promedio de estatura
> mean(d$estatura)
[1] 171
> # Promedio de estatura de las mujeres
> mean(d$estatura[d$sexo == "F"])
[1] 165
> # Cantidad de filas en el data frame
> nrow(d)
[1] 5
> # Cantidad de columnas
> ncol(d)
[1] 4
> # Se añade columna al data frame usando cbind().
> # Se crea el vector con los valores
> aptitud <- c(35,20,32,22,18)
> # Se añade el vector como una columna
> d <- cbind(d, aptitud)
> # Es lo mismo: d <- cbind(d, aptitud = c(35,20,32,22,18))
> print(d)
   nombre sexo estatura peso aptitud
1 Alberto    M      172   95      35
2 Bárbara    F      164   67      20
3  Carlos    M      188   75      32
```

```
4   Juana    F      165   62      22
5   Elena    F      166   55      18
> # Se añade una fila al data frame usando rbind()
> d <- rbind(d, data.frame(nombre = "Jose", sexo = "M",
+      estatura = 175, peso = 77, aptitud = 20))
> print(d)
   nombre sexo estatura peso aptitud
1 Alberto    M      172   95      35
2 Bárbara    F      164   67      20
3  Carlos    M      188   75      32
4   Juana    F      165   62      22
5   Elena    F      166   55      18
6    Jose    M      175   77      20
> # Conversión automática de columnas character a factores
> x<-d[["nombre"]]
> class(x)
[1] "character"
> print(x)
[1] "Alberto" "Bárbara" "Carlos"  "Juana"   "Elena"   "Jose"
> d2 <- data.frame(nombre, sexo, estatura, peso,
+       stringsAsFactors=TRUE)
> print(d2)
   nombre sexo estatura peso
1 Alberto    M      172   95
2 Bárbara    F      164   67
3  Carlos    M      173   75
4   Juana    F      165   62
5   Elena    F      166   55
> x<-d2[["nombre"]]
> class(x)
[1] "factor"
> print(x)
[1] Alberto Bárbara Carlos  Juana   Elena
Levels: Alberto Bárbara Carlos Elena Juana
```

Note que **`rbind()`** y **`cbind()`** no modifican los datos originales, sino que crean un nuevo data frame con las filas o columnas agregadas.

La primera fila y la primera columna no son parte de los datos de la tabla; ellos son, respectivamente, los nombres de las columnas y filas de la tabla o data frame, lo que se puede constatar mediante las funciones **`colnames()`** y **`rownames()`** aplicadas a la última versión del data frame ***d*** antes estudiando:

```
> rownames(d)
[1] "1" "2" "3" "4" "5" "6"
> colnames(d)
[1] "nombre"   "sexo"     "estatura" "peso"     "aptitud"
```

La coerción de un objeto a data frame se puede realizar con la función **`as.data.frame()`**. Por ejemplo, considere la matriz de letras de la "a" a la "i" obtenida con:

```
> m <- matrix(letters[1:9], nrow = 3)
> m
     [,1] [,2] [,3]
[1,] "a"  "d"  "g"
[2,] "b"  "e"  "h"
[3,] "c"  "f"  "i"
```

Al ejecutar:

```
as.data.frame(m)
```

Se obtiene:

```
  V1 V2 V3
1  a  d  g
2  b  e  h
3  c  f  i
```

Los data frames normalmente son creados leyendo desde un fichero de datos externo con las funciones `read.table()`, `read.csv()` u otros formatos, lo cual se estudiará en la unidad 6.

Para obtener un resumen de un data frame se puede emplear la función **`summary()`**:

```
summary(objeto, maxsum = 7, ...)
```

donde:

- `objeto`: dataframe que se desea resumir.
- `maxsum`: valor entero que indica cuantos niveles de factores se mostrarán (por defecto `maxsum=1`).

Para las columnas con datos numéricos, `summary()` proporciona los cálculos de mínimo, primer cuartil, mediana, media, tercer cuartil y máximo.

A modo de ejemplo se muestra el resumen del dataframe **d** antes presentado.

Script de entrada en R

```
Nombre <- c("Alberto","Bárbara","Carlos","Juana","Elena")
Estatura <- c(172,164,173,165,166)
Peso <- c(95,67,75,62,55)
Sexo <- c("M","F","M","F","F")
d <- data.frame(Nombre,Sexo,Estatura,Peso)
summary(d)
```

Consola de salida de R

```
> nombre <- c("Alberto","Bárbara","Carlos","Juana","Elena")
> estatura <- c(172,164,173,165,166)
> peso <- c(95,67,75,62,55)
> sexo <- c("M","F","M","F","F")
> d <- data.frame(Nombre,Sexo,Estatura,Peso)
>
> summary(d)
    Nombre              Sexo              Estatura        Peso
 Length:5           Length:5           Min.   :164   Min.   :55.0
 Class :character   Class :character   1st Qu.:165   1st Qu.:62.0
 Mode  :character   Mode  :character   Median :166   Median :67.0
                                       Mean   :168   Mean   :70.8
                                       3rd Qu.:172   3rd Qu.:75.0
                                       Max.   :173   Max.   :95.0
```

5.2.2 Funciones auxiliares con data frames

Algunas funciones que pueden auxiliar el trabajo con data frames son: **subset()**, **transform()**, **aggregate()**, **with()** y **within()**.

La función **subset()** permite obtener un fragmento del data frame. Por ejemplo, se puede seleccionar del data frame **d** original (con columnas: Nombre, Estatura, Peso y Sexo) el Nombre, Sexo y Estatura de las personas que pesan más de 70 kg, escribiendo:

```
subset(d, Peso>70, select = c(Nombre, Sexo, Estatura))
```

obteniéndose al ejecutar:

```
   Nombre  Sexo  Estatura
1 Alberto     M       172
3  Carlos     M       173
```

El mismo resultado se obtiene usando el operador [ ] con el siguiente código:

```
d[d$Peso>70, c("Nombre", "Sexo", "Estatura")]
```

La función `subset()` puede ser aplicada a otras clases.

La función **`transform()`** permite modificar/añadir columnas a un data frame. Por ejemplo, si se desea expresar el peso de las personas en libras en **d**, puede obtenerse con:

```
d <- transform(d, Peso = Peso*2.24)
```

Al ejecutar esta instrucción y mostrar el resultado de la transformación, se obtiene:

```
> d <- transform(d, Peso = Peso*2.24)
> d
   Nombre  Sexo  Estatura     Peso
1 Alberto     M       172   212.80
2 Bárbara     F       164   150.08
3  Carlos     M       173   168.00
4   Juana     F       165   138.88
5   Elena     F       166   123.20
```

Si se desea añadir una nueva columna llamada **Edad**, se procede como sigue:

```
d <- transform(d, Edad = c(25,31,27,33,35))
```

Al ejecutar esta instrucción y mostrar el valor nuevo de **d**, se obtiene:

```
> d <- transform(d, Edad = c(25,31,27,33,35))
> d
   Nombre  Sexo  Estatura     Peso   Edad
1 Alberto     M       172   212.80     25
2 Bárbara     F       164   150.08     31
```

```
3  Carlos     M       173   168.00     27

4   Juana     F       165   138.88     33

5   Elena     F       166   123.20     35
```

La función **aggregate(x, by, FUN, ...)** permite resumir toda o parte de la información contenida en un data frame (en general un objeto **x**) atendiendo a los niveles del o los factores especificados por **by** y aplicando a estos datos la función dada en **FUN**.

Por ejemplo, se desea obtener un nuevo data frame que proporcione el promedio del peso y la talla de las 5 personas cuyos datos aparecen en la tabla.

Script de entrada en R

```
d <- data.frame(Estatura=c(172,164,192,160),Peso=c(95,67,75,62),
                Sexo=c("M","F","M","F"))
d
aggregate(d[,-3], by=list(Sexo=d[,3]), FUN=mean)
```

Consola de salida de R

```
> d <- data.frame(Estatura=c(172,164,192,160),Peso=c(95,67,75,62),
+                 Sexo=c("M","F","M","F"))
> d
  Estatura   Peso   Sexo
1      172     95     M
2      164     67     F
3      192     75     M
4      160     62     F
> aggregate(d[,-3], by=list(Sexo=d[,3]), FUN=mean)
  Sexo  Estatura   Peso
1    F       162   64.5
2    M       182   85.0
```

Se observa que los datos en el data frame se resumen por el sexo y se aplica la media a las observaciones de peso y estatura por sexo. Recordar que la instrucción X[, -3] permite mostrar todas las columnas de X excepto la tercera.

Las funciones **with()** y **within()** evalúan expresiones sobre un data frame (o lista), con una sintaxis limpia que permite prescindir del uso de algunos **$** o **[]**.

Supongamos que otra vez **d** es el data frame:

```
   Nombre Sexo Estatura Peso
1 Alberto    M      172   95
2 Bárbara    F      164   67
3  Carlos    M      173   75
4   Juana    F      165   62
5   Elena    F      166   55
```

Para crear un nuevo data frame (**nueva_d**) modificando la estatura a metros y agregando una nueva columna con el valor del índice de masa corporal (IMC), basta escribir el script:

```
nueva_d <- within(d, {
 Estatura <- Estatura/100 # Modifica estatura en metros
 IMC <- Peso/Estatura^2   # Agrega columna con índice de masa
corporal})
nueva_d
```

La salida en la consola sería:

```
   Nombre   Sexo   Estatura   Peso        IMC
1 Alberto      M       1.72     95   32.11195
2 Bárbara      F       1.64     67   24.91077
3  Carlos      M       1.73     75   25.05931
4   Juana      F       1.65     62   22.77319
5   Elena      F       1.66     55   19.95936
```

La función **within()** retorna el objeto modificado (en el ejemplo anterior el data frame) y **with()** retorna el valor de la expresión evaluada. En el siguiente ejemplo, se seleccionan los

nombres de las personas del data frame que son del sexo femenino y cuya estatura sea mayor que 150 cm.

```
nueva_d <- with(d, Nombre[Sexo == "F"  & Estatura > 150])
nueva_d
```

Al ejecutar las instrucciones anteriores, se obtendrá la siguiente salida:

```
> nueva_d <- with(d, Nombre[Sexo == "F"  & Estatura > 150])
> nueva_d
[1] "Bárbara" "Juana"  "Elena"
```

5.2.3 Producto cartesiano y la función `expand.grid()`

Para hallar el producto cartesiano de dos conjuntos se puede emplear la función `expand.grid()`. Por ejemplo, sean los conjuntos $A = \{1, 2\}$ y $B = \{4, 5\}$. Al ejecutar `expand.grid(A,B)` se obtiene como resultado en la consola:

```
  Var1 Var2
1    1    4
2    2    4
3    1    5
4    2    5
```

La función `expand.grid()` ha creado un data frame que representa el producto cartesiano $A \times B$. Esta función tiene otros parámetros que no se mencionarán (puede consultar la ayuda de R o RStudio).

Se puede usar también la función `expand.grid()` para hallar el producto cartesiano cuando al menos uno de sus operandos es un multiconjunto[11]; en tal caso el resultado sería un multiconjunto. Por ejemplo, para $D = \{1, 1, 2\}$ y $E = \{4, 5\}$), `expand.grid(D,E)` resultaría en:

```
  Var1 Var2
1    1    4
2    1    4
3    2    4
4    1    5
5    1    5
6    2    5
```

Ejercicios

1. La tabla siguiente muestra el peso en kilogramos de 10 estudiantes que pertenenecen a 3 grupos. Investigue de qué clase son sus columnas. Para cada modalidad de la variable Grupo calcule su media.

Grupo	a	a	a	b	b	b	b	c	c	c
Peso	55	74	85	65	57	82	79	69	90	74

2. Crear una lista con nombre k1 que contiene los elementos 1, 4 y TRUE en su primera, segunda y tercera posición respectivamente.
   a) Extraer el primer elemento de la lista k1, e imprimir su modo.

[11] En matemáticas un **multiconjunto** (**bag**) difiere de un conjunto en que cada miembro del mismo puede tener una cantidad de apariciones mayor que 1. Por ejemplo, en el multiconjunto {*a*, *a*, *b*, *b*, *b*, *c*}, los elementos *a*, *b*, y *c* aparecen 2, 3, y 1 vez, respectivamente.

b) Agregar un cuarto elemento ‘Buenos días’ a la lista original k1 e imprimir la lista resultante.

3. Crear una lista de nombre “yo” de tres elementos: su primer nombre, su primer apellido y su año de nacimiento. Esos tres elementos serán nombrados respectivamente "Nombre", ", "Apellido" y “Nacimiento”.
   a) Extraer y mostrar el apellido de dos maneras: usando índice y usando el atributo de nombre.
   b) Crear dos nuevas listas con la misma estructura que “yo”, llenarla con datos y darles de nombre “tu” y “el”. Luego crear otra lista con nombre persona que contiene las listas “yo”, “tu” y “el”.
   c) Mostrar nombre y apellido de los elementos en la lista “persona”, nacidas después de 2000.
4. En la siguiente tabla se muestran las materias recibidas por un estudiante y las notas que alcanzó en cada una de ellas. Determine en qué asignaturas obtuvo más de 60 puntos, y en cuáles más de 90. Imprima el resultado.

Materia	Español	Historia	Matemática	Literatura	Computación	Química
Notas	98	90	59	95	60	55

5. Escriba una lista que contenga los nombres de los estudiantes del grupo y sus edades. Determine la edad mayor y muestre el resultado en pantalla.
6. Sea el vector de caracteres siguiente: ("Miguel", "Marcos", "Vanessa", "Ana", "Maximiliano", "Luisa"). Escriba un script que imprima cada nombre que aparece en el vector junto con su longitud (cantidad de caracteres).

7. La tabla que se muestra contiene información referente al Peso, Estatura, Sexo, Equipo y medida de Zapato de 47 estudiantes.

   a) Crear un data frame con estos datos.

   b) Determinar el nombre de las columnas y de las filas de este data frame.

   c) Construir la tabla de frecuencias de la variable Equipo.

   d) Sabiendo que $\mathbf{IMC} = \frac{\mathbf{Peso}}{\mathbf{Estatura^2}}$, calcule el IMC de cada deportista e incorpore esta información al data frame.

Peso	Estatura	Sexo	Equipo	Zapato	Peso	Estatura	Sexo	Equipo	Zapato
75	180	H	OTRO	45	50	161	M	DEP	37
82	186	H	OTRO	45	80	182	H	CEL	45
80	178	H	OTRO	42	70	179	H	DEP	43
78	176	H	RMA	43	71	181	H	DEP	42
92	177	H	OTRO	44	74	162	M	BAR	40
54	170	H	BAR	41	56	166	M	BAR	38
80	182	H	CEL	46	53	155	M	DEP	37
66	171	H	OTRO	43	83	177	H	DEP	43
67	170	H	CEL	41	67	178	H	DEP	42
64	178	H	BAR	45	100	175	H	CEL	43
96	190	H	DEP	45	72	175	H	OTRO	42
70	179	H	OTRO	42	60	180	H	DEP	44
90	173	H	OTRO	43	77	184	H	DEP	45
72	178	M	DEP	41	87	192	H	CEL	45
56	167	M	RMA	38	85	188	H	CEL	46
75	181	H	BAR	45	90	150	H	BAR	42
95	186	H	DEP	43	58	164	M	DEP	40
70	179	H	OTRO	43	52	163	M	DEP	38
75	170	H	DEP	40	60	165	M	DEP	39
78	175	H	DEP	45	49	168	M	CEL	37

75	181	H	DEP	43	58	165	H	OTRO	40
75	174	H	OTRO	44	82	185	H	OTRO	44
56	164	M	RMA	38	65	185	H	OTRO	42
79	179	H	OTRO	42					

Unidad 6. Lectura y escritura de ficheros de datos

Los datos que se han tratado hasta el momento y que han sido procesados residen en la memoria central de la computadora, por tanto, tienen un carácter no persistente, volátil, y cuando la computadora es apagada se pierden, o sea, no están disponibles para futuros procesamientos.

Por otra parte, muchas aplicaciones requieren procesar grandes volúmenes de datos, que por limitaciones obvias no pueden ser procesados en la memoria central, siendo necesario mantenerlos en la memoria auxiliar, donde adquieren persistencia, bajo la denominación de fichero o archivo.

Un ***fichero*** o ***archivo*** es una colección de datos de entidades de una misma tipología, que se almacenan de forma permanente en la memoria auxiliar. A continuación, se explican algunas vías que emplea R para interactuar con ficheros externos.

6.1 Lectura de datos desde ficheros de texto

Si un conjunto de datos viene guardado en un fichero **.txt** se pueden leer con la función **read.table()** que devuelve un **data frame**. Una llamada típica a la función tiene la forma:

```
datos <- read.table(file=nombre_del_fichero, sep="\t", header=T,
                                        dec=".")
```

donde:

- **`datos`** es el nombre de la tabla que recibirá la información leída por `read.table`. De no hacerse la asignación, R se limitará a imprimirlos en la consola.
- **`fichero`** es una cadena de caracteres que denota el fichero que se intenta leer, precedido de un camino absoluto o relativo para su localización. Si el fichero está en el directorio de

trabajo, basta poner su nombre. Puede solicitarse buscar donde se encuentra el fichero, empleando **`file = file.choose().`**

- **`sep`** indica el delimitador que separa los campos de cada línea del fichero. Por defecto es espacio **`" "`**, aquí se dio el delimitador **`"\t"`** (tabulador), que también se acepta por defecto. Otros caracteres pueden ser usados como delimitadores, lo cual debe ser indicado al intérprete de R.
- **`header`** indica que la primera fila del fichero contiene los nombres de las columnas. Por defecto es **`FALSE`**. Si se olvida especificarlo y la primera fila del fichero contiene efectivamente el nombre de las columnas, R interpretará estas erróneamente como datos.
- **`dec`** indica el carácter separador decimal, por defecto **`"."`**.

La función **`read.table()`** tiene otros muchos parámetros que pueden ser consultados en manuales del R. En los próximos ejemplos se usarán los ficheros de texto:

a) **`AptoRentaTab.txt`** con el siguiente contenido dispuesto en 6 líneas:

```
Precio     Piso  Area  Cuartos    Climat
52.00      1     64    2          No
54.75      5     110   3          No
57.50      16    160   4          No
57.50      7     48    1          No
9.75       12    95    3          Si
```

Este fichero se puede crear con cualquier editor de texto (por ejemplo, el block de notas) y guardar en el directorio de trabajo. Note que los datos por filas están separados por una tabulación.

b) **`AptoRentaSNC`**, que consiste en eliminarle a **`AptoRentaTab`** la primera fila.

```
52.00      1     64    2          No
54.75      5     110   3          No
57.50      16    160   4          No
57.50      7     48    1          No
9.75       12    95    3          Si
```

A continuación, se ejemplifican varias posibilidades de lectura de estos ficheros:

a) **Nombre de columnas**: Si; **Separador decimal**: punto; **Separador de columnas**: tabulador.

Script de entrada en R

```
AptoRenta01 <- read.table("AptoRentaTab.txt", header=T)
AptoRenta01
```

Consola de salida de R

```
> AptoRenta01 <- read.table("AptoRentaTab.txt", header=T)
> AptoRenta01
  Precio Piso Area Cuartos Climat
1  52.00    1   64       2     No
2  54.75    5  110       3     No
3  57.50   16  160       4     No
4  57.50    7   48       1     No
5  59.75   12   95       3     Si
```

b) **Nombre de columnas**: Si; **Separador decimal**: punto; **Separador de columnas**: tabulador; el atributo `header` es `FALSE` (valor por defecto).

Script de entrada en R

```
AptoRenta01 <- read.table("AptoRentaTab.txt", header=F)
AptoRenta01
```

Consola de salida de R

```
> AptoRenta01 <- read.table("AptoRentaTab.txt", header=F)
> AptoRenta01
      V1   V2   V3      V4     V5
1 Precio Piso Area Cuartos Climat
2  52.00    1   64       2     No
3  54.75    5  110       3     No
4  57.50   16  160       4     No
5  57.50    7   48       1     No
6  59.75   12   95       3     Si
```

La primera fila del fichero `AptoRentaTab.txt` contiene los nombres de las columnas.

Al no especificar que `header= TRUE,` se asume **erróneamente** que esa fila es una fila

más de datos (por tanto, hay 6 filas de datos) y el sistema internamente le asigna un nombre a cada columna (**V1** hasta **V5**).

c) **Nombre de columnas**: No; **Separador decimal**: punto; **Separador de columnas**: tabulador.

Script de entrada en R

```
fich <- "AptoRentaSNC.txt"
AptoRenta02 <- read.table(fich)
AptoRenta02
```

Consola de salida de R

```
> fich <- "AptoRentaSNC.txt"
> AptoRenta02 <- read.table(fich)
> AptoRenta02
     V1 V2  V3 V4 V5
1 52.00  1  64  2 No
2 54.75  5 110  3 No
3 57.50 16 160  4 No
4 57.50  7  48  1 No
5 59.75 12  95  3 Si
```

Observar que el fichero `AptoRentaSNC` no contiene la fila con los nombres de las columnas y el sistema internamente le asigna un nombre (**V1** hasta **V5**).

d) **Nombre de columnas**: No; pero se desea darle nombres a las filas y a las columnas.

Usando ahora `AptoRentaSNC.txt`

Script de entrada en R

```
AptoRenta03 <- read.table("APtoRentaSNC.txt",
                  row.names = c("A1","A2","A3","A4","A5"),
                  col.names = c("Pr","Pi","Ar","NC","Cli"))
AptoRenta03
```

Consola de salida de R

```
> AptoRenta03 <- read.table("APtoRentaSNC.txt",
+                   row.names = c("A1","A2","A3","A4","A5"),
+                   col.names = c("Pr","Pi","Ar","NC","Cli"))
> AptoRenta03
      Pr Pi  Ar NC Cli
A1 52.00  1  64  2  No
A2 54.75  5 110  3  No
```

```
A3 57.50 16 160  4  No
A4 57.50  7  48  1  No
A5 59.75 12  95  3  Si
```

e) **Nombre de columnas**: No; pero se desea copiar del fichero solo las 3 primeras filas.

Script de entrada en R

```
AptoRenta04 <- read.table("AptoRentaSNC.txt",
        row.names = c("A1","A2","A3"),
        col.names = c("Pr","Pi","Ar","NC","Cli"), nrows = 3)
AptoRenta04
```

Consola de salida de R

```
> AptoRenta04 <- read.table("AptoRentaSNC.txt",
+         row.names = c("A1","A2","A3"),
+         col.names = c("Pr","Pi","Ar","NC","Cli"), nrows = 3)
> AptoRenta04
      Pr Pi  Ar NC Cli
A1 52.00  1  64  2  No
A2 54.75  5 110  3  No
A3 57.50 16 160  4  No
```

f) **Nombre de columnas**: No; pero se desea saltar la primera fila.

Script de entrada en R

```
AptoRenta05 <- read.table("AptoRentaSNC.txt",
        row.names = c("A2","A3","A4","A5"),
        col.names = c("Pr","Pi","Ar","NC","Cli"), skip = 1)
AptoRenta05
```

Consola de salida de R

```
> AptoRenta05 <- read.table("AptoRentaSNC.txt",
+         row.names = c("A2","A3","A4","A5"),
+         col.names = c("Pr","Pi","Ar","NC","Cli"), skip = 1)
> AptoRenta05
      Pr Pi  Ar NC Cal
A2 54.75  5 110  3 No
A3 57.50 16 160  4 No
A4 57.50  7  48  1 No
A5 59.75 12  95  3 Si
```

Las funciones **`read.csv()`** y **`read.csv2()`** son análogas a **`read.table()`** excepto por sus parámetros por defecto.

Para **read.csv()**:

- **sep** es por defecto **","** (carácter coma).
- **header** es por defecto **TRUE**.
- **dec** (punto decimal) por defecto es el carácter punto **".”.**

Para **read.csv2()**:

- **sep** es por defecto **";"** (carácter punto y coma).
- **header** es por defecto **TRUE**.
- **dec** por defecto es el carácter coma **","**.

Ambas funciones tienen la misión de leer ficheros de datos delimitados por coma (**.csv**). La variante **read.csv2()** se usa en países que usan la coma como punto decimal, y el punto y coma como separador de campo. Por ejemplo, sea el fichero externo **AptoRentaComa.csv**:

```
Precio,Piso,Area,Cuartos,Climat
52.00,1,64,2No
54.75,5,110,3,No
57.50,16,160,4,No
57.50,7,48,1,No
59.75,12,95,3,Si
```

el cual puede ser leído con **read.csv("AptoRentaComa.csv")**:

Script de entrada en R

```
AptoRenta06 <- read.csv("AptoRentaComa.csv")
AptoRenta06
```

Consola de salida de R

```
> AptoRenta06 <- read.csv("AptoRentaComa.csv")
> AptoRenta06
  Precio Piso Area Cuartos Calef
1  52.00    1   64       2    No
2  54.75    5  110       3    No
3  57.50   16  160       4    No
4  57.50    7   48       1    No
5  59.75   12   95       3    Si
```

Un fichero **.csv** puede ser formado también con Excel.

En vez de especificar el parámetro fichero se puede emplear la función **file.choose()** que de forma interactiva permite navegar por los directorios para encontrar donde se encuentra el fichero de datos a importar, por ejemplo, con **df <- read.csv(file.choose())** se abrirá una ventana que le permitirá buscar el fichero deseado.

6.1.1 Entrada de valores a una matriz desde un fichero externo

Para leer datos desde un fichero de texto hacia una matriz puede usarse también la función `read.table()`, pero debido a que esta función devuelve un objeto de la clase `data.frame`, dicho objeto leído debe sufrir coerción hacia la clase `matrix`:

```
as.matrix(read.table(file = "dirección_fichero", sep = "", header
                     = TRUE, dec = "."))
```

Ejemplo:

Sea la tabla de datos siguiente, de 3 columnas y 3 filas, guardada en su directorio de trabajo en el fichero coord.txt:

x y z

1 2 3

4 5 6

7 8 9

Note que cada columna tiene un nombre. A continuación, mostramos su lectura hacia una matriz y su procesamiento.

Script de entrada en R

```
puntos<-as.matrix(read.table("coord.txt", sep = " ", header = TRUE))
print(puntos)
print(puntos[1,2])
```

Consola de salida de R

```
> puntos<-as.matrix(read.table("coord.txt", sep = " ", header = TRUE))
> print(puntos)
  x y z
1 1 2 3
2 4 5 6
3 7 8 9
> puntos[1,2]
[1] 2
```

Si se elimina la primera línea del fichero **coord.txt** y se salva el nuevo fichero con el nombre **coordsn.txt**, y la primera línea del programa se transforma en:

```
puntos<-as.matrix(read.table("coordsn.txt", sep=" ", header=FALSE))
```

al ejecutar el programa completo la salida en el panel de consola será:

```
> puntos<-as.matrix(read.table("coordsn.txt", sep=" ", header=FALSE))
> print(puntos)
  V1 V2 V3
1  1  2  3
2  4  5  6
3  7  8  9
> print(puntos[1,2])
[1] 2
```

Observe que la fila con los nombres ha sido sustituida por una por defecto.

Para leer matrices grandes, especialmente aquellas con muchas columnas, la función `scan()` es más flexible que `read.table()`.

Supongamos el fichero mat_x.txt:

```
1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1 0
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1
1 1 0 1 1 0 1 1 0 1 1 0 1 1 0 1 1
0 0 1 0 0 1 0 0 1 0 0 1 0 0 1 0 0
```

Una lectura hacia la matriz **x** usando **scan()** se da a continuación:

Script de entrada en R

```
x <- matrix(scan("mat_x.txt"), nrow=5, byrow=T)
x
```

Consola de salida de R

```
> x <- matrix(scan("mat_x.txt"), nrow=5, byrow=T)
Read 60 items
> x
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10] [,11] [,12]
[1,]    1    0    1    1    0    1    1    0    1     1     0     1
[2,]    1    1    1    1    1    1    1    1    1     1     1     1
[3,]    1    1    0    1    1    0    1    1    0     1     1     0
[4,]    1    1    0    1    1    0    1    1    0     1     1     0
[5,]    0    0    1    0    0    1    0    0    1     0     0     1
```

La función **scan()** comparte algunos de los parámetros que posee **read.table()** y otros, los que pueden ser consultados en los manuales del lenguaje.

6.2 Salida de datos a fichero externo

Consideremos que formamos un data frame con datos referente a rentas de apartamentos:

```
Precio <- c("52.00","54.75","57.50","57.50","59.75")

Piso <- c(1,5,16,7,12)

Area <- c(64,110,160,48,95)

Cuartos <- c(2,3,4,1,3)

Climat <- c("No","No","No","No","Si")
```

```
renta.apto <- data.frame(Precio,Piso,Area,Cuartos,Climat)
```

Precio es el importe mensual, **Piso** el piso donde se encuentra el apartamento, **Area** es la superficie total, **Cuartos** es la cantidad de cuartos que tiene el apartamento y **Climat** si tiene alguna forma de climatización (aire acondicionado o Split). El data frame formado sería:

Precio	Piso	Área	Cuartos	Climat
52.00	1	64	2	No
54.75	5	110	3	No
57.50	16	160	4	No
57.50	7	48	1	No
59.75	12	95	3	Si

Para registrar datos a un fichero externo se usa la función **`write.table()`**. El encabezado típico de esta función es:

```
write.table(x, file = "", append = FALSE, sep = " ", dec = ".",
                    row.names = TRUE, col.names = TRUE, ...)
```

donde:

- **`x`**: objeto a ser escrito, preferiblemente una matriz o un data frame. En otro caso se intenta coercionar a un dataframe.
- **`file`**: una conexión de datos o una cadena de caracteres con el nombre del fichero donde se escribirá x. Si **`file = ""`**, indica salida por consola.
- **`append:`** valor lógico relevante si **`file`** es una cadena de caracteres. Si es **`TRUE`** la salida **`x`** se agrega al fichero, si es **`FALSE`** la información que se encuentra en el fichero **`file`** es destruida antes de grabar la salida **`x`**.
- **`sep:`** indica el separador de los valores en cada fila de x. Por defecto es **`" "`**.

- **`dec:`** carácter simple usado para indicar el separador decimal en valores numéricos o complejos.
- **`row.names:`** un valor lógico indicando si los nombres de filas se escribirán junto con x, o un vector de caracteres con los nombres de filas que serán escritos.
- **`col.names:`** un valor lógico indicando si los nombres de columnas se escribirán junto con **`x`**, o en un vector de caracteres con los nombres de columnas que serán escritos.

Ejemplo:

Para guardar el data frame `renta.apto` en el directorio de trabajo creado, basta escribir los comandos:

```
write.table(renta.apto, "AptoRentaTab.txt")
```

Otro ejemplo:

Script de entrada en R

```
Nombre <- c("Juan","Laura")
Edad <- c(12,10)
df4 <- data.frame(Nombre,Edad)
df4
write.table(df4,"kds.txt")
df5 <- read.table("kds.txt")
df5
```

Consola de salida de R

```
> Nombre <- c("Juan","Laura")
> Edad <- c(12,10)
> df4 <- data.frame(Nombre,Edad)
> df4
  Nombre Edad
1   Juan   12
2  Laura   10
> write.table(df4,"kds.txt")
> df5 <- read.table("kds.txt")
> df5
  Nombre Edad
1   Juan   12
2  Laura   10
```

De forma análoga a la entrada de datos de un fichero de texto, existen las funciones **write.csv (...)** y **write.csv2(...).**

En caso de dar salida a los elementos de una matriz hacia un fichero, se debe establecer en el llamado que no hay filas ni columnas de nombres, por ejemplo:

```
write.table(xc, "xcnew", row.names = F, col.names = F)
```

Sin embargo, para escribir en ficheros externos vectores o matrices, resulta más ventajoso emplear la función **write()**:

```
write(x, file = "data", ncolumns = if(is.character(x)) 1 else 5,
      append = FALSE, sep = " ")
```

donde:

- **x** y **file** como antes.
- **ncolumns**: número de columnas en que se escribirán los datos (por defecto 5 para numéricos, 1 para carácter).
- **append**: Si es **TRUE** el conjunto de datos x se agrega a la conexión indicada.
- **sep**: indica el separador de columnas. Por defecto es **" "**. Para salidas delimitadas por tabulador debe usarse **sep = "\t"**.

Ejemplos:

Script de entrada en R

```
nuevovec <- c(2,4,3,6,8,10)
# Se graba el vector nuevovec en el fichero de nombre Nvector
write(nuevovec, "Nvector.txt")
# Se lee el fichero Nvector
v3 <- scan("Nvector.txt")
print(v3)
```

Consola de salida de R

```
> nuevovec <- c(2,4,3,6,8,10)
> # Se graba el vector nuevovec en el fichero de nombre Nvector
> write(nuevovec, "Nvector.txt")
```

```
> # Se lee el fichero Nvector
> v3 <- scan("Nvector.txt")
Read 6 items
> print(v3)
[1]  2  4  3  6  8 10
```

En el siguiente ejemplo se graba y lee una matriz:

Script de entrada en R

```
# Se forma por columnas la matriz mat3 de 3x3
mat3 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)
mat3
# Se graba la matriz mat3 en el fichero fmat3
write.table(mat3, "fmat3", row.names=F, col.names=F)
# Se lee de fmat3 una matriz de 3 filas formada por filas
mat5 <- matrix(scan("fmat3"), nrow=3, byrow=T)
mat5
class(mat5)
```

Consola de salida de R

```
> # Se forma por columnas la matriz mat3 de 3x3
> mat3 <- matrix(c(1, 2, 3, 4, 5, 6, 7, 8, 9), ncol = 3)
> mat3
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> # Se graba la matriz mat3 en el fichero fmat3
> write.table(mat3, "fmat3", row.names=F, col.names=F)
> # Se lee de fmat3 una matriz de 3 filas formada por filas
> mat5 <- matrix(scan("fmat3"), nrow=3, byrow=T)
Read 9 items
> mat5
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> class(mat5)
[1] "matrix" "array"
```

6.3 Introduciendo datos desde el teclado

Las funciones **`scan()` y `data.entry()`** se pueden usar para introducir datos desde el teclado. Usando `scan()` se entran los datos por la consola, uno por línea (en el caso de un

vector numérico pueden entrarse varios números en una línea separados por espacio) y en la última línea se da Enter (↵) para indicar el fin de la entrada. En el siguiente script se muestran las dos formas de introducir vectores. La función **what()** especifica el tipo que se le dará a los datos a entrar (logical, integer, numeric, complex, character y list).

Script de entrada en R

```
#Introduciendo datos desde el teclado con la función scan()
B<-scan("", what=list(Edad=numeric(),Peso=numeric(),Nombre=character()))
#Introducir los datos desde el teclado. Uno por línea
B
B2<-scan("",what=list(Edad=numeric(),Peso=numeric(),Nombre=character()))
#Introducir los datos desde el teclado. Cada vector en una línea. Los
# elementos del vector separados por un espacio
B2
#Para introducir un vector
z <- scan()
z
```

Consola de salida de R

```
> #Introduciendo datos desde el teclado con la función scan()
> B<-scan("",what=list(Edad=numeric(),Peso=numeric(),Nombre=character()))
1: 50
1: 130
1: Ana
2: 25
2: 100
2: Maria
3:
Read 2 records
> B
$Edad
[1] 50 25

$Peso
[1] 130 100

$Nombre
[1] "Ana"   "Maria"

> B<-scan("",what=list(Edad=numeric(),Peso=numeric(),Nombre=character()))
1: 50 130 Ana
2: 25 100 Maria
3:
Read 2 records
```

```
> B
$Edad
[1] 50 25

$Peso
[1] 130 100

$Nombre
[1] "Ana"   "Maria"

> #Para introducir un vector
> z <- scan()
1: 1 2 3 4 5 6 7 8 9 10
11:
Read 10 items
> z
 [1]  1  2  3  4  5  6  7  8  9 10
```

Con `data.entry()` (más cómodo), se debe crear un vector, matriz, etc., inicializado a 0 y después aplicarle la función para abrir una hoja que nos permite editar fácilmente el conjunto de datos. Por defecto **`what = double()`**, por lo que si no se especifica y **`file = ""`** se podrán introducir vectores.

Script de entrada en R

```
#Introduciendo datos desde el teclado con la función data.entry
A <- matrix(data=0,nrow=2, ncol=3) #Se genera uma matriz nula de 2x3
data.entry(A, Names = c("C1","C2","C3"))
# Introducir los datos en la hoja que se ha abierto
print(A)
```

Consola de salida de R

```
> #Introduciendo datos desde el teclado con la función data.entry
> A <- matrix(data=0,nrow=2, ncol=3) #Se genera uma matriz nula de
2x3
> data.entry(A, Names = c("C1","C2","C3"))
> # Introducir los datos en la hoja que se ha abierto
> print(A)
     C1 C2 C3
[1,]  1  3  5
[2,]  2  4  6
```

Desde el teclado también se puede introducir una cadena de caracteres, mediante la función **`readline()`.** A continuación, se muestran dos ejemplos de su uso:

```
w <- readline()
# Teclear en la consola una cadena de caracteres, que se guarda en w
w
inic <- readline("Escriba sus iniciales: ")
# Escribir sus iniciales en la consola, las que se guardarán en inic
```

6.4 Importando y exportando datos desde/hacia el exterior de R

Como ya planteamos, R es un entorno especialmente diseñado para el tratamiento de datos, por tanto, es importante ser capaces de importar datos externos.

6.4.1 Importación y exportación de datos desde/hacia Excel

Existen varias formas de importar datos desde Excel:

a) Muy simple, usando el siguiente algoritmo en dos pasos:

1) Los datos a importar de Excel deben ser guardados dentro de Excel a datos delimitados por tabulador o por comas.

2) Importar en R esos archivos con `read.delim()` o `read.csv()`; también, si fuera el caso, con `read.delim2()` o `read.csv2()` si se usa coma en vez de punto decimal.

   Al ejecutar las instrucciones:

   ```
   a <-read.csv("AptoRentaComa.csv", fill = TRUE)
   a
   class(a)
   ```

   se obtendrá:

   ```
   > a <-read.csv("AptoRentaComa.csv", fill = TRUE)
   > a
     Precio Piso Area Cuartos Calef
   ```

```
1  52.00    1   64       2    No
2  54.75    5  110       3    No
3  57.50   16  160       4    No
4  57.50    7   48       1    No
5  59.75   12   95       3    Si
> class(a)
[1] "data.frame"
```

b) Usando el mecanismo de corta y pega

Con la ayuda del mouse se selecciona en la hoja de Excel el conjunto de celdas cuyos datos se desean incorporar en R. Luego se copia usando el menú Edit de Excel o la combinación de teclas ***CTRL+C*** hacia el clipboard. Entonces se escribe en la consola de R la instrucción siguiente para que los datos se transfieran del clipboard a un dataframe:

```
x <- read.table(file("clipboard"),sep="\t",header=TRUE)
```

La función **`fix(x)`** abre un minitablero en el ambiente de R que permite visualizar y modificar los datos presentados en **`x`**.

c) Usando paquetes especializados

Existen paquetes que permiten leer directamente ficheros de formato **.xls** o **.xlsx** desde R. Por ejemplo, dentro del paquete **`gdata`** existe la función **`read.xls`**. Para emplear este paquete se requiere que en la computadora se encuentre instalado el lenguaje PERL (que está garantizado si está instalado RTools.exe).

A continuación, se muestra un ejemplo para exportar datos hacia Excel de una de las formas posibles. Para ello se crea un dataframe que luego se exportará a Excel, escribiendo en la consola de R:

```
x <- data.frame(Peso=c(80,90,75),Talla=c(182,190,160))
write.table(x,file("clipboard"),sep="\t",row.names=FALSE)
```

Los datos en x son copiados hacia el clipboard. Para depositarlos en una hoja de Excel basta seleccionar la celda superior izquierda de la zona en que se desea situarlos y entonces pegar (dando por ejemplo ***CTRL+V***).

6.4.2 Importación de datos desde SPSS, Minitab, SAS, MatLab y similares

A continuación, se muestra mediante un ejemplo la lectura de ficheros externos almacenados con SPSS, para lo cual se necesita cargar el paquete **`foreign`**.

Ejemplo:

```
a <- read.spss("covid.sav", to.data.frame = T)
a
```

obteniéndose la salida en consola:

```
> a <- read.spss("covid.sav", to.data.frame = T)
re-encoding from UTF-8
> a
      Fecha Pruebas Positivos Altas Defunciones
1 17-APR-20    1480        61    21           4
2 18-APR-20    1895        63    35           1
3 19-APR-20    1673        49    28           2
```

La tabla 8 muestra información acerca de diversas importaciones de datos a R.

Tabla 8. Importación de datos a R

Software	Extensión del fichero	Paquete requerido	Función de R	Tipo de resultado
SPSS	.sav	foreign	`read.spss()`	list
Minitab	.mtp[12]	foreign	`read.mtp()`	list
SAS	.xpt	foreign	`read.xport()`	dataframe
Matlab	.mat	R.matlab	`readMat()`	list

6.5 Accediendo a ficheros que acompañan al sistema base de R

Desde R se puede acceder a un conjunto de ficheros que acompañan a su sistema base. En la figura 14 se muestra una parte de estos ficheros, lo cual se obtuvo dando clic al nombre del paquete **datasets** en el tabulador **Packages**.

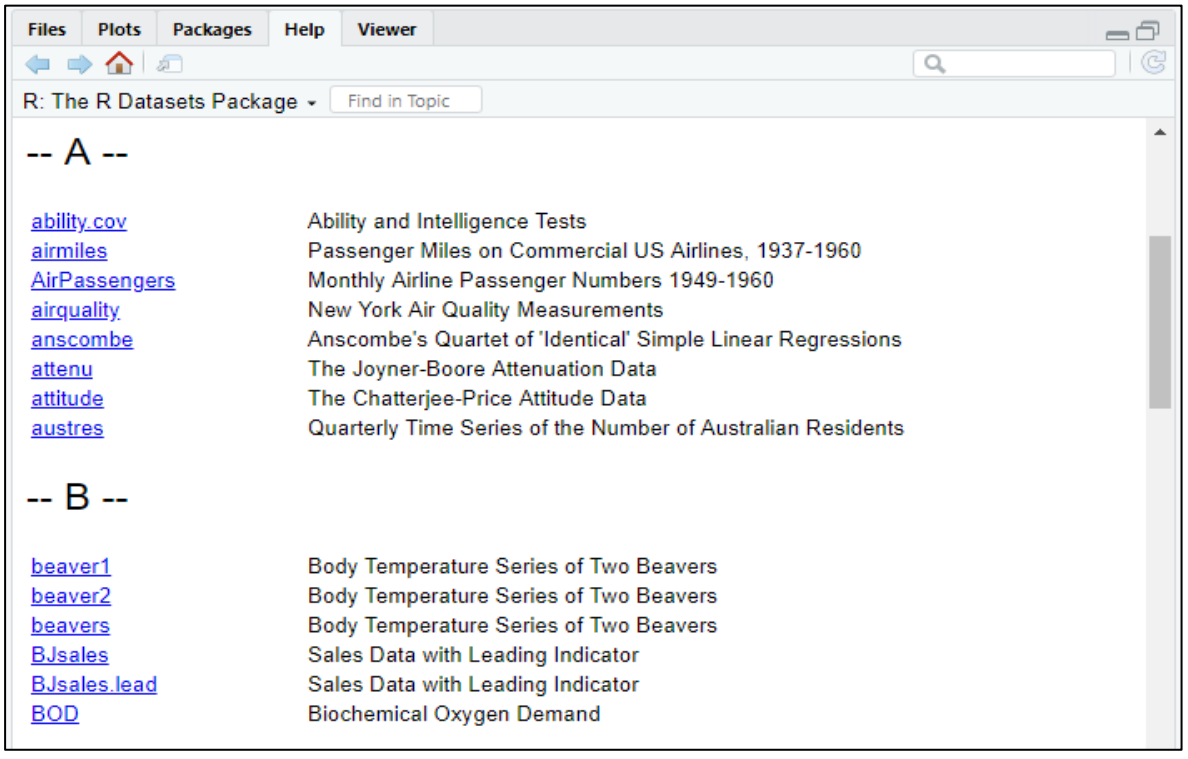


Figura 14. Fragmento de los ficheros almacenados en el paquete datasets.

[12] El fichero debe estar en formato de Minitab Portable Worksheet.

Para cargar un fichero de datos existente en **datasets** se emplea la función **`data(`**`nombre del fichero`**`)`** y para mostrar la base de datos seleccionada basta con escribir su nombre. Como ejemplo vamos a cargar dos ficheros "infert" e "iris".

Script de entrada en R

```
# Cargando dos ficheros de datos existenten en el sistema base de R
data(infert)    # Carga el fichero infert
infert          # Muestra el contenido del fichero

data(iris)
iris
```

Consola de salida de R

```
> # Cargando dos ficheros de datos existenten en el sistema base de R
> data(infert)    # Carga el fichero infert
> infert          # Muestra el contenido del fichero
   education age parity induced case spontaneous stratum pooled.stratum
1     0-5yrs  26      6       1    1           2       1              3
2     0-5yrs  42      1       1    1           0       2              1
.....
> data(iris)
> iris
    Sepal.Length  Sepal.Width  Petal.Length  Petal.Width  Species
1            5.1          3.5           1.4          0.2   setosa
2            4.9          3.0           1.4          0.2   setosa
.....
```

6.6 Importando ficheros con campos de longitud fija

Los ficheros con campos de longitud fija son ficheros de texto en los que las columnas de datos no se separan por ningún delimitador, pero que poseen un ancho fijo de caracteres. Para ello se emplea la función `read.fwf()` con formato simplificado:

```
read.fwf(fich, ancho, header = FALSE, sep = "\t", skip = 0,...)
```

donde:

- **`fich`**: nombre del fichero de donde se leen los datos.
- **`ancho`**: vector entero con los anchos de cada campo.

- **`header`**: indica si la primera fila del fichero contiene los nombres de las columnas; por defecto es **`FALSE`**.
- **`sep`**: separador que se usa al importar los valores, que no puede aparecer en el fichero, excepto en el encabezado.
- **`skip`**: cantidad de filas iniciales que se salta al importar. Por defecto es 0.

La función devuelve un dataframe con la información contenida en el fichero, separada por campos.

Ejemplo:

Sea el fichero fwfpais.txt

```
FranciaFR14.01
Cuba   CU23.02
Chile  CL32.96
Italia IT15.90
Rusia  RU25.48
```

Cada registro tiene 3 campos: nombre del país (7 caracteres), abreviatura (2 caracteres) y un valor (5 caracteres). Obsérvese como se forma un **data.frame** a partir de los datos en el fichero fwfpais.txt:

Script de entrada en R

```
df_fwf <- read.fwf("fwfpais.txt", widths = c(7,2,5))
df_fwf
class(df_fwf)
```

Consola de salida de R

```
> df_fwf <- read.fwf("fwfpais.txt", widths = c(7,2,5))
> df_fwf
       V1   V2      V3
1 Francia   FR   14.01
2 Cuba      CU   23.02
3 Chile     CL   32.96
```

```
4 Italia    IT   15.90
5 Rusia     RU   25.48
> class(df_fwf)
[1] "data.frame"
```

6.7 Ficheros .rds y .RData

Los ficheros `.rds` y `.Rdata` pueden ser usados para almacenar objetos de R en código nativo. Esto da diferentes ventajas comparado con otras formas de almacenamiento no nativas como `write.table()`:

- Es más rápido restaurar los datos en R.
- Mantiene información específica de R codificada en el dato (por ejemplo: atributos, tipos de variables, etc.).

Por ejemplo, para salvar el conjunto de datos iris en un fichero .rds, se haría:

```
saveRDS(object = iris, file = "mi_df.rds")
```

Luego, para cargarla:

```
iris2 <- readRDS(file = "mi_df.rds")
```

Las funciones `saveRDS()/readRDS()` solo procesan un objeto de R. Sin embargo, son más flexibles que el almacenamiento multiobjeto, pues el nombre del objeto restaurado no tiene que ser el mismo del objeto cuando fue almacenado.

Para salvar multiples objetos se emplea la función save() en un fichero de salida .Rdata. Por ejemplo, para salvar los conjuntos de datos iris y cars en un fichero, se puede hacer:

```
save(iris, cars, file = "myIrisAndCarsData.Rdata")
```

Para cargarlos luego:

```
load("myIrisAndCarsData.Rdata")
```

Con esto, ambos ficheros (iris y cars) pueden ser procesados.

Ejercicios

1. Introduzca en un fichero los siguientes datos correspondientes a la altura promedio, por sexo, de la población de 5 países. Las columnas las separa por coma.

País/Región	**Altura promedio (hombres)**	**Altura promedio (mujeres)**
Argentina	174.50 cm	162.77 cm
Chile	174.20 cm	167.80 cm
Colombia	173.60 cm	161.60 cm
El Salvador	165.60 cm	160.30 cm
Venezuela	170.60 cm	153.00 cm

   a) Guarde estos datos en un fichero .txt.

   b) Calcule la altura promedio de dos maneras, sin importar el sexo y por sexo.

   c) Diga en qué países la estatura promedio de sus habitantes es superior a los 170 cm.

2. Introduzca desde el teclado usando **`data.entry()`** los siguientes datos:

	Sex: Male				**Sex: Female**			
Hair\Eye	**Brown**	**Blue**	**Hazel**	**Green**	**Brown**	**Blue**	**Hazel**	**Green**
Black	32	11	10	3	36	9	5	2
Brown	53	50	25	15	66	34	29	14
Red	10	10	7	7	16	7	7	7
Blond	3	30	5	8	4	64	5	8

   a) Guarde los datos en el fichero Color.Pelo.Ojos.txt.

   b) Lea los datos guardados.

3. Con el fichero “iris” que se encuentran en el paquete base de R (datasets):

   a) Investigue los nombres de sus columnas y filas.

   b) Escriba en español los nombres de sus columnas.

   c) Salve el fichero transformado para su posterior uso.

d) Lea el fichero .txt que acabó de salvar.

e) Determine la cantidad de especies que existen en el fichero “iris”.

4. Crear un data frame usando el siguiente código:

```
N <- 10000

df <- data.frame(x = 1:N,y = runif(N))
```

Exporte el data frame resultante a ficheros con formato .csv y .rds.

a) ¿Cuál formato ocupa menos espacio? Pruébelo usando la función **`file.size()`** para chequear el tamaño de un fichero.

b) ¿Cuál exportación (la instrucción de grabación de datos) es más veloz? Pruébelo usando la función **`system.time()`**?

La función `system.time()` permite medir el tiempo de corrida. Ella retorna tres valores:

- **user CPU time**: cantidad de segundos que el CPU gastó haciendo los cálculos.
- **system CPU time**: cantidad de tiempo que el sistema operativo gastó respondiendo a las peticiones del programa.
- **elapsed time**: suma de los dos tiempos anteriores más tiempos de espera en la corrida del programa.

Unidad 7. Estructuras de control y manejo de datos

En los lenguajes de programación, se entiende por **estructuras de control** aquellas construcciones sintácticas del lenguaje que permiten "controlar" el orden de ejecución de las operaciones sobre los datos. Por ejemplo, prácticamente todos los lenguajes tienen una construcción "**if**", que permite ejecutar o saltar un conjunto, bloque o secuencia de instrucciones dentro del código de un programa. R también cuenta con un conjunto de estructuras de control, algunas de las cuales estudiaremos a continuación.

Las estructuras de control a nivel de sentencias (instrucciones) son:

- secuencia o instrucción compuesta
- alternativas o condicionales: if – else, ifelse, switch
- iterativas o ciclos: for, while, repeat

La secuencia o instrucción (comando) compuesta es la disposición consecutiva de las instrucciones de un programa. Su semántica informal es que se ejecutan las instrucciones en el orden que aparecen. Los comandos que componen una instrucción compuesta pueden encerrarse entre llaves **`{ }`**.

7.1 Instrucciones alternativas o condicionales (`if – else`)

Estas estructuras permiten representar situaciones donde es necesario evaluar una condición y escoger entre una de dos alternativas de ejecución. En muchos casos la parte `else` es opcional, pudiéndose presentar en una forma más corta.

La forma más simple:

```
if (Cond) {
   S1
}
```

La forma completa:

```
if (Cond) {
  S1
} else {
  S2
}
```

donde **Cond** es una expresión lógica, y **S1** y **S2** son secuencias de instrucciones (o una expresión), cada una de las cuales se separa por "**;**" o cambio de línea.

Su semántica es como sigue: se evalúa la expresión **Cond**, si es verdadera, se ejecuta **S1** y termina (forma simple). En la forma completa, se ejecuta **S1** si **Cond** es verdadera y si es falsa **S2**. Estas construcciones son semejantes a las de otros lenguajes de programación, con una salvedad, la construcción en sí misma regresa un valor, que puede, si se quiere, ser asignado a una variable o utilizado de otras maneras.

Por ejemplo, el siguiente script muestra como se incrementa en un 25 % el estipendio de un estudiante que tiene promedio de notas mayor o igual a 4.

Script de entrada en R

```
estipendio <- 100
promedio <- 4.5
# Incrementar estipendio un 25%, si el promedio es mayor que 4
if (promedio > 4) {
  estipendio <- estipendio + 0.25*estipendio
}
cat("El nuevo estipendio es ", estipendio, "\n")
```

Consola de salida de R

```
> estipendio <- 100
> promedio <- 4.5
> # Incrementar estipendio un 25%, si el promedio es mayor que 4
> if (promedio > 4) {
+   estipendio <- estipendio + 0.25*estipendio
+ }
```

```
> cat("El nuevo estipendio es ", estipendio, "\n")

El nuevo estipendio es 125
```

Si entre llaves solo hay una expresión, las llaves pueden omitirse, respetando la exigencia sintáctica: si existe **`else`** esta palabra debe escribirse en la misma línea donde se cierra el bloque S1 (sea con la llave "**}**" o no).

La instrucción condicional del script anterior puede escribirse también:

```
if (promedio > 4) estipendio <- estipendio + 0.25*estipendio
```

Arriba se ha mostrado la forma simple de la instrucción **`if`**. En el siguiente script se muestra la forma completa, para determinar si todos los valores en un vector son mayores que otro dado, de cumplirse la condición se imprime un mensaje y de no cumplirse otro.

Script de entrada en R

```
num <- c(1,2,3)
d <- 2
if (all(num > d)) {
  cat("Todos los valores en num son > que ", d, "\n")
} else {
  cat("No todos los valores en num son > que ", d, "\n")
}
```

Consola de salida de R

```
> num <- c(1,2,3)
> d <- 2
> if (all(num > d)) {
+   cat ("Todos los valores en num son > que ", d, "\n")
+ } else {
+   cat ("No todos los valores en num son > que ", d, "\n")
+ }

No todos los valores en num son > que 2
```

El siguiente código asigna a un objeto la evaluación de una expresión condicional:

```
x <- -4

positivo <- if (x>0) {
```

```
                  TRUE
                } else {
                  FALSE
                }
    positivo
```

Su ejecución daría como resultado:

```
    [1] FALSE
```

La anterior condicional, que solo devuelve un valor, puede escribirse como:

```
    positivo <- if (x>0) TRUE else FALSE
```

Hay situaciones prácticas que requieren el empleo de una instrucción condicional más compleja con varias ramas `if - else`, como se muestra a continuación:

```
    if (C1) {
       S1
    } else if (C2) {
       S2
    } else if (C3) {
       S3
    } else {
       S4
    }
```

Ejemplo: Imprimir un título que cualifica una nota dada.

Script de entrada en R

```
nota <- 75
q <- if (nota < 60) {
        "Desaprobado"
```

```
     } else if (nota < 80) {
        "Aprobado"
     } else if (nota < 90) {
        "Bien"
     } else {
        "Excelente"
     }
print(q)
```

Consola de salida de R

```
> nota <- 75
> q <- if (nota < 60) {
+         "Desaprobado"
+      } else if (nota < 80) {
+         "Aprobado"
+      } else if (nota < 90) {
+         "Bien"
+      } else {
+         "Excelente"
+      }
> print(q)
[1] "Aprobado"
```

Otro ejemplo: Dados tres valores de ángulos, verificar que corresponden a un triángulo y en tal caso clasificarlo en rectángulo, acutángulo y obtusángulo.

Script de entrada en R

```
angulo1 <- 60
angulo2 <- 60
angulo3 <- 60
# Determinando si los ángulos corresponden a un triángulo
if ((angulo1+angulo2+angulo3) == 180 &&
    angulo1>0 && angulo2>0 && angulo3>0) { # Hay triángulo
    # Determinando tipo de triángulo según medidas de sus ángulos
    if (angulo1==90 || angulo2==90 || angulo3==90) {
        print("Forman un triángulo rectángulo")
    } else if (angulo1>90 || angulo2>90 || angulo3>90) {
        print("Forman un triángulo obtusángulo")
    } else {
        print("Forman un triángulo acutángulo")
    }
} else {
    print("Esos valores no corresponden a ángulos de un triángulo")
}
```

Consola de salida de R

```
> angulo1 <- 60
> angulo2 <- 60
> angulo3 <- 60
# Determinando si los ángulos corresponden a un triángulo
> if ((angulo1+angulo2+angulo3) == 180 &&
+     angulo1>0 && angulo2>0 && angulo3>0) { # Hay triángulo
+   # Determinando tipo de triángulo según medidas de sus ángulos
+   if (angulo1==90 || angulo2==90 || angulo3==90) {
+      print("Forman un triángulo rectángulo")
+   } else if (angulo1>90 || angulo2>90 || angulo3>90) {
+      print("Forman un triángulo obtusángulo")
+   } else {
+      print("Forman un triángulo acutángulo")
+   }
+ } else {
+   print("Esos valores no corresponden a ángulos de un triángulo")
+ }
[1] "Forman un triángulo acutángulo"
```

7.2 Instrucciones iterativas o ciclos

Estas instrucciones permiten representar algoritmos que exigen la ejecución repetida de un conjunto de instrucciones. Cada ejecución repetida es un ***ciclo*** o ***lazo***.

Un lazo consta de tres partes fundamentales:

- La inicialización: consiste en establecer bajo qué condiciones debe comenzar la ejecución del ciclo.
- La finalización: consistente de una expresión lógica que es probada en cada iteración y cuyo valor permite saber si el ciclo itera una vez más o si termina.
- El cuerpo del ciclo: es el conjunto de instrucciones que se ejecuta repetidamente.

El lenguaje cuenta con varios tipos de **ciclos**:

- Ciclo con un número determinado de iteraciones.
- Ciclo con precondición.
- Ciclo generalizado.

7.2.1 Ciclo con un número determinado de iteraciones

La construcción que posibilita este ciclo es la instrucción **`for`**. El ciclo con for evalúa una expresión iterando sobre los valores de un vector o lista dada. Su sintaxis es:

```
for (var in vector) {
   S
}
```

donde:

- **`var`** es la variable de control del ciclo y toma los valores consecutivos que aparecen en **`vector`**.
- **`S`** es la expresión que se evalúa repetidamente o conjunto de expresiones.

Ejemplo:

El siguiente ciclo permite escribir cada palabra del vector **`c("aprendiendo","ciclo", "for")`** en una línea diferente, empleando a **`palabra`** como variable de control.

Script de entrada en R

```
for (palabra in c("aprendiendo","ciclo", "for")) {
  cat("La palabra actual es", palabra, "\n")
}
```

Consola de salida de R

```
> for (palabra in c("aprendiendo","ciclo", "for")) {
+   cat("La palabra actual es", palabra, "\n")
+ }
La palabra actual es aprendiendo
La palabra actual es ciclo
La palabra actual es for
```

7.2.2 Ciclo con precondición

Da la posibilidad de representar ejecuciones repetidas mientras una cierta condición resulte satisfecha. Resulta muy útil cuando se quieren representar lazos para los cuales no es posible

calcular las veces que van a ser ejecutados, por ejemplo, un enunciado del tipo: “leer y sumar números hasta que sea leído un cero”, su sintaxis es:

```
while (Cond) {
    S
}
```

Esta instrucción permite que, mientras se cumpla la condición `Cond`, se ejecute la instrucción o conjunto de instrucciones `S`. Debemos tener presente que la condición es evaluada antes de entrar al ciclo (por eso se llama a este ciclo con precondición), y por ello puede resultar que la instrucción `S` no se ejecute ni siquiera una vez.

Ejemplo:

En el siguiente ciclo se inicializa x con 0 y en cada iteración se imprime y luego se incrementa su valor en 1, mientras el valor de x sea menor o igual a 5.

Script de entrada en R

```
x <- 0
while (x <= 5) {
  print(x)
  x <- x + 1
}
```

Consola de salida de R

```
> x <- 0
> while (x <= 5) {
+   print(x)
+   x <- x + 1
+ }
[1] 0
[1] 1
[1] 2
[1] 3
[1] 4
[1] 5
```

Note que si quitamos del cuerpo del ciclo la instrucción `x <- x + 1`, entonces el lazo imprime de forma infinita el valor 0. Una salida similar, basada en vectorización, se obtiene con `cat(0:5, sep="\n")`.

7.2.3 Ciclo generalizado

Tiene la estructura:

```
repeat S
```

donde S es el conjunto de instrucciones del cuerpo del ciclo. Permite realizar el cuerpo del ciclo hasta tanto no ocurra la interrupción de la ejecución indicada con un comando de ruptura del lazo, como `break`.

El siguiente ejemplo simula tiradas aleatorias de dos dados, hasta que ambos tengan igual valor.

Script de entrada en R

```
# Se inicializa contador de tiradas
contador <- 0
# Contar tiradas de dos dados hasta que aparezcan valores repetidos
repeat {
  # dado1 y dado2 toman un valor aleatorio entre 1 y 6
  dado1 <- round(runif(1,1,6))
  dado2 <- round(runif(1,1,6))
  # Se incrementa contador de tiradas
  contador <- contador + 1
  # Se imprime tirada
  cat("Tirada:", dado1, "-", dado2, "\n")
  # Condición de salida: ambos dados iguales
  if (dado1 == dado2) break
}
cat("Total de tiradas:", contador, "\n")
```

Consola de salida de R

```
> # Se inicializa contador de tiradas
> contador <- 0
> # Contar tiradas de dos dados hasta que aparezcan valores repetidos
> repeat {
+   # dado1 y dado2 toman un valor aleatorio entre 1 y 6
+   dado1 <- round(runif(1,1,6));
```

```
+   dado2 <- round(runif(1,1,6));
+   # Se incrementa contador de tiradas
+   contador <- contador + 1;
+   # Se imprime tirada
+   cat("Tirada:", dado1, "-", dado2, "\n")
+   # Condición de salida: ambos dados iguales
+   if (dado1 == dado2) break
+ }
Tirada: 3 - 2
Tirada: 4 - 3
Tirada: 2 - 6
Tirada: 6 - 4
Tirada: 3 - 5
Tirada: 3 - 3
> cat ("Total de tiradas:", contador, "\n")
Total de tiradas: 6
```

7.2.4 Interrupción del flujo normal de un ciclo

El flujo normal de los ciclos se puede interrumpir básicamente por medio de tres instrucciones diferentes:

- `break`,
- `next`,
- `return` (esta instrucción se estudiará en la unidad de funciones).

La instrucción `break` interrumpe la ejecución de un ciclo, pasando el control a la instrucción que sigue al mismo.

Su uso se ejemplificó en el problema de la tirada de los dados. La instrucción:

```
if (dado1 == dado2) break
```

provoca que el ciclo sea interrumpido cuando el valor de los dados sea el mismo.

Como otro ejemplo del uso de la instrucción `break`, se presenta un ciclo que simula un procedimiento iterativo para encontrar un valor aproximado a otro ya prefijado anteriormente, para un valor de convergencia dado. Para no caer en ciclos infinitos, estos tipos de

procedimientos limitan el número de iteraciones a un valor suficientemente grande, lo que aquí se hace mediante una instrucción `for` limitada a 1000 repeticiones:

Script de entrada en R

```
# Ejemplo de uso de break
# Se usará un generador de números aleatorios
set.seed(140)      # El argumento puede ser cualquier número
aprox <- 0.003     # Valor límite para la salida del ciclo
y_ini <- 2.7       # Valor inicial de y
for (i in 1:1000) {
  y <- y_ini + 0.008*rnorm(1)  # Nueva aproximación a y_ini
  print(y-y_ini)
  if (abs(y-y_ini) <= aprox)   # Se cumple condición de salida
    break                      # Se sale del ciclo usando break
  y_ini <- y
}
cat("Valor aproximado de la variable: ", y, "\n")
```

Consola de salida de R

```
> # Ejemplo de uso de break
> # Se usará un generador de números aleatorios
> set.seed(140)      # El argumento puede ser cualquier número
> aprox <- 0.003     # Valor límite para la salida del ciclo
> y_ini <- 2.7       # Valor inicial de y
> for (i in 1:1000) {
+   y <- y_ini + 0.008*rnorm(1)  # Nueva aproximación a y_ini
+   print(y-y_ini)
+   if (abs(y-y_ini) <= aprox)   # Se cumple condición de salida
+     break                      # Se sale del ciclo usando break
+   y_ini <- y
+ }
[1] 0.01542321
[1] 0.005853768
[1] 0.007636941
[1] 0.00585024
[1] 0.01659938
[1] 0.01794543
[1] -0.004874965
[1] 0.001393772
> cat("Valor aproximado de la variable: ", y, "\n")
Valor aproximado de la variable: 2.765828
```

En este ejemplo, el objetivo se ha alcanzado en 8 iteraciones.

La instrucción **`next`** permite terminar la iteración en curso de un ciclo, pasando a la siguiente iteración; o sea, impide la ejecución de las instrucciones siguientes y regresa al principio del ciclo para ejecutar la siguiente iteración.

El siguiente ejemplo ilustra esta operación, en un algoritmo inusual (y poco legible) de imprimir los valores del 1 al 10, excepto el 5.:

Script de entrada en R

```
# Imprime valores del 1 al 10 excepto 5
for (n in 1:10) { # itera 10 veces
  if (n == 5) { # si n es 5,
    next  # pasa a la nueva iteración
  }
  cat(n, "  ")
}
```

Consola de salida de R

```
> # Imprime valores del 1 al 10 excepto 5
> for (n in 1:10) { # itera 10 veces
+   if (n == 5) { # si n es 5,
+     next  # pasa a la nueva iteración
+   }
+   cat(n, "  ")
+ }
1   2   3   4   6   7   8   9   10
```

7.2.5 Ventajas de la vectorización

Una implementación imperativa de la suma de dos vectores, por ejemplo, `x=(3,5,7)` e `y=(4,4,8)`, exige inicializar el nuevo vector z y luego calcular la suma de cada componente de los vectores sumandos:

```
z <- c()

x <- c(3,5,7)

y <- c(4,4,8)

for (i in 1:length(x))

   z[i] <- x[i] + y[i]
```

En R, gracias a la vectorización, basta escribir:

```
z <- x + y
```

con una evidente ganancia expresiva. Además, en virtud de que la suma es una operación vectorizada, resulta un cálculo más eficiente en tiempo que la versión con ciclo.

El hecho de que en R muchas operaciones sean vectorizadas, disminuye la necesidad de emplear ciclos. En otros lenguajes hallar la suma de un grupo de valores requiere de un ciclo de suma, en R basta con:

```
sum(x)
```

donde x es el vector con los valores a sumar.

Otro ejemplo, `x <- x + 1` es más rápido que:

```
n <- length(x)
for (i in seq(n))
   x[i] <- x[i] + 1
```

Siempre que sea posible, es preferible evitar el uso de instrucciones de ciclos, por el gasto de tiempo y de ejecución que ellas requieren. En su lugar se deben usar operaciones vectorizadas, que aceleran en gran medida la ejecución.

Esto ocurre, por ejemplo, cuando se desea aplicar una función a cada elemento de un vector o si el resultado de cada iteración no depende de la iteración precedente. La ventaja de usar operaciones vectorizadas está en que ellas llaman funciones escritas en C o FORTRAN que también usan lazos, pero al ser lenguajes compilados y no interpretados, las iteraciones se realizan en un tiempo muy reducido.

7.3 Otras expresiones condicionales

Además de la instrucción `if - else` estudiada en la sección 7.1 existen otras expresiones alternativas o condicionales las cuales se estudiarán a continuación.

7.3.1 Instrucción if vectorizada: `ifelse`

Un método alternativo para ramificar un cálculo es la función **`ifelse()`**, que acepta un vector lógico como condición y devuelve un vector. Para cada elemento del vector de condición, si el valor es TRUE, entonces se escoge el correspondiente valor del segundo argumento y si es FALSE, se escoge el correspondiente elemento del tercer argumento. Por ejemplo:

```
ifelse(c(TRUE, FALSE, FALSE), c(1, 2, 3), c(4, 5, 6))
```

devuelve:

```
 [1] 1 5 6
```

Al ejecutar:

```
x <- c(3:-2)
sqrt(ifelse(x >= 0, x, NA))
```

se obtiene como resultado:

```
1.732051  1.414214  1.000000  0.000000    NA   NA.
```

7.3.2 Instrucción `switch`

A diferencia de `if`, que trata con condiciones verdaderas o falsas, la instrucción `switch` emplea un número o una cadena (denominado **selector**) para escoger una rama de ejecución, de acuerdo con el valor del selector. Su sintaxis es la siguiente:

```
switch(EXPR, ...)
```

Se evalúa la expresión `EXPR` y en dependencia de su valor, devuelve el argumento de `...` que está ubicado en esa posición.

Ejemplo:

Sea n un valor entero. Las siguientes expresiones devuelven el n-ésimo argumento de la lista que sigue al selector:

```
> switch(1, "x", "y")
[1] "x"
> switch(2, "x", "y")
[1] "y"
```

La expresión `switch(2, "x", "y")` es equivalente a:

```
n <- 2
switch(n, "x", "y")
```

Si el valor del selector está fuera de rango y por tanto no concuerda con ningún argumento, la función `switch()` retorna NULL:

```
> print(switch(3, "x", "y"))
NULL
```

La función `switch()` se comporta algo diferente si el selector es una cadena, retornando el valor del primer argumento cuyo nombre concuerde con el selector:

```
> switch("a", a = 1, b = 2)
[1] 1
> switch("b", a = 1, b = 2)
[1] 2
```

Si el nombre de ningún argumento concuerda con el selector, se retorna el valor NULL:

```
> print(switch("c", a = 1, b = 2))
NULL
```

Para cubrir todas las posibilidades se agrega un último argumento sin nombre que captura los casos fuera de rango:

```
> switch("c", a = 1, b = 2, 3)
[1] 3
```

Un ejemplo clásico de empleo de `switch()` es el siguiente: dado un mes y un año, determinar cuántos días tiene ese mes.

Está claro que:

- para los meses de enero, marzo, mayo, julio, agosto, octubre y diciembre, la cantidad de días es 31;
- para los meses de abril, junio, septiembre y noviembre es 30;
- en el caso de febrero, si el año es bisiesto[13] entonces la cantidad de días es 29, si no es bisiesto es 28.

Si analizamos el algoritmo de solución, este se basa en tratar tres alternativas excluyentes (que dependen de cada grupo de meses) y que el selector es por tanto el mes.

A continuación, se propone una solución (probada para febrero de 2020):

Script de entrada en R

```
mes <- 2
anho <- 2020
# Si bisiesto
if (mes==2 &&
   ((anho %% 400 == 0)||((anho %% 100 != 0) && (anho %% 4 == 0)))){
    cdias <- 29
```

[13] Año bisiesto: si es divisible entre 4, excepto si es el último año de cada siglo (termina en 00), en cuyo caso también ha de ser divisible entre 400.

```
} else { # Febrero de año no bisiesto u otro mes
    cdias <- switch(mes,31,28,31,30,31,30,31,31,30,31,30,31)
}
cat("El mes", mes, "del año", anho, "tiene", cdias, "días", "\n")
```

Consola de salida de R

```
> mes <- 2
> anho <- 2020
> # Si bisiesto
> if (mes==2 &&
+    ((anho %% 400 == 0)||((anho %% 100 != 0) && (anho %% 4 == 0)))){
+     cdias <- 29
+ } else { # Febrero de año no bisiesto u otro mes
+     cdias <- switch(mes,31,28,31,30,31,30,31,31,30,31,30,31)
+ }
> cat("El mes", mes, "del año", anho, "tiene", cdias, "días", "\n")
El mes 2 del año 2020 tiene 29 días
```

Ejercicios

1. ¿Qué valores tomarán las variables l, k, m como resultado de ejecutar la instrucción condicional:

```
if ((k<=l) && (k<=m)) {
    k <- k + 5
} else {
  if (l <= m) {
      l <- l + 5
  } else {
      m <- m + 5
  }
}
```

con los siguientes valores iniciales para las variables?:

a) k=3, l=3, m=4.

b) k=5, l=6, m=3.

c) k=3, l=2, m=2.

2. Escriba un programa para resolver los siguientes problemas. Use la vectorización siempre que sea posible.

   a) Averiguar si de dos números uno es divisor de otro e imprimir el resultado.

   b) Dados dos puntos en el plano cartesiano (x_1, y_1) y (x_2, y_2) imprimir las coordenadas de aquel que está más cerca del origen. Si los dos puntos están a la misma distancia, entonces imprimir las coordenadas de ambos.

   c) Dados los vértices de un triángulo $(x_1, y_1), (x_2, y_2), (x_3, y_3)$ determinar si ese triángulo es isósceles, equilátero o escaleno.

   d) Determinar el precio de un pasaje de ida y vuelta en ferrocarril, conociendo la distancia a recorrer y el número de días de estancia en el destino. Si este es mayor que 7 y la distancia es superior a 800 km, el pasaje tiene una reducción del 30 %. El precio por km es $0.80.

   e) Leer un carácter y ver si está antes o después de la letra M.

   f) Obtener todas las raíces de la ecuación $ax^2 + bx + c = 0$. En el caso de raíces complejas, escribirlas en la forma $a \pm bi$.

   g) Calcular la edad actual de una persona, dadas su fecha de nacimiento y la fecha actual, en ambos casos en la forma día, mes y año.

   h) Dadas las coordenadas de cada vértice de un triángulo y las de un cierto punto, determinar si este punto cae dentro, fuera o sobre el triángulo.

   i) Los empleados de una fábrica trabajan en dos turnos, diurno y nocturno. Se desea calcular el salario de un día de trabajo atendiendo a:

- La tarifa horaria diurna es $5.00.
- La tarifa horaria nocturna es $8.00.
- Caso de ser domingo, la tarifa horaria se incrementa en $2.00, si es diurna, y $3.00 en el turno nocturno.

Un día de trabajo tiene 8 horas, todas en uno solo de los turnos.

j) Crear un vector x siguiendo la siguiente fórmula para construir sus componentes:

$$x_i = \frac{(-1)^{i+1}}{2i-1}$$

donde i = 1, 2, …, 100. Obtenga luego la suma de los elementos de x.

Unidad 8. Funciones definidas por el programador

Un método para solucionar problemas es dividirlos en subproblemas, o sea, problemas más sencillos, y repetir el propio proceso hasta llegar a problemas más fáciles de resolver. Por otro lado, podemos componer la solución de un problema a partir de componentes ya existentes, que podemos reutilizar. En ambos casos, tales subproblemas o componentes corresponden a unidades de programación más especializadas, que son los subprogramas.

Un subprograma es una entidad compuesta de declaraciones de datos e instrucciones, conformando una unidad de cómputo, que puede ser invocado (llamado) desde diferentes partes del programa, constituyendo una forma primaria de reuso de código. Cuando un subprograma es llamado se le pasa un conjunto de argumentos que son usados para modificar cada ejecución del subprograma, enviar datos al subprograma o recibir los resultados del cómputo realizado.

Los lenguajes de programación tienen la capacidad de crear subprogramas. En R, un subprograma se denomina **función**. Ya se ha visto que la mayor parte del trabajo en R se realiza usando funciones que aporta su sistema base o diferentes paquetes con sus respectivos argumentos entre paréntesis. Ahora en esta unidad se estudiarán facilidades que brinda el lenguaje al usuario para escribir sus propias funciones, lo cual permite un uso flexible, eficiente y racional del R.

8.1 Definición de una función

La definición de una función se muestra sintéticamente como:

```
nombre_funcion <- function(arg1, arg2, ...)
{ cuerpo_función
}
```

Para **crear o definir una función**, se emplea la directiva "**`function`**".

La definición de una función tiene dos partes:

1) El encabezado de la función, que es su primera línea, que da nombre a la función usando la palabra clave **`function`**, asociándola en general con un símbolo que da nombre a la función (**`nombre_funcion`**) mediante una operación de asignación y que define sus argumentos formales **`arg1, arg2, ...`**.

   ```
   nombre_funcion <- function(arg1, arg2, ...)
   ```

2) El cuerpo de la función (**`cuerpo_función`**).

   El cuerpo de la función está constituido por una o más expresiones (comandos u operaciones) válidas del lenguaje, escritas entre llaves y separadas entre sí por punto y coma o por nueva línea (esto último es lo usual). El valor calculado por la función se indica poniendo como último comando el nombre de la propia función o el comando **`return(nombre_funcion).`**

Se muestra a continuación la definición de una función que devuelve un vector formado por el cociente y producto de dos valores:

```
mivec <- function(x1, x2)
{ z1 <- x1 / x2
  z2 <- x1 * x2
  mivec <- c(z1, z2)
  mivec   # igual con return(mivec)
}
```

La llamada de una función se describe con la sintaxis:

```
nombre_funcion(act1, act2, ...), o
```

```
nombre_funcion(arg1=..., arg2=..., ...)
```

donde `act1, act2, ...` son los nombres que se dan a los argumentos actuales en disposición posicional, o `arg1, arg2, ...` son los nombres de los argumentos formales en disposición nominal, como se explica en el próximo epígrafe.

La **llamada** o **invocación** de una función provoca la ejecución de la misma, para los valores de sus argumentos actuales que aparecen en la llamada, que pueden precederse o no del nombre de los argumentos formales. El valor retornado por la función es el valor del cuerpo de la función, que generalmente se denota como una última expresión del cuerpo o por la función **`return()`**. Por ejemplo la llamada **`mivec(5,5)`** provocaría la salida en consola de:

```
[1] 1 25
```

En R las funciones son **entidades de primera clase**, lo que significa que son tratadas prácticamente igual que cualquier otro objeto de datos de R, fundamentalmente:

- se pueden pasar como argumentos a otras funciones,
- se pueden retornar como el valor de una función,
- se pueden definir en el interior de otra función,
- pueden ser asignadas a variables, etc.

Para consultar el código de una función en R, ya sea una función propia de R o bien una del usuario, basta con teclear su nombre en la línea de comandos (sin paréntesis ni ningún tipo de argumentos) y dar clic en la tecla Enter.

8.2 Argumentos formales y actuales

Cada vez que un subprograma es invocado, se establece una correspondencia entre los **argumentos o parámetros actuales** que aparecen en la llamada o invocación a una función y los **argumentos o parámetros formales** que aparecen en la definición de la función llamada.

Los argumentos formales y actuales sirven como vía de comunicación entre el subprograma invocado y otro que lo llama, permitiendo la transferencia de datos y resultados entre ellos.

R usa como mecanismo de transferencia el **mecanismo por valor**. Al parámetro formal, que actúa como un parámetro de entrada, se le asigna una nueva dirección de memoria que es inicializada con el valor del argumento actual, el cual puede ser una expresión (lo cual incluye también una variable o una constante). Como el parámetro formal y su correspondiente actual ocupan diferentes direcciones de memoria **cualquier modificación que sufra el valor del parámetro formal no afectará el valor del argumento actual.**

Al momento de ejecutarse la función, los argumentos formales se asocian con o toman sus valores de los argumentos actuales. Hay dos formas de establecer esta asociación:

- Posicional
- Por nombre (nominal)

La **asociación posicional** consiste en asociar argumentos actuales con formales mediante el orden o la posición en que aparecen en la definición. El ejemplo que se muestra a continuación, una función que calcula el área de un trapecio[14], hace uso de esta característica.

Script de entrada en R

```
area.trap <- function(a,c,h)
{ area <- ((a+c)/2)*h
  area
}
area.trap (2,5,4)
area.trap (2,4,5)
```

Consola de salida de R

```
> area.trap(2,5,4)
```

---

[14] El área **A** de un trapecio de bases **a** y **c,** y altura **h** es igual a: $A = \frac{a+c}{2} * h$.

```
[1] 14
> area.trap(2,4,5)
[1] 15
```

La definición de la función está dada por:

```
area.trap <- function(a,c,h)

{ area <- ((a+c)/2)*h

  area

}
```

El nombre de la función es **`area.trap`**. Sus argumentos formales son **`a`**, **`c`** y **`h`**. Note que la última expresión del cuerpo de la función devuelve el valor del área calculada.

En la primera invocación **`area.trap (2,5,4)`** los argumentos actuales son 2, 5 y 4. Cuando se ejecuta esa invocación, en la cual se está usando la asociación posicional entre argumentos actuales y formales, esos argumentos actuales se corresponden con los formales `a`, `c` y `h`. Con la segunda invocación pasa algo similar, lo que cambia el valor calculado, al invertir los valores actuales de `c` y `h`.

La correspondencia entre argumentos formales y actuales puede ocurrir por **asociación nominal**. En este caso se asignan nombres a los parámetros actuales que corresponden a los de los argumentos formales, y la correspondencia en la invocación se establece entre argumentos actuales y formales del mismo nombre, por ejemplo:

Script de entrada en R

```
area.trap(a=2, h=4, c=5)
```

Consola de salida de R

```
> area.trap(a=2, h=4, c=5)
[1] 14
```

Pueden mezclarse la asociación posicional y nominal. Cuando un argumento es asociado por nombre, es extraído virtualmente de la lista de argumentos y los restantes se asocian en el orden en que aparecen en la definición de la función.

Los argumentos de las funciones son evaluados de forma demorada (lazy evaluation), o sea, ellos son evaluados solo si son necesitados en el cuerpo de la función. Por ejemplo, la siguiente función f tiene dos argumentos, pero en su cuerpo solo se usa uno de ellos:

```
f <- function(a, b) {a^2}
```

La llamada `f(2)` no provoca error y devuelve el valor 4, pues el argumento actual `2` corresponde posicionalmente con el argumento formal `a`.

Sin embargo, sea la función f1:

```
f1<- function(a, b)
{  print(a)
   print(b)
}
```

Al llamarla con f(45) ocurre:

```
> f(45)
[1] 45
Error in print(b) : argument "b" is missing, with no default
>
```

Note que "45" se imprime antes de que el error sea lanzado. Esto es así, pues `b` no es evaluado hasta después de imprimir `a`. Una vez que la función trata de evaluar `print(b)` muestra el error.

8.3 Argumentos formales con valor por defecto u omisión

Las funciones en R pueden tener argumentos formales por defecto u omisión, o sea, valores que deben asignarse automáticamente a esos argumentos formales en caso que no sean especificados para ellos valores de argumentos actuales en la llamada de la función. Los argumentos por omisión se especifican cuando se declara la función, especificando algunos parámetros con un valor dado.

En el siguiente ejemplo se presenta una versión de la función que calcula el área del trapecio, vista en la sección anterior, donde el argumento formal `h` toma por defecto el valor 4. Por ello, las dos llamadas a la función que se muestran a continuación devuelven igual resultado, pues en la llamada **`area.trap2(2,5)`** no se trasmite valor al argumento `h` y la asociación es posicional; por tanto automáticamente `h` toma el valor por omisión 4.

Script de entrada en R

```
area.trap2 <- function(a,c,h=4)
{ area <- ((a+c)/2)*h
  area
}
area.trap2(2,5,4)
area.trap2(2,5)
```

Consola de salida de R

```
> area.trap2 <- function(a,c,h=4)
+ { area <- ((a+c)/2)*h
+   area
+ }
> area.trap2(2,5,4)
[1] 14
> area.trap.v2(2,5)
[1] 14
```

Si se omite el valor de un argumento formal que no ha sido dado por omisión en la definición de la función, ocurre un error, como provoca la siguiente invocación:

```
  area.trap2(c=5,h=2)
```

la cual muestra como salida:

```
> area.trap2(c=5,h=2)

Error in area.trap2(c = 5, h = 2) :

argument "a" is missing, with no default
```

Lo que ocurrió es que el argumento `a` no es un argumento por defecto y en la invocación no se le trasmitió ningún valor.

8.4 Funciones `args()` y `formals()`

La función `args()` devuelve el encabezado de la función que se le pasa como argumento, permitiendo ver la lista de argumentos de la función tal como fue definida. Esta función le resulta útil a un usuario del lenguaje para revisar la lista de argumentos de cualquier función.

La función `formals()` devuelve una lista con cada uno de los argumentos formales y los valores asignados por omisión. Esta función le resulta más útil a un programador que desea manipular esta lista.

Script de entrada en R

```
args(area.trap)
args(area.trap2)
formals(area.trap2)
```

Consola de salida de R

```
> args(area.trap)
function (a, c, h)
NULL
> args(area.trap2)
function (a, c, h = 4)
NULL
> formals(area.trap2)
$a
$c
$h
[1] 4
```

8.5 Uso del argumento "…"

El argumento "**...**" indica un número variable de argumentos que por lo general son pasados a otras funciones. Se puede interpretar como "y el resto de los argumentos".

En el siguiente ejemplo se extiende la función `area.trap2` en la nueva función `Nueva_area.trap2` sin copiar la lista completa de los argumentos de la función original.

Script de entrada en R

```
# Argumento ...
Nueva_area.trap2 <- function(a=4, ...)
{ area.trap2(a,...)
}
Nueva_area.trap2(c=2)
Nueva_area.trap2(c=2, h=5)
```

Consola de salida de R

```
> # Argumento ...
> Nueva_area.trap2 <- function(a=4, ...)
+ { area.trap2(a,...)
+ }
> Nueva_area.trap2(c=2)
[1] 12
> Nueva_area.trap2(c=2, h=5)
[1] 15
```

Se observa que el resto de los argumentos de la función `area.trap2` (c y h) son pasados a la nueva función como si se hubiesen declarado en su definición.

Otro uso del argumento "**...**" es al principio de la lista de argumentos formales de funciones para las que no se conoce de antemano el número de argumentos que les serán enviados. Uno de estos casos es la función **`paste()`**, que sirve para construir una cadena de caracteres a partir de un número cualquiera de cadenas de caracteres que le pasen como argumentos. Los argumentos de esta función se pueden observar ejecutando **`args(paste)`**:

```
> args(paste)
function (..., sep = " ", collapse = NULL)
```

```
NULL
```

Puede comprobarse que:

- El argumento "**...**" está al inicio de la lista de argumentos formales de la función.
- La asociación con todos los argumentos que siguen al argumento especial "**...**", es ***nominal*** y de **ninguna manera *posicional***; esto es, deben ser explícitamente nombrados si su valor se requiere al momento de ser ejecutada la función.

Por ejemplo, la invocación de:

```
paste("uno", "dos", "tres", sep = "++")
```

devuelve una cadena de caracteres a partir de las tres cadenas de caracteres dadas, separadas por el símbolo ++.

```
> paste("uno", "dos", "tres", sep = "++")
[1] "uno++dos++tres"
```

8.6 Nombres de funciones como variables

Las funciones en R son objetos que pueden ser asignados a otros objetos (variables). Supongamos la siguiente función:

```
f1 <- function(x, y)
{   ifelse(x > y, x + y, x - y)
}
```

En la función precedente cada rama de la condicional conduce a expresiones diferentes que pueden resultar en valores diferentes. Para alcanzar la misma meta, podemos modificar la función anterior a la siguiente:

```
f2 <- function(x, y)
{ op <- ifelse(x > y, `+`, `-`)
```

```
  op(x, y)
}
```

En R, los operadores como + o – también son considerados como funciones. En dependencia del valor de la condición **`x > y`** se asigna a la variable op (que nombra a una función) una de esas dos funciones operadoras **`` `+` ``** o **`` `-` ``** y entonces **`op`** puede usarse como una función también. Note que + y – van entre apóstrofos invertidos (`), pues no son nombres permitidos en el lenguaje, pero al estar encerrados entre esos caracteres pueden usarse como nombres.

8.7 Funciones como argumentos de funciones

En R las funciones son objetos de orden superior, por lo que cualquier función puede tomar una función como argumento. Sean dos funciones con nombres suma y producto:

```
suma <- function(x, y, z)
{ x + y + z
}
producto <- function(x, y, z)
{ x * y * z
}
```

Ahora definamos otra función, capaz de combinar `x`, `y`, `z` en la forma especificada por un argumento, el cual a su vez es una función que llamaremos `f`, la cual toma tres argumentos:

```
combine <- function(f, x, y, z)
{   f(x, y, z)
}
```

Para hallar la suma o el producto de tres números, hacemos las llamadas:

```
combine(suma, 3, 4, 5)
```

```
combine(producto, 3, 4, 5)
```

que respectivamente dan los resultados en consola:

```
[1] 12
[1] 60
```

Como otro ejemplo algo más complejo, supongamos debemos calcular la suma $\sum_{x=m}^{n} f^2(x)$. Note que, para escribir una función capaz de lograr esa suma, es necesario pasarle como parámetro la función particular que se toma en el proceso de suma, además de los valores de `m` y `n`. Una posible solución se muestra a continuación:

```
Suma_Cuadrado <- function(f, m, n)
{   sum <- 0
    for (k in m:n)
    {  sum <- sum + f(k) * f(k)
    }
    return (sum)
}
```

Se desea calcular $\sum_{x=1}^{5} \left(\frac{1}{x}\right)^2$, como la función $g(x) = \frac{1}{x}$ no está definida en el lenguaje, se hace a través del script:

```
g <- function(x)
{  return (1.0 / x)
}
```

Entonces la suma cuadrática se ordena ejecutando la llamada:

```
a <- Suma_Cuadrado(g, 1, 5)
```

Para obtener y $\sum_{x=0}^{4} sen^2(x)$ se procede con la llamada:

```
b <- Suma_Cuadrado(sin, 0, 4)
```

pues la función seno está predefinida en el lenguaje (**`sin`**).

8.8 Asociación de símbolos con valores

Cada vez que se introduce un nuevo símbolo en alguna parte del código de un programa escrito en R, al ejecutarse, el lenguaje tiene que resolver de algún modo de qué manera se asocia ese símbolo con algún objeto de dato creado o por crearse dentro del programa. Básicamente hay dos maneras de hacer esto:

1. **Las expresiones de asignación**, que exigen que la expresión a la derecha esté bien definida. Sin embargo, observe el siguiente script:

   ```
   rm(list=ls())
   x <- y
   ```

   que al ejecutarse da como salida:

   ```
   > rm(list=ls())
   > x <- y
   Error: objeto 'y' no encontrado
   ```

   El error está dado porque la expresión de la derecha de la asignación (símbolo “y”) no está bien definida, pues el objeto `y` no ha sido creado. Si `y` hubiera estado bien definido, entonces la asignación hubiera creado un objeto `x` con el valor de `y`.

2. **La declaración de argumentos formales de alguna función**. Otra manera de introducir los símbolos correspondientes a nuevas variables es mediante la declaración de los argumentos formales en la definición de alguna función. Retomemos para explicar esto, el ejemplo de la primera función que hemos definido en esta unidad:

```
area.trap<-function (a, c, h) # Area del trapecio
{  area <- ((a + c)/2)*h
   area
}
```

En este ejemplo, el símbolo "`area.trap`", se ha introducido por la *asignación* (`<-`), y los símbolos "`a`", "`c`" y "`h`", se han introducido mediante su *declaración como argumentos formales de la función*, y cuyo tiempo de vida se enmarca en el código de la función. Pero, ¿dónde y cómo se organizan esas asociaciones entre símbolos y valores?

Las asociaciones entre símbolos y valores son guardadas en lo que se denomina **ambiente** (environment en inglés), conjunto de asociaciones entre nombres y objetos denotables que existen en ejecución en un punto y momento específico dado durante la ejecución del programa.

Los ambientes se organizan jerárquicamente: cada ambiente tiene un ambiente padre, excepto el denominado ambiente vacío; cada ambiente puede tener múltiples ambientes hijos.

El **ambiente de referencia** de una instrucción es la colección de alcances que han declarado variables que son visibles en dicha instrucción.

El lenguaje R permite declaraciones implícitas que introducen una asociación en el ambiente para un nombre la primera vez que es usado. El tipo del objeto denotado se deduce del contexto en que se usó por primera vez, de subsiguientes asignaciones o de la estructura sintáctica del nombre.

El ambiente de referencia asociado con una función se compone de:

- Ambiente local: Asociaciones para nombres declarados dentro de la función, más las asociaciones correspondientes a los argumentos formales.

- Ambiente no local: Asociaciones para nombres que son visibles dentro de la función, pero no son declaradas dentro de ella.
- Ambiente global: Formado por las asociaciones creadas para ser empleadas en cualquier función del programa.

La función **`search()`** permite conocer los ambientes activos cuando se ejecuta un programa en R.

Script de entrada en R

```
search()
```

Consola de salida de R

```
> search()
 [1] ".GlobalEnv"  "tools:rstudio"  "package:stats"
"package:graphics"
 [5] "package:grDevices"  "package:utils"  "package:datasets"
 [8] "package:methods" "Autoloads"  "package:base"
```

8.9 Reglas de alcance

Las reglas de alcance se refieren a la forma en que el lenguaje resuelve la asociación entre una variable o símbolo libre y su correspondiente valor.

Alcance o ***ámbito*** de un identificador es el segmento del texto del programa en el que el identificador es visible, tiene un determinado significado y el objeto de dato, por ejemplo, una variable, nombrado por él puede ser referenciado por su nombre en esas instrucciones.

Una variable es ***local*** a una función si en su cuerpo resulta definida (por ejemplo, porque recibe un valor). Dicho objeto tendrá validez en toda la función, incluyendo las funciones que se definan en su cuerpo, a menos que no sean redefinidas en los cuerpos de las funciones interiores. Los argumentos formales de una función se consideran **locales** a la misma.

Una ***variable libre*** o ***no local*** en una función es aquella que no se ha definido en el código de la misma, pero es visible dentro de ella; por tanto, su valor puede ser empleado al incluir su

nombre en una expresión, por ejemplo, en el lado derecho de una asignación, como se muestra en la figura 15.

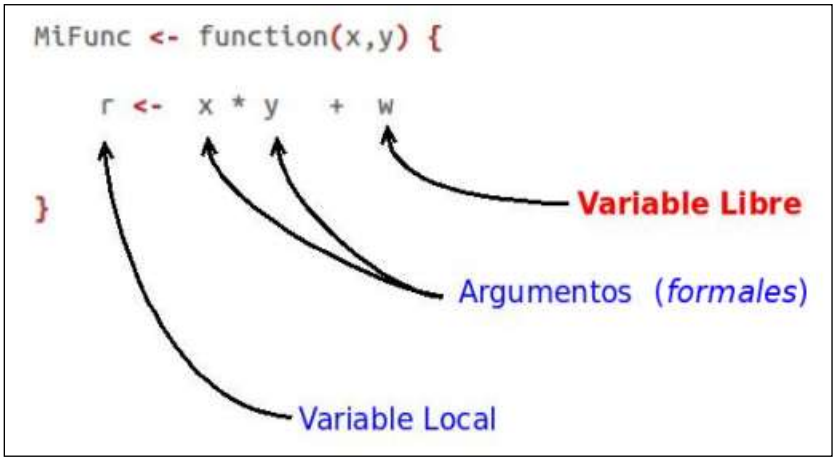


Figura 15. Tipos de símbolos en el interior de una función.
Tomado de Santana y Farfán (2014, p.98)

En la función `MiFunc` la variable `r` es local y la variable `w` es libre. Las variables `x e y` son también locales al ser argumentos formales de la función `MiFunc`.

Las variables **globales** constituyen una categoría especial de las variables libres, cuando su definición aparece fuera de todas las funciones definidas en el programa. Por ello son potencialmente visibles para todas estas funciones.

Las **reglas de alcance** determinan como se asocia un valor a una variable libre en una función. En particular, las reglas de alcance determinan como las referencias a variables declaradas fuera de su ambiente de declaración se asocian a su declaración y con ello a sus atributos.

En R los ambientes se organizan en una jerarquía; esto es, cada ambiente tiene un ambiente padre y a su vez puede tener cero o más hijos. El único ambiente que no tiene padre es el **ambiente vacío**, que se encuentra en lo más alto de la jerarquía. El lugar dónde R buscará el valor de una variable libre dependerá del lugar dónde se encuentre escrito el código de la función en el que se hace referencia a ella.

Cuando se usa una función en R se crea un ambiente local temporal, que queda incluido dentro del ambiente global, por lo que desde el ambiente local se puede tener acceso a cualquier objeto del ambiente global. Cuando la función termina su ejecución, el ambiente local es destruido y con ello todos los objetos que contiene. Cuando R detecta un nombre de objeto, lo busca en el ambiente local y si no lo encuentra, entonces en ambientes que sucesivamente lo contengan hasta el ambiente global.

Esta forma de organizar el alcance es conocida como alcance **estático** o **léxico**. Alcance estático en R significa que el valor de las variables libres se encuentra en el ambiente en que la función fue definida, lo que determina una de tres posibles situaciones para las variables libres (o no locales):

- Se considera indefinida fuera de la función donde esté declarada y por tanto su uso no es permitido.
- Se considera libre en una función f y por tanto pueden ser usadas, porque está declarada en otra función g que contiene a f.
- Es una variable global; por tanto, tiene validez para todo el programa, a partir del punto de declaración.

En el caso de la función `MiFunc()`, mostrada en la figura 15, observe la diferencia en el valor que tomará la variable libre `w`, al insertar el código de la función en distintas partes, en (A) en el ambiente global y en (B) en el ambiente de la función ff. Esto se muestra gráficamente en la figura 16.

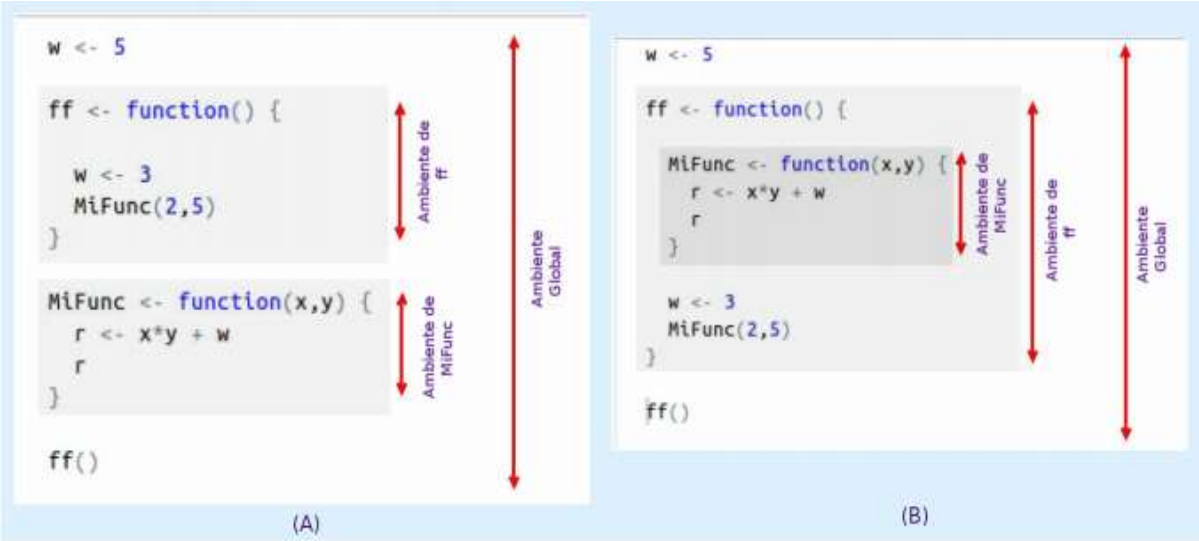


Figura 16. Función MiFunc en diferentes ambientes.
Tomado de Santana y Farfán (2014, p.99).

En el caso (A), el ambiente global tiene dos hijos, el correspondiente a la función `ff()` y el correspondiente a `MiFunc()`, mientras que en el caso (B), el ambiente global tiene un solo hijo que es el ambiente correspondiente a la función `ff()`, que a su vez tiene un hijo, el correspondiente a la función `MiFunc()`. El caso (B) también sirve para ilustrar que una función puede ser definida en el interior de otra función.

Para resolver el valor de `w` dentro de la función `MiFunc`, R busca primero en el ambiente de la propia función, si no encuentra el símbolo ahí, procede a buscar en el ambiente padre, y así lo sigue haciendo con todos los predecesores hasta encontrar una asociación. Si en este proceso se llega al ambiente vacío sin encontrar una asociación posible, el lenguaje emitirá un mensaje de error.

En el código que sigue se muestra el comportamiento de R en las situaciones A y B de la figura 16 y se añade una situación más, para ejemplificar lo que se ha explicado aquí.

Script de entrada en R

```
# Figura 16 A
w <- 5  # w en ambiente global
```

```
ff <- function()
{  w <- 3  # w local a ff
   MiFunc(2,5)
}
MiFunc <- function(x,y)
{  r <- x*y + w
   r
}
w
ff()
```

Consola de salida de R

```
> # Figura 16 A
> w <- 5  # w en ambiente global
> ff <- function()
+ { w <- 3  # w local a ff
+   MiFunc(2,5)
+ }
>
> MiFunc <- function(x,y)
+ { r <- x*y + w
+   r
+ }
> w
[1] 5
> ff()
[1] 15
```

En este caso, a pesar de que en el interior de la función `ff()` se asigna a w el valor 3, de acuerdo con la regla explicada, se busca primero el valor de `w` en el código de la función `MiFunc` y como no lo encuentra, lo busca entonces en el ambiente padre del ambiente de `MiFunc,` que en este caso es el ambiente global, en el cual `w` toma valor **5** y por tanto el valor que retorna es 2*5+**5**, o sea, 15.

Script de entrada en R

```
# Figura 16 B
w <- 5  # w en ambiente global
ff <- function()
{  # Definición anidada de MiFun dentro de ff
   MiFunc <- function(x,y)
   {  r <- x*y + w
      r
```

```
   }
   w <- 3  # w local a ff
   MiFunc(2,5)
}
w
ff()
```

Consola de salida de R

```
> # Figura 16 B
> w <- 5  # w en ambiente global
> ff <- function()
+ {  # Definición anidada de MiFun dentro de ff
+    MiFunc <- function(x,y)
+    {  r <- x*y + w
+       r
+    }
+    w <- 3   # w local a ff
+    MiFunc(2,5)
+ }
> w
[1] 5
> ff()
[1] 13
```

En este segundo caso, se busca el valor de `w` en el código de la función `MiFunc` y como no lo encuentra, lo busca entonces en el ambiente padre del ambiente de `MiFunc` que en este caso es el ambiente de la función `ff` y como en este ambiente `w`=**3**, le asigna este valor a `w` y devuelve 2*5+**3**, o sea, el valor 13.

Si eliminamos del script anterior la segunda asignación a `w` (**`w <- 3`**) puede comprobarse que la llamada `ff()` devuelve valor 15. Como en los casos anteriores, se busca el valor de w en el código de `MiFunc` y como no lo encuentra, lo busca entonces en su ambiente padre, que es el ambiente de la función `ff` y como tampoco lo encuentra, lo busca entonces en el ambiente padre de `ff` que es el ambiente global, en el cual w vale **5**. Por ello `ff()` devuelve el valor 2*5+**5**, o sea, 15.

Para forzar a una variable local a existir globalmente debe usarse el operador de asignación global **<<-**.

Script de entrada en R

```
# Asignación global <<-
sum <- function()
{ x <- 5
  z <- x + y  # Usa la y global
  return(z)
}
glob_assign <- function()
{ y <<- 3 # y pasa a global
  z <- sum()
  return(z)
}
glob_assign()
y
```

Consola de salida de R

```
> # Asignación global <<-
> sum <- function()
+ { x <- 5
+   z <- x + y  # Usa la y global
+   return(z)
+ }
> glob_assign <- function()
+ { y <<- 3 # y pasa a global
+   z <- sum()
+   return(z)
+ }
> glob_assign()
[1] 8
> y
[1] 3
```

El uso de asignación global y de variables globales dentro de una función se considera una **mala práctica de programación**. Una de las ideas principales que defiende el lenguaje R (por considerarse un lenguaje funcional) es que el resultado de evaluar una función debe ser solo dependiente de los valores de los argumentos de la función. Si a la función se le da iguales valores de argumentos, entonces devolverá iguales resultados.

8.10 Recursividad

Una función es **recursiva** si para ser ejecutada requiere realizar llamadas a sí misma.

La recursividad es un recurso muy conveniente, pues existen muchos problemas que tienen una naturaleza definida intrínsecamente de forma recursiva. En matemáticas es común encontrar definiciones de algunos conceptos en las que se aplican relaciones de recurrencia. Por ejemplo, el cálculo del factorial[15] de 5 es:

$$5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$$

Esto equivale a decir recurrentemente que: $5! = 5 \times 4!$.

De manera general, el factorial de un número podría ser descrito recursivamente como:

$$\boldsymbol{n!} = \begin{cases} \boldsymbol{1} & \boldsymbol{si\ n = 0} \\ \boldsymbol{n(n-1)!} & \boldsymbol{si\ n > 0} \end{cases}$$

La programación en R de este esquema recursivo sería:

```
# Cálculo recursivo del factorial de un número n
fact <- function(n)
{  if (n==0)
   {  return (1)  # Caso base, salida inmediata
   }
   return (n*fact(n-1))  # Caso general, invocación recursiva
}
```

Al ejecutar la invocación `fact(5)`, daría como resultado:

```
> fact(5)
[1] 120
```

[15] La operación de factorial aparece en muchas áreas de las matemáticas, particularmente en la combinatoria y el análisis matemático. En lo fundamental el factorial de *n* representa el número de formas distintas de ordenar *n* objetos distintos.

Sin embargo, la invocación `fact(-5)` daría error:

```
> fact(-5)
Error: C stack usage 19910000 is too close to the limit
```

Una posible solución para detectar el error es agregar otra rama alternativa que devuelva explícitamente un valor indicador del error o que dé un mensaje adecuado cuando el argumento de la función no sea un entero positivo:

```
# Cálculo recursivo del factorial de un número n
# Se detecta error de n<0
fact <- function(n)
{  if (n==0)
   {  return (1)  # Caso base, salida inmediata
   }
   if (n>0)
   {    return  (n*fact(n-1))    #  Caso  general,  invocación
recursiva
   }
   return (NULL)  # Señala error de entrada de valor de n
}
```

Ahora la invocación `fact(-5)` daría como salida el valor `NULL`:

```
> fact(-5)
NULL
```

También, en vez de retornar `NULL` se pudo emitir un mensaje descriptivo del error. El ambiente que invoca la función deberá tomar las medidas necesarias ante este error.

Todo esquema de algoritmo recursivo consta de dos partes:

1. Definición de ***casos base*** o ***de parada*** o ***solución trivial***. Son los casos del problema que se resuelven con un segmento de código sin recursividad. Siempre debe existir al menos un caso base. Constituyen las condiciones de salida o terminación del esquema recursivo, que dan un cálculo para la función que no requiere llamada a sí misma; en el ejemplo del factorial, n! = 1, si n=0.
2. Definición de los ***casos generales*** (***cláusula recursiva***): Es la definición del cuerpo recursivo, donde el cálculo del valor en un paso del algoritmo requiere del cálculo en pasos previos y de ahí la llamada a sí mismo. En el ejemplo, n! = n(n-1)!

   Los casos generales siempre deben avanzar hacia un caso base. Es decir, la llamada recursiva se hace a un subproblema más pequeño y, en última instancia, los casos generales alcanzaran un caso base.

El paso 1 es importante en el sentido que una mala selección de las condiciones de salida puede conducir a que el proceso recursivo no pare.

Otro ejemplo de algoritmo recursivo es el **algoritmo de Euclides**. El paso esencial que garantiza la validez del algoritmo consiste en mostrar que el **máximo común divisor** (mcd) de a y b, (a , b $\geq$ 0), es a si b es cero, en otro caso (si b > 0) es igual al mcd de b y el resto de la división de a por b, algoritmo que se muestra en el siguiente esquema:

$$mcd\ (a,b) = \begin{cases} a & si\ b = 0 \\ mcd(b, a\ mod\ b) & si\ b > 0 \end{cases}$$

En el esquema "**a mod b**" denota el resto de la división de **a** entre **b**. Una función en R que programa este algoritmo y corridas para cuatro pares de valores se brindan a continuación:

Script de entrada en R

```
# Cálculo recursivo del mcd(a,b). Se usa algoritmo de Euclides
mcd <- function(a,b)
{ if (b == 0)
```

```
   { return (a)
   }
   else
   { return (mcd(b, a %% b))
   }
}
# Algunas invocaciones
mcd(25, 5)
mcd(18, 6)
mcd(35, 16)
mcd(6, 18)
```

Consola de salida de R

```
> # Cálculo recursivo del mcd(a, b). Se usa algoritmo de Euclides
> mcd <- function(a, b)
+ { if (b == 0)
+   {  return (a)
+   }
+   else
+   { return (mcd(b, a %% b))
+   }
+ }
> # Algunas invocaciones
> mcd(25, 5)
[1] 5
> mcd(18, 6)
[1] 6
> mcd(35, 16)
[1] 1
> mcd(6, 18)
[1] 6
```

Si se desea mayor velocidad de ejecución, es preferible en muchos casos emplear programas iterativos equivalentes a los recursivos que resuelven el problema, principalmente si usan funciones vectorizadas. Por ejemplo, las siguientes funciones (equivalentes) para calcular el factorial de un número, economiza tiempo de ejecución, pues no son recursivas ni usan instruccion de ciclos:

Usando funciones vectorizadas

```
fact <- function(n)
```

```
{  if (n<0)
   { return (NULL)
   }
   if (n==0)
   { return (1)
   }
   return (prod(1:n))
}
```

equivalente con:

```
fact <- function(n)
{  if (n<0) NULL
   else if (n==0) 1
   else prod(1:n)
}
```

Con la ejecución de:

```
fact(-3)
fact(0)
fact(5)
```

se logra la salida:

```
> fact(-3)
NULL
> fact(0)
[1] 1
```

```
> fact(5)
[1] 120
```

Aún mejor sería la función:

```
fact <- function(n)
{ ifelse(n<0, NULL, ifelse(n==0, 1, prod(1:n)))
}
```

que emplea la función vectorizada **`ifelse`**.

Ejercicios

1. Escriba una función que realice cada una de las operaciones siguientes:
   a) La suma de 2 números complejos.
   b) La multiplicación de 2 números complejos.
   c) La división de 2 números complejos.
2. Escriba una función que permita determinar si dadas las coordenadas de un punto P(x,y) este pertenece o no a una circunferencia de centro (O_x, O_y) y radio r.
3. Construya un dataframe que contenga de cada estudiante de un grupo su nombre, año que cursa y notas en las asignaturas de Matemática, Español e Historia. Luego escriba una función que recibe este dataframe como argumento y retorna el índice promedio de cada estudiante.
4. Escriba una función para evaluar, dada la indeterminada x, polinomios de la forma:

$$P(x) = c_n x^{n-1} + c_{n-1} x^{n-2} + \cdots + c_2 x + c_1.$$

   La función debe recibir a x y al vector de coeficientes del polinomio como argumentos y retornar el valor del polinomio en la indeterminada x.

5. Para valores grandes de n, la evaluación de un polinomio en x puede ser más eficiente usando la regla de Horner:

   (a) Hacer $a_n = c_n$.

   (b) Para $i = n-1, n-2, \ldots, 1$ hacer $a_i = a_{i+1}x + c_i$.

   (c) Retornar a_1. (Es el valor calculado de P(x).)

   Escriba una función en R que recibe a x y al vector de coeficientes del polinomio como argumentos y retorna el valor del polinomio en la indeterminada x. Asegure que la función retorna un vector apropiado si x es un vector de cualquier longitud.

6. Supongamos que las direcciones de correo de los estudiantes de la Universidad de Oriente tienen la siguiente estructura:

   nombre.apellido@estudiantes.uo.edu.cu

   donde **nombre** es el primer nombre y **apellido** el primer apellido del estudiante, separados por un punto. Escriba una función que recibe una dirección de correo como parámetro y retorne un data frame que contiene tres variables para el nombre, el apellido y el correo del estudiante. Pruebe la función con varias direcciones de correos válidas.

7. Escriba una función que dado como argumento un valor entero entre 1 y 100 ejecute un lazo que genere un número aleatorio entre 1 y 100 y lo compare con el valor del argumento. El ciclo termina cuando se genere un valor igual al del argumento. La función devolverá como resultado la cantidad de iteraciones necesarias para obtener el valor del argumento. Pruebe la función llamándola con varios valores aleatorios enteros entre 1 y 100.

Unidad 9. Funciones de transformación y agregación de datos

Una función de orden superior (funciones como objetos de primera clase) es una función que toma como entrada (argumento) una función o que retorna como valor una función, lo cual es característico del lenguaje R. En la presente unidad se presentan diversos funcionales, funciones que toman como argumento una función y devuelven algún objeto (no función), que son la base para trabajar con un conjunto de funciones de transformación, clasificación y agregación de datos.

9.1 Funciones Map(), Reduce(), Filter(), Find(), Position() y Negate()

`Map(f, x, ...)`

La función **`Map()`** aplica la función **`f`** dada a cada uno de los valores dentro de un objeto x:

`Map(f, x, ...)`, donde **`f`** es una función y **`...`** denota otros argumentos.

Por ejemplo,

```
palabras <- list("este ", "es ", "un ", "ejemplo")
Map(toupper, palabras)
```

daría la salida en consola:

```
> palabras <- list("este ", "es ", "un ", "ejemplo")
> Map(toupper, palabras)
[[1]]
[1] "ESTE "
[[2]]
[1] "ES "
[[3]]
[1] "UN "
```

```
[[4]]

[1] "EJEMPLO"
```

`Reduce(f, x, init, right = FALSE, accumulate = FALSE)`

La función **`Reduce()`** reduce una lista x a un valor simple o vector aplicando recursivamente una función binaria (de dos argumentos), combinando los valores de la lista. Sus parámetros son (señalados sus valores por defecto cuando corresponde):

- **`f`**: una función binaria de 2 argumentos. El primer valor que se pasa a **`f`** es el valor acumulado y el segundo es el próximo valor a acumular. **`f`** puede ser un operador del lenguaje encerrado entre comillas dobles (**"**), apóstrofo (‘) o apóstrofo invertido (`).
- **`x`**: una lista o vector atómico.
- **`init`**: si se suministra, será usado como valor inicial para comenzar la acumulación, en vez de tomar **`x[[1]]`**. Resulta útil para asegurar que la función **`Reduce`** retorne un valor adecuado cuando la lista **`x`** está vacía, pues si no se suministra y **`x`** está vacía, entonces ocurre un error.
- **`right`**: un valor lógico indicando si se procede a acumular de izquierda a derecha (por defecto) o de derecha a izquierda.
- **`accumulate`**: un valor lógico indicando si las sucesivas combinaciones producidas por `Reduce()` deben ser mostradas o no. Por defecto toma el valor FALSE (solo se muestra la última combinación).

Esta función combina de entrada los dos primeros elementos, luego combina el resultado obtenido con el tercer elemento y así sucesivamente. Así, la llamada `Reduce(f, 1:3)` es equivalente a f(f(1, 2), 3).

Como ejemplo, se construye una función alternativa **miprod()** a la función predefinida **prod()**, usando **Reduce()**. Se ejemplifica también como obtener los productos parciales (**acummulate=T**).

Script de entrada en R

```
# Se define un vector p
p <- c(2, 3, 5)
# Se declara una función que acumula productos
miProd <- function(v) Reduce("*", v)
# Producto de elementos del vector usando "prod"
prod(p)
# Producto de elementos del vector usando "miprod"
miProd(p)
# Función que muestra las acumulaciones parciales de productos
miProdAcum <- function(v) Reduce("*", v, accumulate=T)
# Producto parciales de elementos del vector
miProdAcum(p)
```

Consola de salida de R

```
> # Se define un vector p
> p <- c(2, 3, 5)
> # Se declara una función que acumula productos
> miProd <- function(v) Reduce("*", v)
> # Producto de elementos del vector usando "prod"
> prod(p)
[1] 30
> # Producto de elementos del vector usando "miprod"
> miProd(p)
[1] 30
> # Función que muestra las acumulaciones parciales de productos
> miProdAcum <- function(v) Reduce("*", v, accumulate=T)
> # Producto parciales de elementos del vector
> miProdAcum(p)
[1]  2  6 30
```

La aplicación de la función **Reduce()** no está limitada a operadores, la operación puede estar definida por cualquier función de dos argumentos, y, opcionalmente, puede además devolver el arreglo de resultados parciales, cada vez que se aplica la operación, como en el siguiente ejemplo:

Script de entrada en R

```
# mif: Función a aplicar a los elementos del vector
mif <- function(a, b)
{  return(a * b/(a + b))
}
# Vector w
w <- c(-2,3,4)
Reduce(mif, w)
```

Consola de salida de R

```
> # mif: Función a aplicar a los elementos del vector
> mif <- function(a, b)
+ {  return(a * b/(a + b))
+ }
> # Vector w
> w <- c(-2,3,4)
Reduce(mif, v)
 [1] 12
```

¿Cómo calculó estos valores?

- **-2** es el primer valor de w.
- Aplica la función **mif(-2,3)** lo cual devuelve $\frac{-2*3}{-2+3} = \frac{-6}{1} = -6$
- Aplica la función **mif(-6,4)** lo cual devuelve $\frac{-6*4}{-6+4} = \frac{-24}{-2} = 12$

Filter(p,x)

Dado una función predicado **p** y una lista x de valores, la función **Filter()** retorna una lista filtrada con aquellos valores para los cuales el predicado unario **p** toma valor TRUE. Por ejemplo, la ejecución del script **Filter(is.character, list(1,"a",2,"b",3,"c"))** genera la salida (una lista):

```
[[1]]

[1] "a"

[[2]]

[1] "b"
```

```
[[3]]
[1] "c"
```

`Find(p, x, right = FALSE, nomatch = NULL)`

Retorna el valor del primer elemento en la lista o vector **`x`** que cumple **`p`**(x)=TRUE, donde **`p`** debe haber sido definida como una función lógica unaria. Los otros parámetros son (señalados sus valores por defecto):

- **`right`**: valor lógico que indica si se recorre la lista de izquierda a derecha (por defecto) o de derecha a izquierda.
- **`nomatch`**: valor a retornar si ningún elemento de la lista satisface el predicado **`f`** (por defecto es `NULL`).

Ejemplo:

Para la llamada `Find(is.character, list(1,"a",2,"b",3,"c"))` el valor retornado es "a" y para `Find(is.character,list(1,"a",2,"b",3,"c"), right=TRUE)` el valor retornado es "c".

Si deseáramos saber cuál es el primer valor par de un vector, basta escribir una función que determine si un número es par y luego usar **`Find()`** con esa función como argumento:

```
# Otro ejemplo de uso de find
es_par <- function(x) {
   x %% 2 == 0
}
Find(es_par, c(1,2,3,4,5))
```

La salida sería:

```
[1] 2
```

`Position(p, x, right = FALSE, nomatch = NA_integer_)`

Retorna la posición del primer elemento en la lista x que cumple **p**(x)=TRUE, donde **p** debe haber sido definida antes como una función lógica unaria. El parámetro **`nomatch`** denota el valor a retornar si ningún elemento de la lista satisface el predicado **p**.

Ejemplo:

Para la llamada `Position(is.character, list(1,"a",2,"b",3,"c"))` el valor retornado es 2; para `Position(is.character,list(1,"a",2,"b",3,"c"), right=TRUE)` es 6 y para `Position(is.character, list(1,2,3))` es NA.

`Negate(p)`

Invierte el valor lógico de una función predicado **p**. A modo de ejemplo, se define una función predicado `is.noncharacter` que niega a la función `is.character,` o sea, devuelve `TRUE` si analiza un valor que no es una letra y `FALSE` si lo es. Con ello, al ejecutar el script:

```
is.noncharacter <- Negate(is.character)
is.noncharacter("a")
```

daría como respuesta:

```
[1] FALSE
```

9.2 Las funciones de la familia apply()

R tiene predefinidas varias funciones que admiten como parámetros otras funciones, de las cuales son muy conocidas las de la familia de funciones **`apply`**. Algunas funciones de esta familia y sus sintaxis de invocación son:

`lapply(x, f, ...)`

- **`x:`** es una lista o vector.

- **f:** es una función a aplicar a cada elemento en x.

Devuelve una lista de igual longitud que x, en que cada elemento es el resultado de aplicar **f** al correspondiente elemento de x.

sapply(x, f, ..., simplify = TRUE, USE.NAMES = TRUE)

Es similar a **lapply()**, llevando el resultado de salida a un vector o una matriz, siempre que sea posible.

vapply(x, f, FUN.VALUE, ..., USE.NAMES = TRUE)

Es una variante de **sapply()** en la cual debe especificarse el tipo de objeto de salida.

mapply(f, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)

Es similar a **lapply()**, pero pueden ser pasados varios vectores como entrada de la función especificada. La salida puede simplificarse como en **sapply().**

donde:

- **x**: vector (atómico o lista) o un objeto de tipo expression.
- **f**: función a aplicar a cada elemento en **x**. Si la función corresponde a un operador, como +, %*%, ese operador debe ir entre comillas dobles (“), apóstrofos (‘) o apóstrofos invertidos (``).
- **...**: argumentos opcionales. En **mapply** son argumentos a ser vectorizados.
- **simplify**: valor lógico o cadena de caracteres. Indica si el resultado debe simplificarse a un vector, matriz o arreglo, si es posible.
- **USE.NAMES**: valor lógico. Si es TRUE y x es de tipo character, se usa x como nombre del resultado, a menos que ya tenga nombre.
- **MoreArgs**: una lista de otros argumentos.

Se especifica en la tabla 9 como son las entradas y salidas para cada una de estas funciones.

Tabla 9. Entrada y salida para las funciones de la familia apply()

Función	Entrada	Salida
apply()	Matriz o array o data frame	Vector o array o lista
lapply()	Lista o vector	Lista
sapply()	Lista o vector	Vector o matriz o lista
vapply()	Lista o vector	Vector o matriz o lista
mapply()	Listas y/o vectores	Vector o matriz o lista

Ejemplo:

Se tiene una lista o un data frame, exclusivamente de columnas numéricas y se quiere conocer el promedio de cada una de estas columnas. En este caso, la función **`sapply()`** permite aplicar una operación o una función a cada uno de los elementos de la lista o data frame, dado como argumento.

Script de entrada en R

```
# Creando un data frame de tres columnas numéricas
misDatos <- data.frame(uno = runif(5, 10.5, 40.3),
              dos = runif(5), tres = runif(5, 155, 890))
misDatos
# Aplicando a cada columna la función mean (promedio)
# Como simplify=TRUE devuelve un vector de medias por columnas
as <- sapply(misDatos, mean, simplify=TRUE)
as
class(as)
# Ahora simplify=FALSE, entonces devuelve una lista de medias
al <- sapply(misDatos, mean, simplify=FALSE)
al
class(al)
```

Consola de salida de R

```
> # Creando un data frame de tres columnas numéricas
> misDatos <- data.frame(uno = runif(5, 10.5, 40.3),
+               dos = runif(5), tres = runif(5, 155, 890))
> misDatos
       uno       dos     tres
1 17.08258 0.5332929 212.4407
2 24.36229 0.2889515 403.9519
3 26.57805 0.7889478 782.0941
```

```
4 29.44498 0.7244701 736.5005
5 36.51389 0.7386451 512.2286
> # Aplicando a cada columna la función mean (promedio)
> # Como simplify=TRUE devuelve un vector de medias por columnas
> as <- sapply(misDatos, mean, simplify=TRUE)
> as
        uno         dos        tres
 26.7963599   0.6148615 529.4431652
> class(as)
[1] "numeric"
> # Ahora simplify=FALSE, entonces devuelve una lista de medias
> al <- sapply(misDatos, mean, simplify=FALSE)
> al
$`uno`
[1] 26.79636

$dos
[1] 0.6148615

$tres
[1] 529.4432

> class(al)
[1] "list"
```

El argumento opcional **simplify**, especificado en la primera llamada de la función como TRUE, obliga a que el resultado, si se puede, sea devuelto como un vector, con un elemento correspondiente a cada una de las columnas en este caso, de otra manera, el resultado es devuelto como una lista, lo cual se muestra en la segunda llamada a la función.

El resultado obtenido con la función **lapply** es más o menos similar a **sapply**, pero la salida siempre es una lista:

Script de entrada en R

```
bl<-lapply(misDatos, mean)
bl
class(bl)
```

Consola de salida de R

```
> bl<-lapply(misDatos, mean)
> bl
$uno
```

```
[1] 26.0963
$dos
[1] 0.5102836
$tres
[1] 536.6521
> class(bl)
[1] "list"
```

A continuación, se muestran dos ejemplos que usan **mapply**. Primero aparece un pequeño script y luego el resultado calculado.

```
# 3 es reciclado por mapply (obteniendo un vector de 5 valores 3)

mapply(prod, 1:5, 10:6, 3)
```

Salida:

```
   [1] 30 54 72 84 90
```

```
mapply(rep, times = 1:4, MoreArgs = list(c(10,12)))
```

Salida:

```
   [[1]]
   [1] 10 12
   [[2]]
   [1] 10 12 10 12
   [[3]]
   [1] 10 12 10 12 10 12
   [[4]]
   [1] 10 12 10 12 10 12 10 12
```

En el anterior ejemplo la función repite los elementos de la lista, primero una vez, luego 2 y así sucesivamente hasta 4 veces.

En el siguiente ejemplo se construyen dos listas, cuyos primeros dos elementos son matrices y el tercero es un vector:

Script de entrada en R

```
# Se construyen dos listas
lista1 <- list(A=matrix(1:16, 4), B=matrix(1:16, 2), C=1:5)
lista1
```

```
lista2 <- list(A=matrix(1:16, 4), B=matrix(1:16, 8), C=15:1)
lista2
# Probar elemento a elemento si son idénticos
mapply(identical, lista1, lista2)
```

Consola de salida de R

```
> # Se construyen dos listas
> lista1 <- list(A=matrix(1:16, 4), B=matrix(1:16, 2), C=1:5)
> lista1
$`A`
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

$B
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]    1    3    5    7    9   11   13   15
[2,]    2    4    6    8   10   12   14   16

$C
[1] 1 2 3 4 5

> lista2 <- list(A=matrix(1:16, 4), B=matrix(1:16, 8), C=15:1)
> lista2
$`A`
     [,1] [,2] [,3] [,4]
[1,]    1    5    9   13
[2,]    2    6   10   14
[3,]    3    7   11   15
[4,]    4    8   12   16

$B
     [,1] [,2]
[1,]    1    9
[2,]    2   10
[3,]    3   11
[4,]    4   12
[5,]    5   13
[6,]    6   14
[7,]    7   15
[8,]    8   16

$C
 [1] 15 14 13 12 11 10  9  8  7  6  5  4  3  2  1
```

```
> # Probar elemento a elemento si son idénticos
> mapply(identical, lista1, lista2)
    A     B     C
 TRUE FALSE FALSE
```

Ahora se propone una función que suma la cantidad de filas (o elementos si es un vector) de dos matrices y luego se usa esa función como argumento en `mapply()` para saber la suma de las longitudes de los elementos de las listas definidas en el script anterior:

Script de entrada en R

```
# Función que suma la cantidad de filas (o longitud)
# de cada uno de los respectivos elementos de las listas
simpleFunc <- function(x, y)
{  NROW(x) + NROW(y)
}
# Aplicando SimpleFunc sobre las dos lista
# Suma cantidad de filas elemento a elemento de las listas
mapply(simpleFunc, lista1, lista2)
```

Consola de salida de R

```
> # Función que suma la cantidad de filas (o longitud)
> # de cada uno de los respectivos elementos de las listas
> simpleFunc <- function(x, y)
+ {  NROW(x) + NROW(y)
+ }
> # Aplicando SimpleFunc sobre las dos lista
> # Suma cantidad de filas elemento a elemento de las listas
> mapply(simpleFunc, lista1, lista2)
 A  B  C
 8 10 20
```

Se puede emplear la función `sapply()` para obtener la compuesta de dos funciones matemáticas. Por ejemplo, sean $f(x) = log_{10}(x)$ y $g(x) = x^2$. La función compuesta:

$$f \circ g(x) = f\big(g(x)\big) = log_{10}(x^2)$$

evaluada para x variando dese 1 hasta 5, se codifica en R con el siguiente script:

Script de entrada en R

```
x <- 1:5
x^2
g <- function(x) x^2
f <- function(x) log10(x)
```

```
sapply(x, function(a) f(g(a)))[16]

# Equivalente
sapply(sapply(x, g), f)
```

Consola de salida de R

```
> x <- 1:5
> x^2
[1]  1  4  9 16 25
> g <- function(x) x^2
> f <- function(x) log10(x)
> sapply(x, function(a) f(g(a)))
[1] 0.0000000 0.6020600 0.9542425 1.2041200 1.3979400
>
> # Equivalente
> sapply(sapply(x, g), f)
[1] 0.0000000 0.6020600 0.9542425 1.2041200 1.3979400
```

9.2.1 Operaciones marginales en matrices y la función apply()

Como objetos numéricos, las matrices resultan útiles para hacer diversidad de cálculos. Una de sus principales características es que tanto sus filas como sus columnas pueden ser tratadas como elementos individuales. De esta forma, hay operaciones que se efectúan para todas sus columnas o para todas sus filas; a estas se les denominará ***operaciones marginales***.

El lenguaje R tiene algunas de estas operaciones implementadas directamente, entre ellas están las funciones **`rowSums()`**, **`colSums()`**, **`rowMeans()`**y **`colMeans()`**, como se muestra a continuación:

Script de entrada en R

```
# Creando un data frame de tres columnas numéricas
misDatos1 <- data.frame(uno = sample(5), dos = sample(5),
                        tres = sample(5))
# Convirtiendo el data frame a una matriz
miMatriz<-as.matrix(misDatos1)
miMatriz
```

[16] Aquí se ha hecho uso de las funciones anónimas. Ver 9.3.

```
# Ejemplos de operaciones marginales
colSums(miMatriz)  # Suma por columnas
rowSums(miMatriz)  # Suma por filas
colMeans(miMatriz)  # Promedios por columnas
rowMeans(miMatriz)  # Promedios por filas
```

Consola de salida de R

```
> # Creando un data frame de tres columnas numéricas
> misDatos1 <- data.frame(uno = sample(5), dos = sample(5),
+                         tres = sample(5))
> # Convirtiendo el data frame a una matriz
> miMatriz<-as.matrix(misDatos1)
> miMatriz
     uno dos tres
[1,]   5   5    1
[2,]   1   1    3
[3,]   2   3    2
[4,]   3   2    5
[5,]   4   4    4
> # Ejemplos de operaciones marginales
> colSums(miMatriz)  # Suma por columnas
 uno  dos tres
  15   15   15
> rowSums(miMatriz) # Suma por filas
[1] 11  5  7 10 12
> colMeans(miMatriz)  # Promedios por columnas
 uno  dos tres
   3    3    3
> rowMeans(miMatriz)  # Promedios por filas
[1] 3.666667 1.666667 2.333333 3.333333 4.000000
```

El lenguaje provee además la posibilidad de construir el mismo tipo de operación marginal para el caso general de cualquier función, mediante la función **`apply()`**:

`apply(x, MARGIN, FUN, ...)` – retorna un vector, arreglo o lista de valores obtenidos al aplicar una función a las filas o columnas de un arreglo o matriz,

donde:

- **`x`**: un arreglo o una matriz.
- **`MARGIN`**: un vector con los subíndices sobre los que la función **`FUN`** será aplicada. Para una matriz el valor **1** indica fila, el valor **2** indica columna y **c(1, 2)** indica filas y columnas.

Cuando **x** tiene atributo `dimnames`, puede ser un vector de caracteres especificando el nombre de la dimensión.

- **FUN**: La función a ser aplicada. En caso de aplicar funciones operadoras como +, %*%, etc., la función debe encerrarse entre apóstrofos o apóstrofos invertidos.
- **...**: argumentos opcionales de la función.

Por ejemplo, la función **sd()** calcula la desviación estándar para un conjunto de números dados como un vector numérico. Si se quisiera aplicar esta función a todas las filas o a todas las columnas de una matriz, el problema se puede resolver haciendo uso de la función **apply()**.

Script de entrada en R

```
# miMatriz del ejemplo anterior
# Aplicando sd a las filas, MARGIN=1
apply(miMatriz, 1, sd)
# Aplicando sd() a las columnas, MARGIN=2
apply(miMatriz, 2, sd)
# Cuando x tiene atributo dimnames
names(dimnames(miMatriz)) <- c("FILA", "COLUMNA")
# Aplicando sd a las filas, MARGIN="FILA"
apply(miMatriz, "FILA", sd)
# Aplicando sd() a las columnas, MARGIN="COLUMNA"
apply(miMatriz, "COLUMNA", sd)
# Cuando x tiene atributo dimnames
```

Consola de salida de R

```
> # miMatriz del ejemplo anterior
> # Aplicando sd a las filas, MARGIN=1
> apply(miMatriz, 1, sd)
[1] 2.3094011 1.1547005 0.5773503 1.5275252 0.0000000
> # Aplicando sd() a las columnas, MARGIN=2
> apply(miMatriz, 2, sd)
     uno      dos     tres
1.581139 1.581139 1.581139
> # Cuando x tiene atributo dimnames
> names(dimnames(miMatriz)) <- c("FILA", "COLUMNA")
> # Aplicando sd a las filas, MARGIN="FILA"
> apply(miMatriz, "FILA", sd)
[1] 1.0000000 1.7320508 0.5773503 2.3094011 1.7320508
```

```
> # Aplicando sd() a las columnas, MARGIN="COLUMNA"
> apply(miMatriz, "COLUMNA", sd)
     uno      dos     tres
1.581139 1.581139 1.581139
```

Una variante de aplicación de `apply()` es con la función **`sweep()`**, que aplicada a un vector o matriz retorna otro vector o matriz resultante de aplicarle la función **`FUN`** por fila o columna, empleando los valores especificados en **STATS**, de forma marginal.

```
sweep(x, MARGIN, STATS, FUN = "-", check.margin = TRUE, ...)
```

donde **`x`**, **`MARGIN`**, **`FUN`** y **`...`** tienen el mismo significado que en `apply()`, **`STATS`** valores a ser empleados por **FUN** y **`check.margin`** es un valor lógico (TRUE por defecto) para advertir si la longitud o dimensiones de `STATS` no coinciden con las dimensiones especificadas para `x`. Si se sabe que las dimensiones coinciden, es mejor colocar FALSE y se gana en velocidad.

Se brinda un ejemplo de transformaciones de una matriz usando esta función. Note como se emplea cada expresión en `STATS` sobre los elementos de cada fila o columna seleccionada de la matriz, aplicando la función dada en el parámetro `FUN`.

Script de entrada en R

```
m <- matrix(c(1:4, 1, 6:8), nr = 2)
m
# Restar 3 a la fila 1 y 5 a la fila 2
sweep(m, MARGIN=1, STATS=c(3,5), FUN="-")
# Dividir cada elemento de las columnas 1 a 4 por 2,2,5,5; respectivamente
sweep(m, MARGIN=2, STATS=c(2,2,5,5), FUN="/")
```

Consola de salida de R

```
> m <- matrix(c(1:4, 1, 6:8), nr = 2)
> m
     [,1] [,2] [,3] [,4]
[1,]    1    3    1    7
[2,]    2    4    6    8
> # Restar 3 a la fila 1 y 5 a la fila 2
> sweep(m, MARGIN=1, STATS=c(3,5), FUN="-")
     [,1] [,2] [,3] [,4]
[1,]   -2    0   -2    4
[2,]   -3   -1    1    3
```

```
> # Dividir cada elemento de las columnas 1 a 4 por 2,2,5,5; respectivamente
> sweep(m, MARGIN=2, STATS=c(2,2,5,5), FUN="/")
     [,1] [,2] [,3] [,4]
[1,]  0.5  1.5  0.2  1.4
[2,]  1.0  2.0  1.2  1.6
```

9.3 Funciones anónimas

Las funciones usadas hasta el momento son de dos tipos: predefinidas en R (o en alguno de sus paquetes) o han sido desarrolladas por el programador. Pero en ocasiones es conveniente usar funciones anónimas de esta manera:

```
sapply(1:10, function(x) if(x < 5) x^2 else -x^2)
```

Una función anónima no tiene asignado un nombre. Es útil en contextos en que forman parte de otra operación y ellas en sí ocupan poco espacio. En R son bastante usadas por la familia de las funciones apply.

Otro ejemplo de uso de una función anónima es el cálculo de la raíz de la suma cuadrática de cada columna de un data frame:

```
df <- data.frame(first=5:9, second=(0:4)^2, third=-1:3)

apply(df, 2, function(x) sqrt(sum(x^2)))
```

Su ejecución resulta en:

```
    first     second      third

15.968719  18.814888   3.872983
```

Como último ejemplo, se desea encontrar el primer valor par de un vector de números enteros. Una posible solución es:

```
es_par <- function(n) if (n%%2==0) TRUE else FALSE

vp <- c(1,3,2,5,4)

Find(es_par,vp)
```

con salida en consola:

```
[1] 2
```

Una solución similar (con idéntico resultado) usando funciones anónimas es:

```
vp <- c(1,3,2,5,4)
Find(function(n) if (n%%2==0) TRUE else FALSE,vp)
```

Las funciones anónimas, debidamente usadas, confieren brevedad y expresividad al código.

9.4 Funcionales del análisis matemático y numérico

9.4.1 Raíces de un polinomio en una indeterminada

La función **`polyroot(x)`** permite obtener las raíces (reales y complejas) de un polinomio en una indeterminada con coeficientes reales, donde `x` es un vector con los coeficientes del polinomio, dispuestos en orden creciente de potencia. La función retorna las raíces de la ecuación, expresadas como números complejos.

Por ejemplo, se desea hallar las raíces de $\boldsymbol{f(x) = x^2 + x - 2}$**.** La ecuación se puede descomponer en $\boldsymbol{(x + 2)(x - 1) = 0}$, por lo que sus raíces son: $\boldsymbol{x_1 = -2}$ y $\boldsymbol{x_2 = 1}$. Con la llamada:

```
polyroot(c(-2, 1, 1))
```

se obtiene como resultado:

```
[1]  1-0i -2+0i
```

9.4.2 Obtención de la raíz de una función en un intervalo dado

Para hallar una raíz de una función en un intervalo dado se usa la función **`uniroot()`**:

`uniroot(f, interval,..., lower = min(interval), upper = max(interval), etc)`

donde:

- **`f:`** función a la que se le busca la raíz.

- **`interval:`** vector que contiene los extremos del intervalo donde se busca la raíz.
- **`...:`** argumentos adicionales nombrados o no a ser pasados a f.
- **`lower, upper`**: extremos inferior y superior del interval de búsqueda. Se debe introducir interval o estos valores.
- **`etc:`** otros argumentos.

Por ejemplo, si deseamos hallar una raíz de la función $f(x) = x^2 + x - 2$ en el intervalo [0,2]; que ya sabemos que es 1, hacemos:

```
uniroot(function(x) x ^ 2 + x - 2, c(0, 2))
```

La función devolverá una lista que contiene la raíz aproximada, el valor de la función f en ese punto, el número de iteraciones que requiere el cálculo y la precisión con que fue calculada la raíz. Por tanto la salida será (aproximadamente):

```
$`root`
[1] 0.9999882
$f.root
[1] -3.53757e-05
$iter
[1] 6
$init.it
[1] NA
$estim.prec
[1] 6.103516e-05
```

Otro ejemplo: hallar la raíz de la función $f(x) = x^2 - e^x$ en el intervalo [-2,1].

Si se invoca:

```
uniroot(function(x) x ^ 2 - exp(x), c(-2, 1))
```

se obtendrá la salida:

```
$`root`
[1] -0.7034583
$f.root
[1] -1.738305e-05
$iter
[1] 6
$init.it
[1] NA
$estim.prec
[1] 6.103516e-05
```

En los llamados precedentes de búsqueda de raíces se pasa una función anónima a **`uniroot()`**.

9.4.3 Derivada simbólica de una función

La función **`D()`** calcula la derivada simbólica de una función respecto a una variable dada.

`D(expr, nomvar)`

- **`expr`**: es una expresión o fórmula simbólica.
- **`nomvar`**: es un vector de caracteres con el nombre de la variable (solo una) respecto a la cual se calcula la derivada.

Una fórmula simbólica se puede obtener usando la función **`quote(expr)`**, donde **`expr`** es cualquier expresión sintácticamente válida en R.

Por ejemplo, para hallar la derivada de $f(x) = x^2$ respecto a x se hace la invocación:

```
D(quote(x ^ 2), "x")
```

que da como resultado:

```
2 * x
```

lo cual es una función no evaluada (en R estas funciones son del tipo call).

Para $f(x) = sin(x) + cos(x.y)$, su derivada se calcula con:

```
D(quote(sin(x) * cos(x * y)), "x")
```

o también usando la función **`expression()`** que crea objetos de la clase **expression**:

```
D(expression(sin(x) * cos(x * y)), "x")
```

y el resultado será:

```
cos(x) * cos(x * y) - sin(x) * (sin(x * y) * y)
```

Puede observarse el uso de las funciones **`quote()`** y **`expression()`** que mantienen la función no evaluable y los símbolos son accedidos según han sido escritos. Como la derivada es también una función no evaluada, ella puede evaluarse, dando los valores de todos los símbolos necesarios, usando la función **`eval()`**:

Script de entrada en R

```
# Calculando derivada símbolica de sin(x)*cos(x*y) respecto a x
fd <- D(quote(sin(x) * cos(x * y)), "x")
fd
class(fd)
# Se evalúa la función derivada para x e y
eval(fd, list(x = 1, y = 2))
```

Consola de salida de R

```
> # Calculando derivada símbolica de sin(x)*cos(x*y) respecto a x
> fd <- D(quote(sin(x) * cos(x * y)), "x")
> fd
cos(x) * cos(x * y) - sin(x) * (sin(x * y) * y)
> class(fd)
[1] "call"
> # Se evalúa la función derivada para x e y
> eval(fd, list(x = 1, y = 2))
[1] -1.75514
```

En este ejemplo **`quote()`** crea un objeto de clase `call` y **`eval()`** evalúa la expresión dada para valores determinados de sus símbolos (para una función serían sus variables).

9.4.4 Integración numérica

R permite la integración numérica de una función, a través de la integral definida en un intervalo dado. Por ejemplo, $\int_0^2 x\,dx$, que es 2, puede ordenarse con la llamada:

Script de entrada en R

```
result <- integrate(function(x) x, 0, 2)
result
```

Consola de salida de R

```
> result <- integrate(function(x) x, 0, 2)
> result
2 with absolute error < 2.2e-14
```

En la salida aparece el resultado (2) y el error que se comete (2.2e-14). Un formato simplificado de la función **`integrate()`** se da a continuación:

```
integrate(f, lower, upper, ..., subdivisions = 100L, etc)
```

donde:

- **`f`**: una función que toma un primer argumento y retorna un vector numérico de igual longitud.
- **`lower, upper`**: límites de integración. Pueden ser infinitos `(-Inf, Inf)`.
- **`...`** : argumentos adicionales a ser pasados a f.
- **`subdivisions`**: cantidad máxima de subintervalos de integración (numérica).
- **`etc`**: otros argumentos.

Obsérvese que:

```
result <- integrate(function(x) x, 0, 2)
```

es lo mismo que:

```
mif <- function(x) {x}
result <- integrate(mif, 0, 2)
```

sin usar funciones anónimas.

9.4.5 Obtención del máximo o mínimo de una función en un intervalo dado

La función **optimize()** devuelve la abscisa del máximo (o el mínimo) de una función en un intervalo dado y su correspondiente valor de ordenada.

```
optimize(f, interval, ..., lower = min(interval),
         upper =  max(interval), maximum = FALSE,
         tol = .Machine$double.eps^0.25)
```

con:

- **f:** función a optimizar. En dependencia del valor del parámetro **maximum** se calcula el máximo o el mínimo.
- **interval:** vector que contiene los extremos del intervalo en que se busca el mínimo o máximo.
- **...** : parámetros adicionales opcionales.
- **lower**, **upper:** extremos inferior y superior del intervalo. Es otra forma de expresar **interval**.
- **maximum:** si es FALSE (valor por defecto) se calcula el mínimo, si es TRUE se calcula el máximo.
- **tol:** precisión deseada.

La función devuelve una lista de dos elementos: **minimum** y **objective.** En **minimum** aparece el valor donde se alcanza el máximo o el mínimo de la función **f** y en **objective** el valor de **f** evaluada en ese máximo o mínimo.

Script de entrada en R

```
# Mínimo del seno entre 0 y pi
optimize(sin, c(0, pi))
```

Consola de salida en R

```
> # Mínimo del seno entre 0 y pi
> optimize(sin, c(0, pi))
$`minimum`
[1] 7.932714e-05

$objective
[1] 7.932714e-05
```

Script de entrada en R

```
# Máximo del seno entre 0 y pi
optimize(sin, c(0,pi), maximum=T)
```

Consola de salida en R

```
> # Máximo del seno entre 0 y pi
> optimize(sin, c(0,pi), maximum=T)
$`maximum`
[1] 1.570796

$objective
[1] 1
```

Script de entrada en R

```
# Función f(x) = x^2
f <- function(x) x^2
# Mínimo de f entre -1 y 2
optimize(f, c(-1, 2))
```

Consola de salida en R

```
> # Función f(x) = x^2
> f <- function(x) x^2
> # Mínimo de f entre -1 y 2
> optimize(f, c(-1, 2))
$`minimum`
[1] 0

$objective
[1] 0
```

Script de entrada en R

```
# Máximo de f como función anónima entre -1 y 2
optimize(function(x) x^2, c(-1,2), maximum = T)
```

Consola de salida en R

```
# Máximo de f como función anónima entre -1 y 2
> optimize(function(x) x^2, c(-1,2), maximum = T)
$maximum
[1] 1.999924

$objective
[1] 3.999697
```

9.4 Desarrollo de funciones binarias propias

En R todos los operadores predefinidos son realmente funciones, usando como sintaxis una notación infija[17] para operadores binarios, por ejemplo, `x+y`. También es permitido usar la notación de función para ello, usando apóstrofo invertido, de tal forma que `x+y` se podría escribir también como **`` `+`(x,y) ``**. Entonces ejecutar **`` `+`(4,5) ``** daría como resultado **9**.

El programador de R puede escribir sus propias funciones binarias, escribiendo su nombre entre caracteres de porcentaje (**%**), con dos argumentos de un tipo conveniente y un comando "return" que devuelve un valor de ese tipo. A continuación, se muestra una función binaria que suma el primer operando con el doble del segundo.

Script de entrada en R

```
# Definición de función binaria a2b
"%a2b%" <- function(a,b) return(a+2*b)
# Uso de la función a2b
x <- 3 %a2b% 5
print(x)
```

Consola de salida de R

```
> # Definición de función binaria a2b
> "%a2b%" <- function(a,b) return(a+2*b)
> # Uso de la función a2b
> x <- 3 %a2b% 5
> print(x)
[1] 13
```

---

[17] Notación infija es aquella en que se escriben los operadores entre los operandos sobre los que actúan, por ejemplo, 6+7.

9.5 La función split()

Generalmente, aunque los datos con los que se quiere trabajar están dados en tablas, no están organizados de la manera que se pretende trabajar sobre ellos. La función **split(),** permite clasificar los datos, típicamente dados como un vector o un data frame.

Por ejemplo, si se tiene un vector (data) con un grupo de letras y se quiere separar en consonantes y vocales podemos hacer:

```
data <- c("e", "o", "r", "g", "a", "y", "w", "q", "i")
# Primero se crea un grupo de letras (las vocales)
vocales <- c('a','e','i','o','u')
# Separando data en vocales y consonantes (resultado en tipo_letra)
tipo_letra <- ifelse(data %in% vocales,"data_vocal","data_consonante")
```

Note que el vector `tipo_letra` tiene la misma cantidad de elementos que `data`. Para dividir `data` en los dos grupos, consonantes y vocales, empleamos la función **`split`**:

```
split(data, tipo_letra)
```

Con su ejecución se obtiene la salida:

```
$data_consonante
[1] "r" "g" "y" "w" "q"
$data_vocal
[1] "e" "o" "a" "i"
```

Suponga que se dispone de datos de las precipitaciones para distintas estaciones, para distintos años y distintos meses, los cuales se encuentran en el fichero Precipitaciones.txt, en su directorio de trabajo.

¿Cómo leer estos datos?

Estación	Año	Enero	Febrero	Marzo
E1	1978	54.0	38.5	NA
E1	1979	21.5	21.0	38.5
E1	1980	81.0	31.0	4.0
E1	1982	37.5	NA	30.0
E2	1979	NA	NA	155.0
E2	1980	105.2	17.5	246.3
E2	1981	60.3	3.2	100.0
E3	1979	10.8	30.3	60.6
E3	1980	10.1	NA	70.6
E3	1981	20.2	40.4	80.6
E3	1982	20.3	50.5	90.6

Script de entrada en R

```
p <- read.table("Precipitaciones.txt",header=T)
p
```

Consola de salida en R

```
> p <-read.table("Precipitaciones.txt",header=T)
> p
   Estacion   Anho   Enero   Febrero   Marzo
1        E1   1978    54.0      38.5      NA
2        E1   1979    21.5      21.0    38.5
3        E1   1980    81.0      31.0     4.0
4        E1   1982    37.5        NA    30.0
5        E2   1979      NA        NA   155.0
6        E2   1980   105.2      17.5   246.3
7        E2   1981    60.3       3.2   100.0
8        E3   1979    10.8      30.3    60.6
9        E3   1980    10.1        NA    70.6
```

```
10       E3   1981    20.2      40.4    80.6
11       E3   1982    20.3      50.5    90.6
```

Si se desea tener la misma información clasificada por años, se puede emplear la función **split()**, con sintaxis:

```
split(x, f, drop = FALSE, ...)
```

donde:

- **x:** es el objeto a ser clasificado (típicamente un data frame, lista o un vector)
- **f:** es una serie de datos que servirán para la clasificación (típicamente un factor o dato que puede ser interpretado como factor, como vector numérico o de cadenas de caracteres) o lista de factores. Este segundo argumento debe tener la misma longitud que las columnas del data frame dado como primer argumento, o del vector dado como argumento. No es necesario que **f** sea parte del objeto dado como primer argumento.
- **drop**: indica si niveles de factores vacios pueden eliminarse o no. Si es TRUE, son eliminados. Por defecto, drop es FALSE.

El valor retornado por la función **split()** es una lista de vectores que contiene los valores para los grupos definidos por **f**. Las componentes de la lista son nombradas por los niveles de **f** y si **drop = TRUE**, se eliminan niveles no usados.

En el siguiente ejemplo pasamos como objeto a clasificar las columnas 3 a la 5 del data frame creado y almacenado en el objeto **p** y como segundo argumento pasamos la columna correspondiente a los años, obteniendo como resultado al aplicar la función **split()** una ***lista de tablas***, cada una correspondiente a cada uno de los años registrados en la columna **p$Anho,** como se muestra a continuación:

Scritp de entrada en R

```
porAnho<-split(p[,3:5], p$Anho)
porAnho
```

Consola de salida del R

```
> porAnho<-split(p[,3:5], p$Anho)
> porAnho
$`1978`
  Enero Febrero Marzo
1    54    38.5    NA

$`1979`
  Enero Febrero Marzo
2  21.5    21.0  38.5
5    NA      NA 155.0
8  10.8    30.3  60.6

$`1980`
  Enero Febrero Marzo
3  81.0    31.0   4.0
6 105.2    17.5 246.3
9  10.1      NA  70.6

$`1981`
   Enero Febrero Marzo
7   60.3     3.2 100.0
10  20.2    40.4  80.6

$`1982`
   Enero Febrero  Marzo
4   37.5      NA   30.0
11  20.3    50.5   90.6
```

Se puede observar la diferencia de salida, según el valor asignado al argumento `drop`, (cuando hay niveles vacíos) en la corrida de los siguientes scritps, que separan por año y estación las lluvias del mes de marzo:

Script de entrada en R (drop=FALSE)

```
porAnhoEs <- split(p[,5], list(p$Anho, p$Estacion))
porAnhoEs
```

Consola de salida de R (drop=FALSE)

```
> porAnhoEs <- split(p[,5], list(p$Anho,p$Estacion))
> porAnhoEs
$`1978.E1`
[1] NA
```

```
$`1979.E1`
[1] 38.5

$`1980.E1`
[1] 4

$`1981.E1`
numeric(0)

$`1982.E1`
[1] 30

$`1978.E2`
numeric(0)

$`1979.E2`
[1] 155

$`1980.E2`
[1] 246.3

$`1981.E2`
[1] 100

$`1982.E2`
numeric(0)

$`1978.E3`
numeric(0)

$`1979.E3`
[1] 60.6

$`1980.E3`
[1] 70.6

$`1981.E3`
[1] 80.6

$`1982.E3`
[1] 90.6
```

Script de entrada en R (drop=TRUE)

```
porAnhoEs<-split(p[,5], list(p$Anho,p$Estacion), drop = TRUE)
porAnhoEs
```

Consola de salida de R (drop=TRUE)

```
> porAnhoEs<-split(p[,5], list(p$Anho,p$Estacion), drop = TRUE)
> porAnhoEs
$`1978.E1`
[1] NA

$`1979.E1`
[1] 38.5

$`1980.E1`
[1] 4

$`1982.E1`
[1] 30

$`1979.E2`
[1] 155

$`1980.E2`
[1] 246.3

$`1981.E2`
[1] 100

$`1979.E3`
[1] 60.6

$`1980.E3`
[1] 70.6

$`1981.E3`
[1] 80.6

$`1982.E3`
[1] 90.6
```

Para aplicar ahora una operación a cada una de las tablas de la lista resultante, por ejemplo, calcular el promedio de las precipitaciones (función **`colMeans()`**), pueden usarse las funciones **`lapply()`** o **`sapply()`**.

Script de entrada en R

```
# porAnho es una lista,
pp <- lapply(porAnho, colMeans)
pp
```

Consola de salida de R

```
> # porAnho es una lista
> pp <- lapply(porAnho, colMeans)
> pp
$`1978`
  Enero Febrero   Marzo
   54.0    38.5      NA

$`1979`
  Enero Febrero   Marzo
     NA      NA    84.7

$`1980`
    Enero   Febrero      Marzo
 65.43333        NA  106.96667

$`1981`
  Enero Febrero   Marzo
  40.25   21.80   90.30

$`1982`
  Enero Febrero   Marzo
   28.9      NA    60.3
```

Las acciones de los dos últimos ejemplos, que abarcan clasificar y luego aplicar una operación como el promedio, pueden realizarse usando una única función, como **`by()`**:

```
by(obj1, obj2, fun, ...)
```

donde:

`obj1`: objeto al cual se aplica la función (típicamente un data frame o una matriz),

`obj2`: factor o lista de factores usados para clasificar, de longitud igual a la cantidad de filas de **`obj1`**,

`fun`: función que se aplica a cada uno de los elementos resultantes de la clasificación.

`...`: Argumentos adicionales de la función **`fun`**.

Entre los argumentos adicionales está **`na.rm`**, que será pasado a la función que se aplicará, si **`na.rm = TRUE`** se obvian los valores NA en el cálculo.

Como ejemplo, haremos la clasificación y aplicación del promedio sobre el data frame con los datos de precipitaciones, sin considerar las valores NA.

Script de entrada en R

```
# Usando datos de precipitaciones ya conocidos
p <- read.table("Precipitaciones.txt", header=T)
# Clasificando y operando, resultados a rr
rr <- by(p[,3:5], p$Anho, colMeans, na.rm = T)
rr
```

Consola de salida en R

```
> # Usando datos de precipitaciones ya conocidos
> p <- read.table("Precipitaciones.txt", header=T)
> # Clasificando y operando, resultados a rr
> rr <- by(p[,3:5], p$Anho, colMeans, na.rm = T)
> rr
p$Anho: 1978
  Enero Febrero   Marzo
   54.0    38.5     NaN
------------------------------------------------------------------------
p$Anho: 1979
  Enero Febrero   Marzo
  16.15   25.65   84.70
------------------------------------------------------------------------
p$Anho: 1980
    Enero   Febrero     Marzo
 65.43333  24.25000 106.96667
------------------------------------------------------------------------
p$Anho: 1981
  Enero Febrero   Marzo
  40.25   21.80   90.30
------------------------------------------------------------------------
p$Anho: 1982
  Enero Febrero   Marzo
   28.9    50.5    60.3
```

9.5.1 Función strsplit() para cadenas de caracteres

La función **`strsplit(x, split = separador)`** separa el vector de cadenas **`x`** en una lista de cadenas, usando el vector de caracteres **`separador`** (o **`split=separador`**) como referencia. El separador se omite del resultado.

Ejemplo:

Script de entrada en R

```
strsplit("programación", "a")  # Separador letra "a"
strsplit(c("05 Mar", "06 Abr"), split=" ") # Separador espacio
strsplit("programación", split="") # Separador vacío, deletrea
```

Consola de salida de R

```
> strsplit("programación", "a")  # Separador letra "a"
[[1]]
[1] "progr" "m"     "ción"

> strsplit(c("05 Mar", "06 Abr"), split=" ") # Separador espacio
[[1]]
[1] "05"  "Mar"

[[2]]
[1] "06"  "Abr"

> strsplit("programación", split="") # Separador vacío, deletrea
[[1]]
 [1] "p" "r" "o" "g" "r" "a" "m" "a" "c" "i" "ó" "n"
```

9.6 Funciones para importar y exportar códigos

9.6.1 Importación de código

El comando **`source(`**`"fichero.R"`**`)`** permite introducir comandos procedentes de ficheros. Es útil para cargar funciones elaboradas por el usuario o bien para ejecutar macros[18] y scripts. Como ejemplo se define una función que calcula la media de un vector y ese código se guarda en el fichero "MiMedia.R":

```
Media <- function(x)

{ Med<-sum(x)/length(x)

  Med

}
```

[18] **Macros**: Conjunto de comandos que se invocan con una palabra clave, opcionalmente seguidas de parámetros que se utilizan como código literal. Son manejadas por el compilador y no por el ejecutable compilado.

Si en otro programa necesitásemos usar ese código, entonces hacemos:

Script de entrada en R
```
source("MiMedia.R")
Media(x=c(1,2,3,4))
```

Consola de salida en R
```
> source("MiMedia")
> Media(x=c(1,2,3,4))
[1] 2.5
```

9.6.2 Importación con la función `load()`

A continuación se ejemplifica cómo, usando la función **`save()`**, podemos guardar una función y usarla luego en el programa que lo requiera, empleando la misma función. Para ello utilizaremos la función **Media** del epígrafe anterior, pero renombrándola **Media2**:

```
Media2 <- function(x)
{  Med <- sum(x)/length(x)
   Med
}
```

Ahora se guarda en el fichero "**Media2Save.RData**" con:

```
save(Media2,file="Media2Save.RData")
```

Se muestran dos posibles empleos en un nuevo script, limpiando previamente el ambiente global:

Script de entrada en R
```
rm(list = ls())
# Primera llamada
Media2(x=c(1,2,3,4))
```

Consola de salida en R
```
> rm(list = ls())
# Primera llamada
> Media2(x=c(1,2,3,4))
Error in Media2(x = c(1, 2, 3, 4)) : could not find function "Media"
```

Script de entrada en R

```
# Llamada con load previo
load("Media2Save.RData")
Media2(x=c(1,2,3,4))
```

Consola de salida en R

```
# Llamada con load previo
> load("Media2Save.RData")
> Media2(x=c(1,2,3,4))
[1] 2.5
```

Cuando se invoca por primera vez a la función **Media2(1,2,3,4)** ocurre un error, pues la función no ha sido aún cargada en el programa que la llama.

En la segunda llamada, como la función ya ha sido cargada con la función **load()**, el resultado es la media del conjunto de datos dados.

9.6.3 Exportación de código sink

El comando **sink("fichero")** permite que la salida de los comandos posteriores a su invocación se almacene en "**fichero**". Para devolver la salida a la pantalla se usa nuevamente el comando **sink()**. Se usa fundamentalmente para guardar salidas de datos en ficheros de texto. Se muestra a continuación su uso.

Script de entrada en R

```
sink("Copia.txt")
A <- 5
B <- 10
C <- (A+B)/2
# Las salidas van a "Copia.txt"
print(A); print(B); print(C)

# Retorna la salida a la pantalla
sink()
print(A); print(B); print(C)
# Se lee lo exportado a "Copia.txt"
read.csv("Copia.txt", header = F, sep = ",")
```

Consola de salida de R

```
> sink("Copia.txt")
> A <- 5
```

```
> B <- 10
> C <- (A+B)/2
> # Las salidas van a "Copia.txt"
> print(A); print(B); print(C)
>
> # Retorna la salida a la pantalla
> sink()
> print(A); print(B); print(C)
[1] 5
[1] 10
[1] 7.5
> # Se lee lo exportado a "Copia.txt"
> read.csv("Copia.txt", header = F, sep = ",")
       V1
1   [1] 5
2  [1] 10
3 [1] 7.5
```

Ejercicios

1. Escribir una función que calcule la varianza muestral de un conjunto de datos dados, los cuales se introducen mediante un vector, usando la función `Reduce()`.
2. Escribir una función que calcule la covarianza entre dos conjuntos de datos, usando la función `Reduce()`.
3. Dado un conjunto de datos que representan las frecuencias absolutas de n objetos, calcular el vector de las frecuencias acumuladas, usando la función `Reduce()`. Representar esto en forma matricial, donde el nombre de las filas sean los objetos y las columnas representen las frecuencias absolutas y acumuladas.
4. Genere un objeto con la suma acumulada de la secuencia de 1 a 100 y determine cual es el valor y la posición del primer elemento mayor que 50.
5. Guardar las funciones escritas en los ejercicios del 1 al 3, para luego, desde un programa principal en el que se tienen los datos correspondientes al peso (lb) y la talla (cm) de 10 individuos, calcular:

a) La varianza de la talla y el peso (No recalcular la media, usar la función definida en clase).

b) La correlación entre Peso y Talla.

Peso	75	80	94	64	58	70	85	70	69
Talla	180	190	170	160	159	176	192	170	160

6. Dado los datos siguientes, que muestran el número de accidentes, fallecidos y lesionados por accidentes de tránsito en Cuba en el año 2014, calcule la media y la varianza de estos 3 items.

Provincia	**Accidentes**	**Fallecidos**	**Lesionados**
Pinar del Río	458	29	461
Artemisa	422	20	372
La Habana	4 766	160	1 707
Mayabeque	337	33	355
Matanzas	509	57	426
Villa Clara	664	60	706
Cienfuegos	417	46	418
Sancti Spíritus	285	25	223
Ciego de Ávila	256	22	222
Camagüey	528	63	602
Las Tunas	333	32	337
Holguín	710	77	775
Granma	398	35	568
Santiago de Cuba	829	68	1 132
Guantánamo	332	17	482
Isla de la Juventud	50	2	45

7. Use **`integrate()`** y una función anónima para encontrar el área bajo la curva de las siguientes funciones en los intervalos dados.

a) $y = x^2 - x\ , x \in [0, 10]$

b) $y = \sin(x) + \cos(x), x \in [-\pi,\ \pi]$

c) $y = {}^{e^x}/_{x},\ x \in [10, 20]$

8. En el siguiente ejemplo explicar por qué la función devuelve los resultados dados. ¿De qué otra forma usted escribiría una función que permita duplicar, triplicar, etc. un valor dado?

```
Construye.multiplicador <- function(n)
{  fff <- function(x)
   {   n*x
   }
   fff # Retorna como resultado la función creada
}
duplica <- Construye.multiplicador(2)
triplica <- Construye.multiplicador(3)
```

Script de entrada en R

```
duplica(5)
triplica(5)
```

Consola de salida de R

```
> duplica(5)
[1] 10
> triplica(5)
[1] 15
```

9. Generar los números de Fibbonacci del 1 al 50.

   Recuerde que los números de Fibonacci responden a la siguiente regla de formación:

$$\boldsymbol{F(n) = \begin{cases} 1 & si\ \ n = 0\ o\ n = 1 \\ F(n-2) + F(n-1) & si\ \ n > 1 \end{cases}}$$

10. Definir una función que, dado un vector que representa las edades de **n** individuos, determine la edad promedio y la cantidad de individuos cuyas edades están por encima de

la edad promedio. Luego, genere aleatoriamente una matriz que contenga las edades (filas) de 10 individuos para 3 municipios del país (columnas) y calcule la edad promedio y la cantidad de individuos cuyas edades están por encima de la edad promedio para cada municipio. Aplique para ello la función `apply()`.

11. Crear una matriz de 20x5, donde cada columna se compone de valores aleatorios siguiendo una distribución normal. Luego usar la función `apply()` para calcular la media y desviación estándar de cada columna.
12. Dada la ley de distribución de una magnitud aleatoria discreta bidimensional, confeccione en R una función que permita, dada la ley de probabilidad de la variable aleatoria bidimensional probar que es una ley de probabilidad y en caso afirmativo:
    a) Hallar la ley de probabilidad de cada variable aleatoria.
    b) Verificar si las variables aleatorias son independientes.
    c) Para probar la función use los datos que se muestran en la siguiente tabla.

	X1	X2	X3
Y1	0.10	0.30	0.20
Y2	0.06	0.18	0.16

Unidad 10. Gráficos

El modo de operar el R con las funciones gráficas es diferente del visto hasta el momento, pues el resultado de evaluar una función gráfica no puede ser asignado a un objeto, en vez de ello es enviado a un dispositivo gráfico, el cual no es más que una **ventana gráfica** o un **fichero**. R abre una ventana para mostrar el gráfico si no hay ningún dispositivo abierto. Un dispositivo gráfico se puede abrir con una función que depende del sistema operativo, bajo Windows es **`windows()`.**

Dispositivos gráficos que son ficheros se pueden abrir con una función que depende del tipo de fichero que se quiere crear: **`postscript()`**, **`pdf()`**, **`png()`**, etc. La lista de dispositivos gráficos disponibles se obtiene con el comando **`?device`**.

Cuando están abiertas varias ventanas gráficas, cada una se asocia a un número de dispositivo, siendo el 1 para la consola. El último dispositivo en ser abierto se convierte en el dispositivo "activo" sobre el cual se dibujan (o escriben) los gráficos generadas. A continuación, algunas funciones para manipular las ventanas gráficas usando su número de dispositivo (ndisp):

- `dev.off(ndisp)`: Cierra la ventana cuyo número de dispositivo se especifica en `ndisp`. Si no se especifica, se cierra la ventana activa. Si el dispositivo activo se cierra y hay otros abiertos, el próximo dispositivo se convierte en el activo. Es un error tratar de cerrar el dispositivo 1.
- `graphics.off()`: Cierra todas las ventanas abiertas.
- `dev.list()`: Devuelve los números de dispositivos abiertos.
- `dev.set(ndisp)`: Activa la ventana especificada por el número de dispositivo `ndisp`.
- `dev.cur()`: Devuelve el número de la ventana activa (1 para la consola).

El lenguaje R cuenta con varios sistemas, en general separados, para organizar o especificar visualizaciones gráficas. Por ahora se estudia uno de ellos, el **sistema gráfico básico**, que es el que está disponible en la instalación inicial del lenguaje. El formato general de creación de la mayoría de los gráficos es:

```
función_gráfica(datos, argumento1, argumento2, ...)
```

10.1 Ploteo de gráficos: función `plot()`

La función más simple para graficar es **`plot()`**, que permite observar las relaciones entre dos variables. Cada punto en el gráfico representa una observación en dos variables, donde en el eje de las abscisas (X) se representa una variable y en el eje de las ordendas (Y) la otra.

Como ejemplo, consideremos los datos que se muestran en la tabla 10, que contiene el índice de natalidad por cada 1 000 habitantes de Cuba en los años 1970, 1975, 1980, 1985, 1990, 1995, 2000 y de 2005 a 2016 y escribamos con ellos el fichero de datos Nat2016.txt.

Tabla 10. Indice de natalidad por cada 1 000 habitantes

Año	Nat	Año	Nat	Año	Nat	Año	Nat
1970	27.7	1995	13.5	2008	10.9	2013	11.2
1975	20.8	2000	12.9	2009	11.6	2014	10.9
1980	14.1	2005	10.7	2010	11.4	2015	11.1
1985	18.1	2006	9.9	2011	11.8	2016	10.4
1990	17.6	2007	10.0	2012	11.3		

Script de entrada en R

```
rm(list = ls())
P3 <- read.table("Nat2016.txt", header = TRUE)
P3
plot(P3$Nat)
```

Salida de R: Produce un gráfico de "dispersión", el cual se muestra en la figura 17.

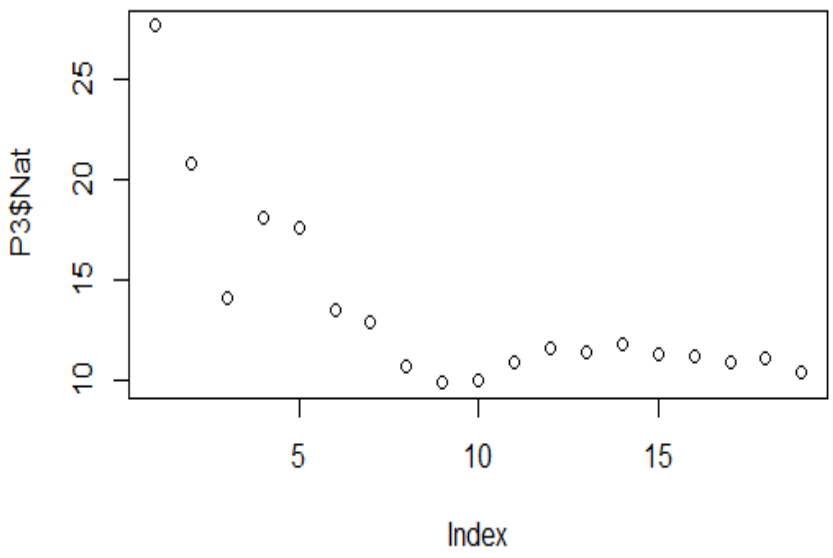


Figura 17. Gráfico de natalidad.

El formato general de la función **`plot()`** es el siguiente:

```
plot(x, y, ...)
```

`x`: abcisas de los puntos a plotear.

`y`: ordenadas de los puntos a plotear.

`...`: otros argumentos, como los parámetros gráficos. Algunos argumentos ampliamente aceptados son:

a) **`type`**: Tipo de ploteo a dibujar, sus valores pueden ser:

- **`"p"`:** puntos (gráfico de dispersión)
- **`"l"`:** líneas
- **`"b"`:** ambos (puntos y líneas)
- **`"c"`:** sólo las líneas del type=`"b"`
- **`"o"`:** puntos y líneas, sobre-graficados
- **`"h"`:** agujas (tipo gráfico de barras o de columnas)
- **`"s"`:** función escalera (horizontal a vertical)
- **`"S"`:** función escalera (vertical a horizontal)
- **`"n"`:** no-grafica (sólo se grafican referencias)

b) Argumentos para cambiar la apariencia del gráfico. A continuación, se mencionan y describen algunos de los más importantes.

- **`main:`** Título principal del gráfico
- **`sub:`** Subtítulo del gráfico
- **`xlab:`** Etiqueta del eje de las abscisas
- **`ylab:`** Etiqueta del eje de las ordenadas
- **`asp:`** Relación de aspecto del área gráfica (y/x)
- **`xlim, ylim:`** Vectores que indican el rango de valores en los ejes x e y
- **`log:`** Los ejes a los que se les aplicarán la función logarítmo, por ejemplo: “”, “x”, “y”, “yx”
- **`col:`** Color de las líneas y los puntos
- **`bg:`** Color del fondo
- **`pch:`** Símbolos para puntos
- **`cex:`** Para escalado de los símbolos de puntos
- **`lty:`** Tipos de línea
- **`lwd:`** Grosor de las líneas.

Otros argumentos y más características de los presentados pueden ser consultados mediante el comando **`?plot`.**

En general, la función **`plot()`** toma como argumentos dos vectores de la misma dimensión, uno para los valores de las abscisas (x) y otro para los valores de las ordenadas (y). Sin embargo, cuando se omite uno de ellos, el lenguaje entiende que las abscisas serían simplemente los índices de los elementos en el vector introducido, es por eso que, por defecto, etiqueta el eje de las abscisas con la palabra **Index**, y que las ordenadas serán, por tanto, cada uno de los elementos del vector introducido. El lenguaje supone que lo que se quiere graficar son los puntos como tal y con el **color** y **tipo** de símbolo seleccionados por omisión o defecto.

Para ejemplificar algunos de estos argumentos se graficará la misma información dada en el fichero Nat2016.txt, pero ahora como líneas azules y considerando como abscisas los años, que son los nombres de las filas del data frame que contiene los datos.

Script de entrada en R

```
P3 <- read.table("Nat2016.txt", header = TRUE)
plot(P3$Anho,     # las abscisas (x)
     P3$Nat,      # las ordenadas (y)
     type="l",    # tipo de gráfico: línea
     col="blue",  # color: azul
     main="Índice de Natalidad 1970-2016",    # Título
     sub="República de Cuba",                 # subtítulo
     xlab="Años",           # etiqueta del eje X
     ylab="Natalidad")      # etiqueta del eje y
```

La salida se muestra en la figura 18.

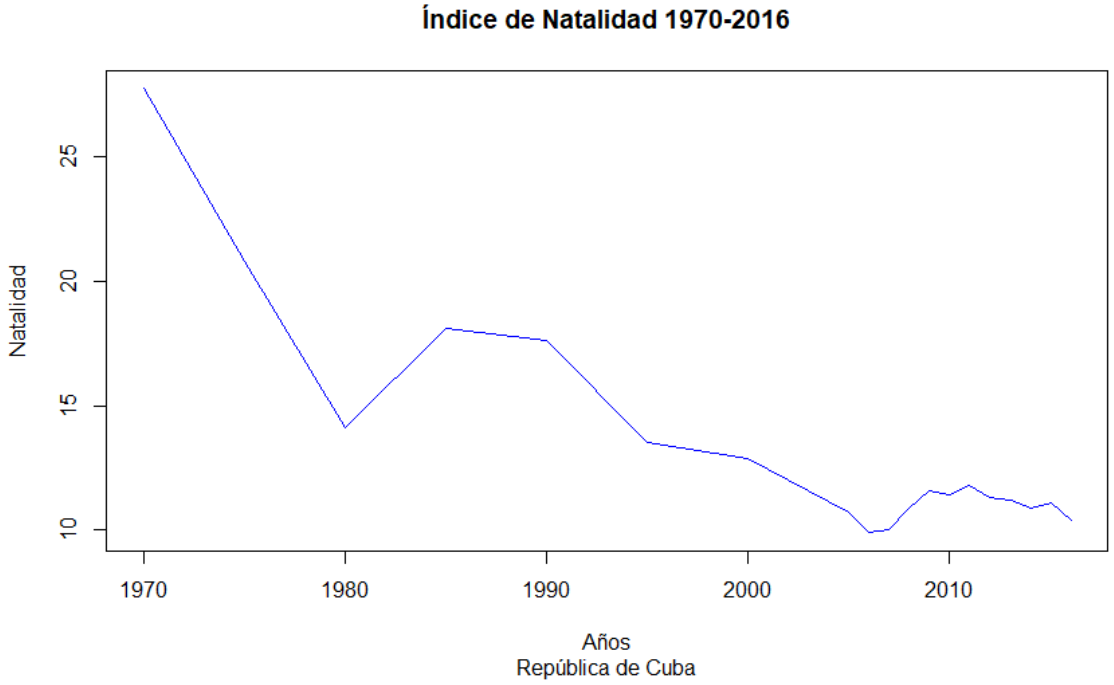


Figura 18. Gráfico de líneas con la misma información que la figura 17.

En general, el gráfico más simple que se pudiera dibujar es el gráfico de una función en el plano cartesiano, por ejemplo, el gráfico de la función **sen(x)** en función de los ángulos. A continuación, se muestra el código para producir ese gráfico.

Script de entrada en R

```
xgrados <- 0:360                # Vector numérico en grados
xradianes <- xgrados*pi/180      # Para convertir los grados en
radianes
seno <- sin(xradianes)
titulo <- "Función seno"
plot(xradianes, seno, type="b", main=titulo, xlab="Ángulos",
     ylab="Seno(x)", col="red", pch=20)
```

Salida gráfica en R:

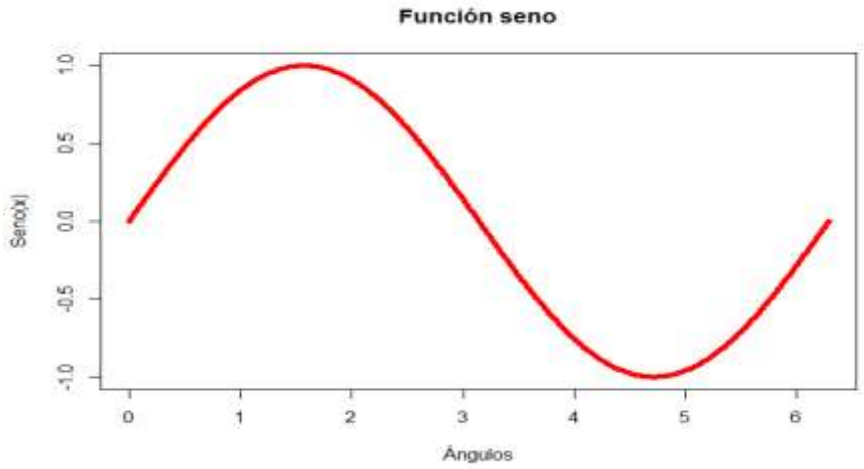
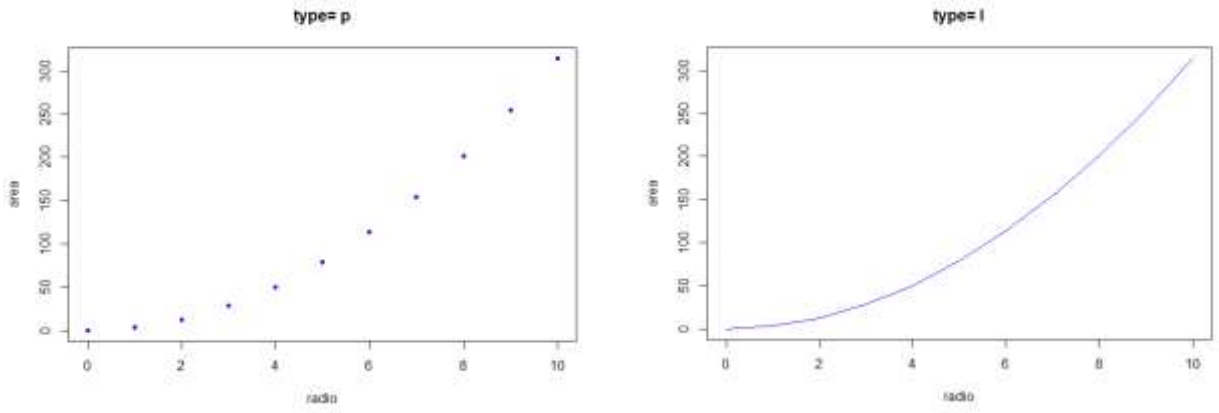
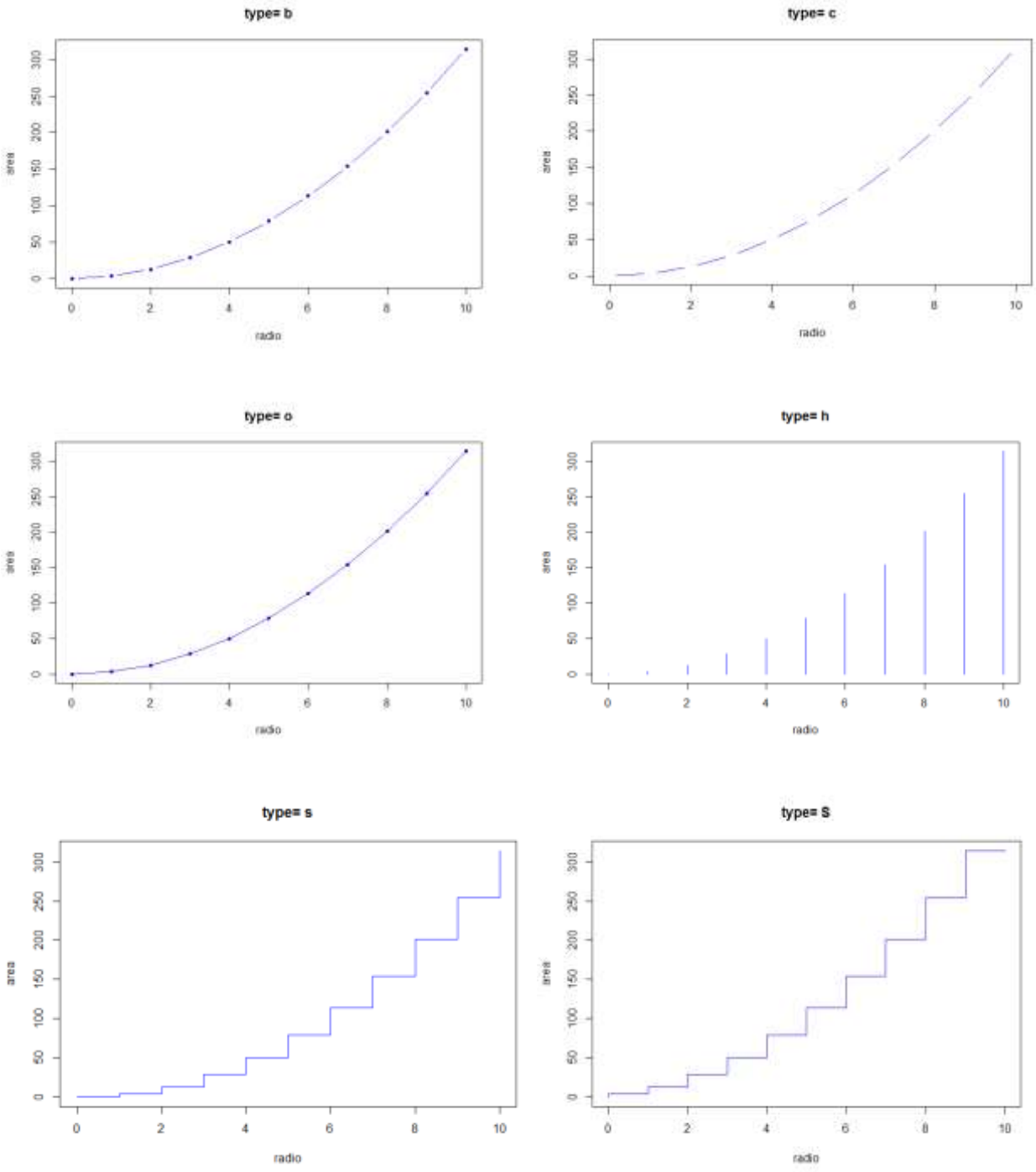


Figura 19. Gráfico de la función seno.

La figura 20 muestra algunos de los tipos de gráficos dibujados con **`plot()`**:




type= b
type= c
type= o
type= h
type= s
type= S
area
radio
0
50
100
150
200
250
300
0
2
4
6
8
10

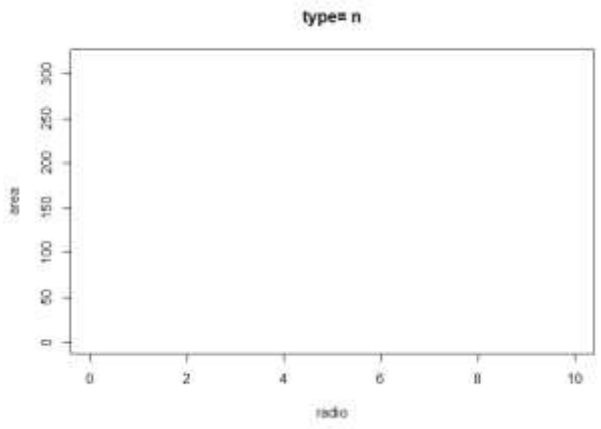


Figura 20. Tipos de gráficos que se pueden producir con plot().

10.1.1 Colores y símbolos para el ploteo de puntos del gráfico

En todos los ejemplos anteriores, se han manejado los colores mediante nombres, tales como: “red”, “blue”, “navyblue”, “violetred”, etc. El lenguaje proporciona diversos mecanismos para especificar los colores, sea por su nombre o por un número entero equivalente.

La figura 21 muestra los colores numerados del 1 al 16; del 1 al 8, en orden: **"black"** negro, **"red"** rojo, **"green"** verde, **"blue"** azul, **"cian"** azul celeste, **"magenta"** malva, **"yellow”** amarillo y **"gray"** gris. A partir del color 9 hasta el 16 la secuencia se repite:

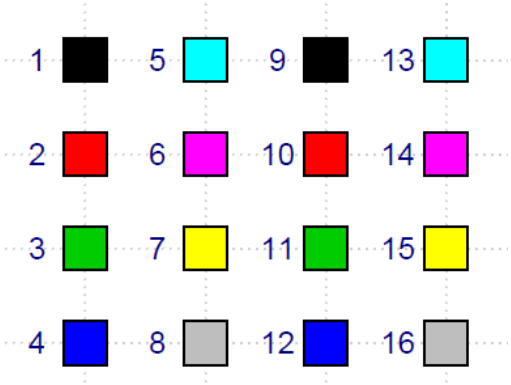


Figura 21. Colores del 1 al 16.

La figura 22 muestra los símbolos para el ploteo de los puntos (argumento **pch**):

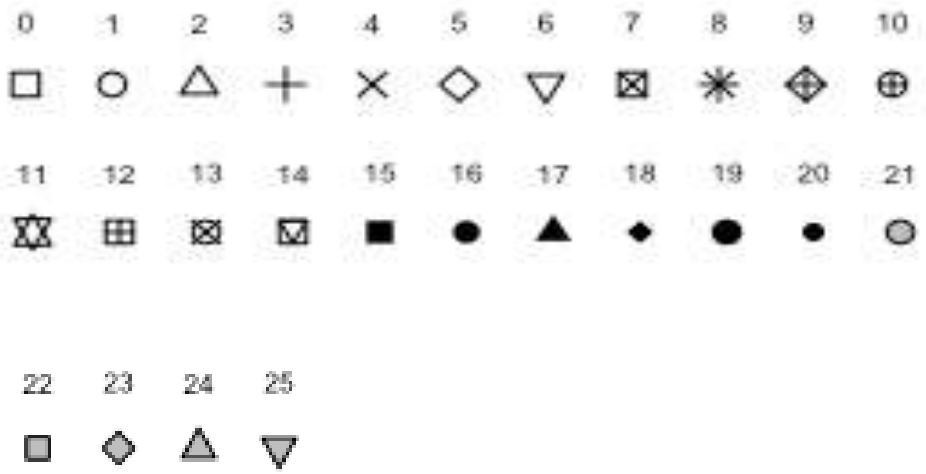


Figura 22. Valores del argumento pch.

10.1.2 Superponiendo dos gráficos y el uso de la leyenda

A modo de ejemplo, la figura 23 muestra un gráfico con dos funciones: el área y el perímetro de un círculo dado su radio. Para poder dibujarlas se emplea la función **`lines()`**, la cual permite agregar al gráfico actual otro de líneas y sus argumentos son semejantes a los de **`plot()`**. Además, para distinguir una curva de la otra resulta muy conveniente añadir una leyenda con la simbología y textos adecuados; esto se hace por medio de la función **`legend()`**.

Script de entrada en R

```
rm(list = ls())
aa <- "Área Círculo"          # Texto para leyenda
pp <- "Perímetro Círculo"     # Texto para leyenda
etiq.x <- "Radio"             # Etiqueta eje X
etiq.y <- "Área-Perímetro"    # Etiqueta eje Y
radio <- seq(0,5, by=0.5)     # Vector de radios
area <- pi*radio^2            # Vector de áreas
perim <- 2*pi*radio           # Vector de perímetros
plot(radio, area, type="o", xlab=etiq.x, ylab=etiq.y, pch=21,
       col="red", bg="gold")
# Colocando sobre el gráfico anterior el del perímetro
lines(radio, perim, type="o", pch=23, col="blue", bg="magenta")
legend("topleft",             # ubicación leyenda
      legend = c(aa, pp),     # textos de leyenda
```

```
       lty = c(1, 1),          # tipo de línea
       pch = c(21, 23),        # símbolos
       col = c("red", "blue"),          # color líneas
       pt.bg = c("gold", "magenta") )  # relleno de símbolo
```

El gráfico resultante se presenta en la figura 23.

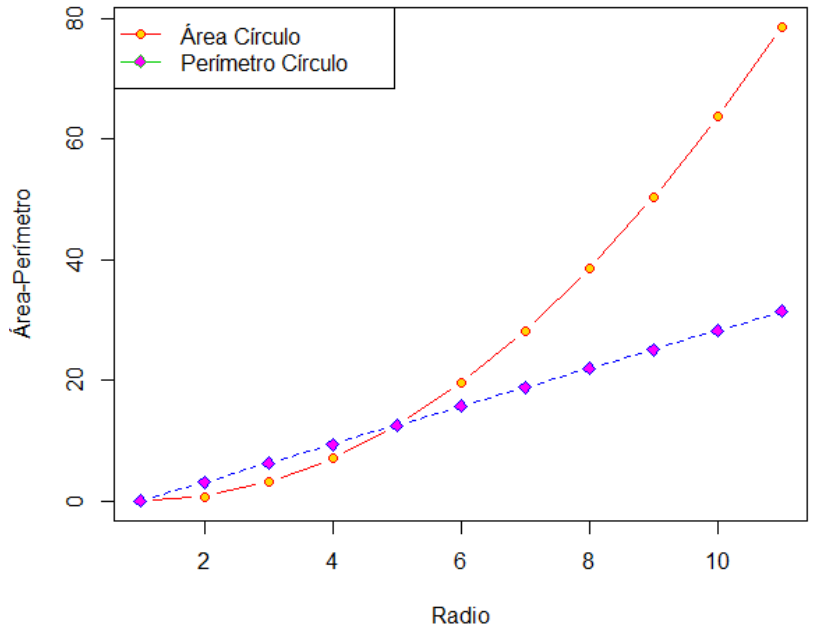


Figura 23. Área y perímetro de círculos dado su radio.

10.1.3 Superposición de ploteos usando `matplot()`

La función matplot permite plotear más de una línea en un gráfico, de una forma más simplificada.

Por ejemplo, se vuelven a dibujar las funciones de área y perímetro como en la figura 23, ahora usando **`matplot()`**:

```
aa <- "Área Círculo"          # Texto para leyenda

pp <- "Perímetro Círculo"     # Texto para leyenda

etiq.x <- "Radio"             # Etiqueta eje X

etiq.y <- "Área-Perímetro"    # Etiqueta eje Y

radio <- seq(0,5, by=0.5)     # Vector de radios
```

```
area <- pi*radio^2           # Vector de áreas
perim <- 2*pi*radio          # Vector de perímetros
tabla <- cbind(area, perim)  # Matriz con los valores a plotear por
                             # columnas
matplot(tabla, type="b", pch=c(21, 23), col=c("red","blue"),
        bg=c("gold","magenta"), xlab=etiq.x, ylab=etiq.y)
legend("topleft",             # ubicación leyenda
       legend = c(aa, pp),    # textos de leyenda
       lty = c(1, 1),         # tipo de línea
       pch = c(21, 23),       # símbolos
       col = 2:3,             # color líneas
       pt.bg = c("gold", "magenta"))  # relleno de símbolo
```

A continuación, se muestra un programa que plotea 3 funciones, dados sus valores.

```
# Se plotean 3 gráficos cuyos datos van a una matriz por columnas
g1 <- c(3.8, 5.5, 9.9, 15.7, 21.5, 26.3)
g2 <- c(19.5, 19., 19.7, 20.8, 21.3, 22.7)
g3 <- c(13.7, 15.4, 20.0, 24.6, 28.5, 32.7)
tabla <- cbind(g1, g2, g3)
print(tabla)
matplot(tabla, type="b", pch=15:17, col=2:4)
```

Se grafica en la figura 24:

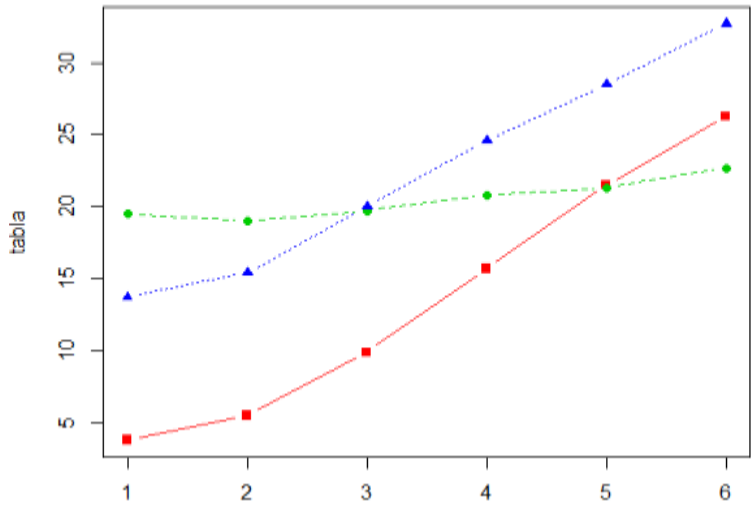


Figura 24. Superposición de tres gráficos.

10.2 Gráficos de una variable

10.2.1 Diagramas o gráficos de barras (barplot)

Los gráficos de barra muestran la distribución (frecuencias) de una variable categórica mediante barras verticales u horizontales. Su forma más simple es **`barplot(heigh)`** donde **`height`** es un vector o una matriz. Algunos de los parámetros de `barplot()` se describen en la tabla 11.

Tabla 11. Parámetros de barplot()

Argumento	**Descripción**	**Valor por defecto**
`height`	Vector o matriz principal de datos	No tiene
`width`	Vector con el ancho de las bandas	1
`space`	Cantidad de espacio a la izquierda de cada banda	NULL
`names.arg`	Vector de nombres	NULL
`legend.text`	Leyenda aclaratoria del gráfico	NULL
`horiz`	Valor lógico. Si es FALSE, las barras se dibujan verticalmente, con la primera barra hacia la izquierda. Si es TRUE, las barras se dibujan horizontalmente, la primera en la posición más abajo.	FALSE

Por ejemplo, sea un conjunto de datos que contiene el listado con los nombres de los asistentes a un cierto evento celebrado en 2017, con el tipo de transporte que empleó cada quien para llegar al evento.

La figura 25 muestra el gráfico de barras correspondiente a la cantidad de personas que ha empleado cada tipo de transporte usado en el evento. En el eje de las abscisas se han anotado las categorías (tipo de transporte) y se ha dibujado una barra con una altura correspondiente al número de elementos encontrados en cada categoría (personas). Se ha empleado la función **`table()`**, que toma como argumento un objeto de tipo factor o convertible a factor y devuelve las frecuencias de ocurrencias de los diferentes niveles del factor (ver unidad 4). Dicha figura se obtuvo a partir del siguiente script:

Script de entrada en R

```
participantes <- c("María", "Juan", "Pedro", "Juana", "Rolando",
            "Rosa", "Manuel", "Carmen", "Mario", "Lisbeth","Lisett",
            "Octavio", "Abel", "Gladys", "Katia", "Norma")
transporte <- c("aereo", "terrestre", "tren", "maritimo", "tren",
           "tren", "terrestre", "terrestre", "aereo", "terrestre",
           "maritimo", "terrestre", "aereo", "terrestre",
           "terrestre", "aereo")
info <- data.frame(participantes, transporte)
tt <- table(info$transporte)
barplot(tt)
```

Salida gráfica en R:

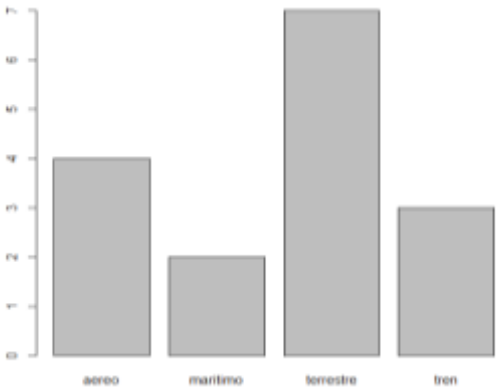


Figura 25. Diagrama de barras.

Si en vez de barras de color gris (color por defecto) se desea otro color, por ejemplo azul, sustituimos la línea de código **`barplot(tt)`** por **`barplot(tt, col="blue")`** y la salida gráfica será:

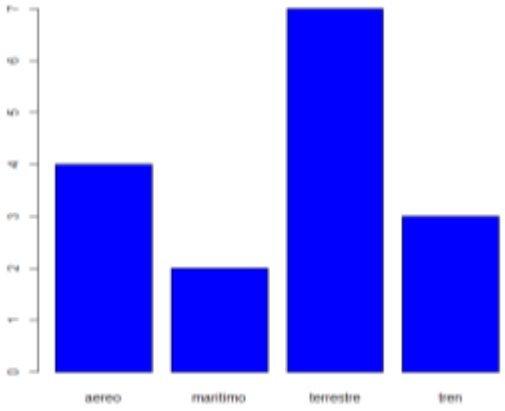


Figura 26. El mismo gráfico de la figura 25 con todas las barras en azul.

Se debe notar el uso del argumento **`col="blue"`**. Por último, si se desean barras de diferentes colores, títulos de las barras verticales y ordenadas por frecuencias, sustituimos la última línea introducida por:

```
color <- c("gray", "orangered", "cyan", "red")
barplot(sort(tt), las=2, col=color)
```

Se obtiene como salida el gráfico de la figura 27:

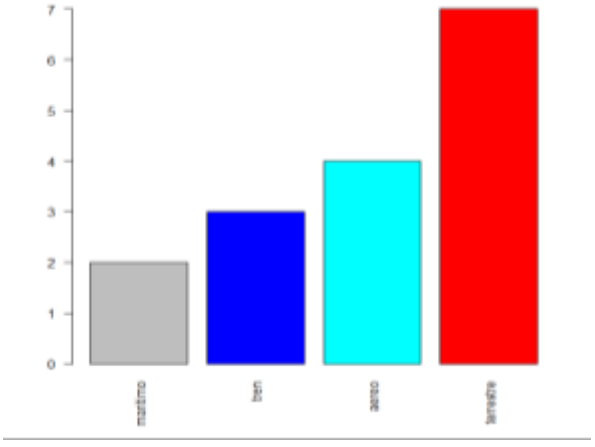


Figura 27. El mismo gráfico de barras, ordenadas las barras por su altura y coloreadas.

Para obtener un diagrama de barras de las frecuencias relativas, ejecute las instrucciones:

```
rr <- tt/sum(tt)

rr

barplot(rr, las=2)
```

Los datos siguientes muestran la frecuencia relativa de empleo del transporte en el evento del 2017:

aéreo	maritimo	terrestre	tren
0.25	0.125	0.4375	0.1875

y la salida gráfica se observa en la figura 28:

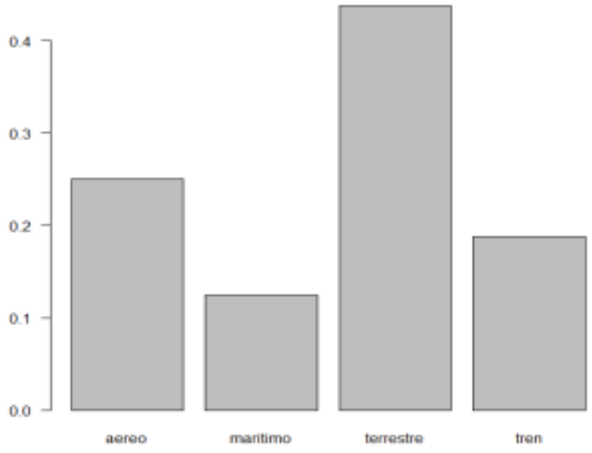


Figura 28. Gráfico de barras de frecuencias relativas de transporte usado en 2017.

10.2.2 Diagrama de barras agrupadas

Una de las características interesantes de los gráficos de barras es que cuando se tienen diversas series de datos sobre un mismo aspecto, se pueden utilizar para hacer comparaciones. Supóngase, para el mismo ejemplo del evento, que en el año 2018 las proporciones de tipo de transporte fueron las siguientes:

aéreo	marítimo	terrestre	tren
0.1200	0.1875	0.4925	0.2000

Lo primero es construir una matriz que contenga todos los datos, los del 2017 y los del 2018. Siguiendo el script anterior, ya en **rr** están guardadas las frecuencias relativas de 2017, por lo que se crea un vector con las frecuencias relativas de 2018 y se adjunta a las de 2017:

Script de entrada en R

```
evento2018 <-c(aereo=0.12,maritimo=0.1875, terrestre=0.4925,
                tren=0.2)
evento2018
rr1 <- rbind(rr, evento2018)
rownames(rr1) <- c("2017","2018") # se dan nombres a las filas
rr1
```

Consola de salida de R

```
> evento2018 <- c(aereo=0.12, maritimo=0.1875, terrestre=0.4925,
                  tren=0.2)
> evento2018
    aereo  marítimo  terrestre      tren
   0.1200    0.1875     0.4925    0.2000
>
> rr1 <- rbind(rr, evento2018)
> rownames(rr1) <- c("2017","2018") # se dan nombres a las filas
> rr1
     aereo maritimo  terrestre   tren
2017  0.25   0.1250     0.4375 0.1875
2018  0.12   0.1875     0.4925 0.2000
```

En este caso el argumento **height** será la matriz **rr1** en vez de un vector, resultando un gráfico de barra agrupado (o apilado como veremos luego). Ahora se empleará además el argumento lógico **besides**, con el siguiente significado:

- **besides=TRUE**: cada columna de la matriz representa un grupo y los valores en cada columna se muestran uno junto al otro.
- **besides=FALSE** (valor por defecto): cada columna de la matriz produce una barra, con los valores de la columna superpuestos en pila (se estudiará en la sección 10.2.3).

Ahora, al agregar y ejecutar al script anterior las líneas de código:

```
barplot(rr1, beside=T, col=c(4,2), las=1)
legend("topleft", legend=rownames(rr1), col=c(1,2), pch=15)
```

se muestra el gráfico de la figura 29:

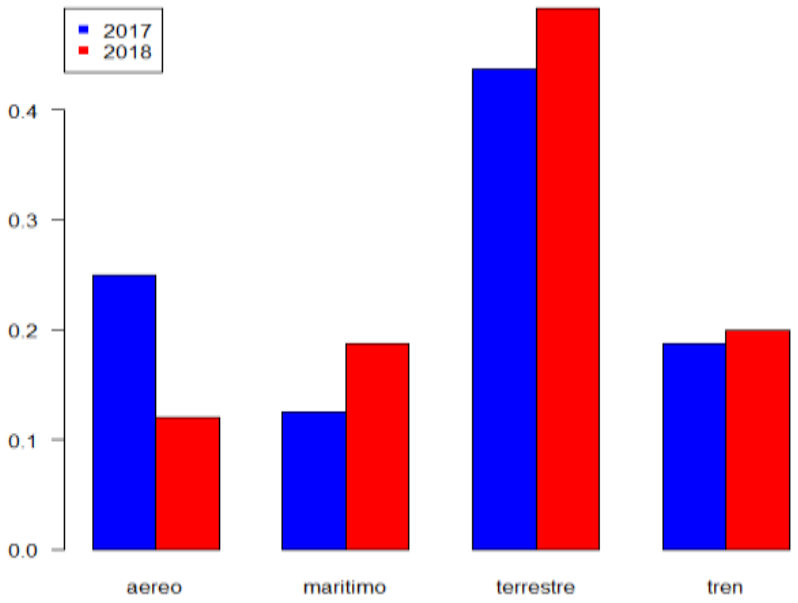


Figura 29. Diagrama de barras agrupadas que compara el uso del transporte en 2017 y 2018.

Se puede observar en la figura 29 que las barras corresponden a las frecuencias del tipo de transporte por años (filas de la matriz `rr1`). ¿**Cómo se procedería si se quisiera mostrar las barras por tipo de transporte**? Se trabajaría con la transpuesta de la matriz `rr1`, se sustituyen las dos líneas anteriores de código por la que siguen y la salida gráfica se muestra en la figura 30.

```
barplot(t(rr1), beside=T, col=1:4)
legend(x=4.1, y=0.48, legend=colnames(rr1), col=1:4, pch=15)
```

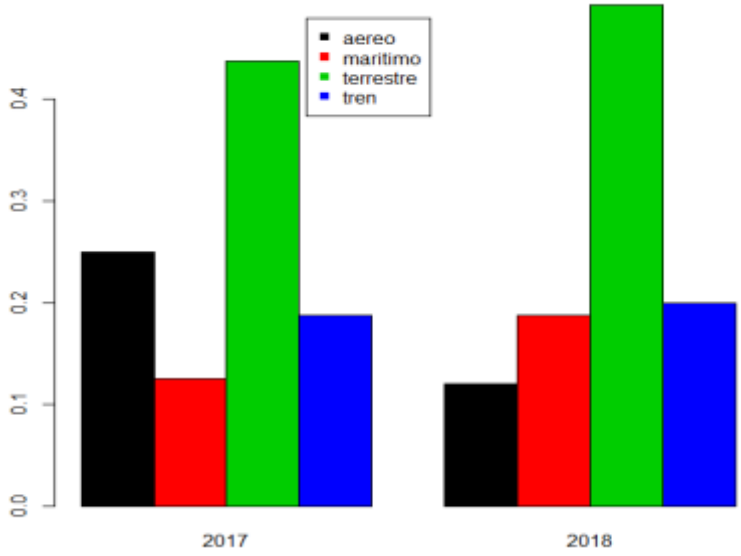


Figura 30. Diagrama de barras agrupadas que compara por tipo de transporte.

En el siguiente ejemplo se muestra un diagrama de barras horizontal. Se toman los datos correspondientes al tipo de transporte (empleado en los scripts anteriores).

Código en R:

```
.........
tt <- table(info$transporte)
barplot(tt,col=c("red","green","blue","violetred"), horiz = T)
```

Gráfico de salida (figura 31):

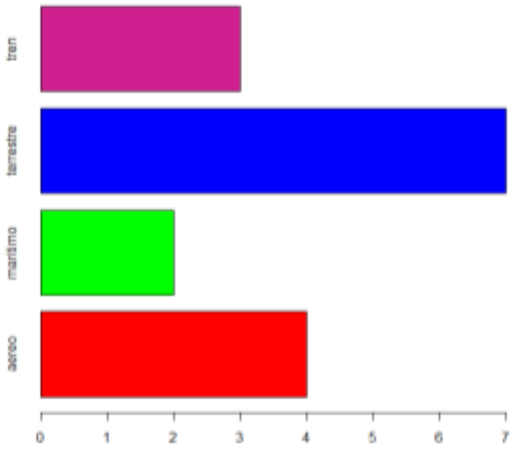


Figura 31. Diagrama de barras agrupado horizontalmente.

10.2.3 Diagrama de barras apiladas

La misma información de los gráficos anteriores se podría obtener con otra apariencia mediante **barras apiladas** o **pilas**. En este caso sólo se debe tener cuidado de agregar espacio para colocar la leyenda. Por omisión, en estos casos, cada grupo de barras ocupa un espacio de 1.2 unidades, contando el espacio entre grupos, que es de 0.2. Se debe tener en cuenta que el desplegado del primer grupo empieza justamente a partir de 0.2.

Script en R

```
barplot(rr1, beside=F, col=c("red","blue"), las=2)
legend("topleft", legend=rownames(rr1), col=c("red","blue"), pch=15)
# El caso de la traspuesta de la matriz. Nótese el argumento xlim
# para dejar espacio para la leyenda
barplot(t(rr1), beside=F, col=2:5, xlim=c(0.2, 3*1.2+0.2))
legend(x=2.5, y=0.6, legend=colnames(rr1), col=2:5, pch=15)
```

Las salidas por la ejecución del código anterior se muestran en las figuras 32 y 33.

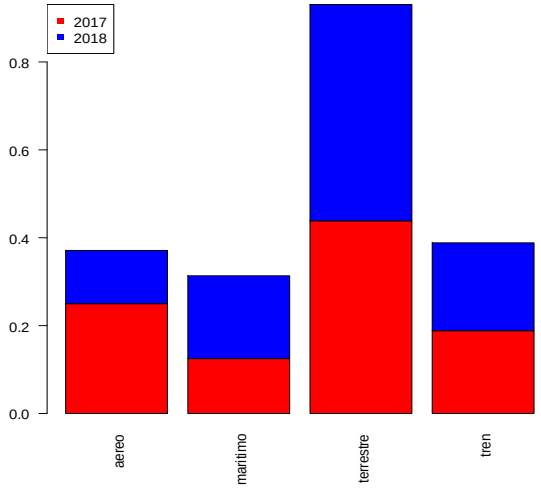


Figura 32. Diagrama de barras apiladas que compara la frecuencia de uso del tipo de transporte.

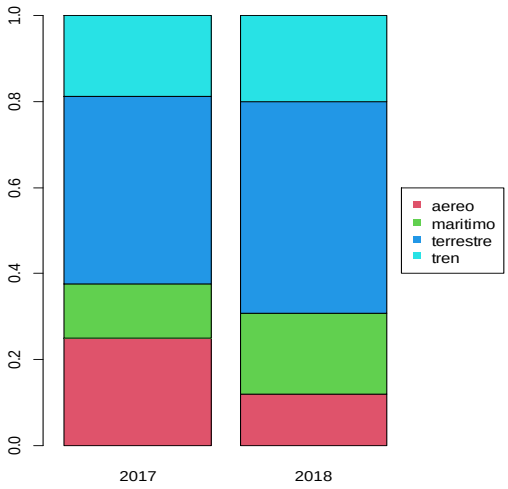


Figura 33. Diagrama de barras apiladas que compara el uso del transporte por año.

10.2.4 Gráficos de pastel (pie)

Una forma de mostrar información de datos cualitativos es representar el conjunto completo de datos como un pastel y las categorías como tajadas del pastel, donde el tamaño de las tajadas representa la proporción del pastel asociada con esa categoría.

Para crear un diagrama de pastel se usa la función **`pie()`**:

```
pie(x, labels, ...)
```

Algunos de sus argumentos son:

- **`x`**: vector numérico no negativo que indica el área de cada tajada
- **`labels`**: vector de caracteres con las etiquetas de cada tajada.
- **`rainbow(n, alpha=k)`**: Crea un vector de **`n`** colores contiguos, con un nivel de transparencia fijado con el parámetro alpha, con valor **`k`** entre 0 y 1.
- **`radius:`** radio del círculo del pastel (por defecto 1).

Para introducir este tipo de gráfico se trabajará con las frecuencias relativas de cada uno de los elementos del factor (tipo de transporte), como se hizo en el epígrafe 10.2.1:

```
participantes <- c("María", "Juan", "Pedro", "Juana", "Rolando",
       "Rosa","Manuel","Carmen","Mario","Lisbeth","Lisett","Octavio",
       "Abel", "Gladys", "Katia", "Norma")
transporte <- c("aereo", "terrestre", "tren", "maritimo", "tren",
       "tren", "terrestre", "terrestre", "aereo", "terrestre",
       "maritimo", "terrestre", "aereo", "terrestre", "terrestre",
       "aereo")
info <- data.frame(participantes, transporte)
tt <- table(info$transporte)
```

Con **`pie(tt, col=rainbow(4, alpha=0.5), radius=1)`** se obtiene el gráfico de la figura 34:

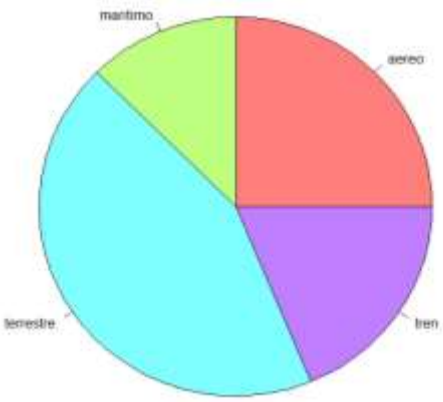


Figura 34. Diagrama de pastel para las frecuencias del tipo de transporte en 2017.

El diagrama se ha construido ubicando las categorías en sentido anti-horario, a partir de las 3:00 (valor por defecto); para cambiar a sentido horario se emplea el argumento **`clockwise`**, lo que se ejemplifica en el siguiente script. Observar que en este caso el inicio es a las 12:00.

```
pie(tt, col=rainbow(4, alpha=0.6), clockwise=T, radius=1)
```

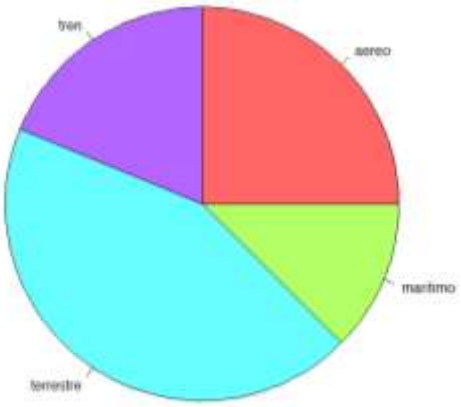


Figura 35. Diagrama de pastel para las frecuencias del tipo de transporte en 2017 (sentido horario).

Si en lugar de mostrar las etiquetas de las categorías, se desea que aparezcan los valores que representan cada una de ellas, se procede como se muestra a continuación, donde se emplean las frecuencias relativas en lugar de las absolutas:

```
tt2<-as.character(rr)
pie(rr, col=rainbow(4,alpha=0.5), clockwise=T, radius=1,
     labels=tt2)
```

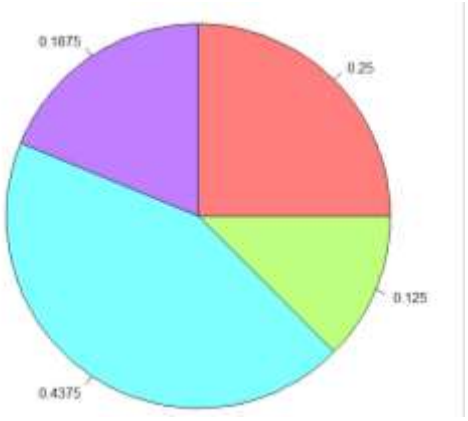


Figura 36. Diagrama de pastel para las frecuencias del tipo de transporte en 2017 (sentido horario y valores de las categorías).

En el siguiente ejemplo se dan las frecuencias de ocurrencia de 4 medidas y se dibuja el gráfico de pastel que aparece en la figura 37, etiquetado con el nombre de cada medida:

```
frec_Medidas <- c(A = 2, B = 10, C = 12, D = 8)
pie(frec_Medidas, main = "Frecuencia_Medidas", radius = 0.8)
```

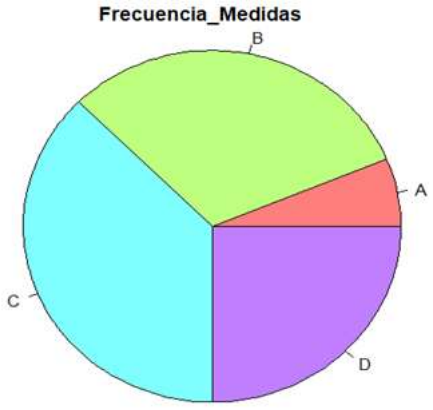


Figura 37. Gráfico de pastel de las frecuencias A, B, C y D.

10.3 Histogramas (función `hist`)

Un histograma es un gráfico que muestra los valores de una variable continua y la frecuencia con que ocurre cada uno de ellos, en cada uno de los intervalos en que se particiona el recorrido de la variable.

A modo de ejemplo, se dan 20 calificaciones en una asignatura dada y se construye el histograma mostrado en la figura 38:

```
# Calificaciones tomadas
z <- c(10,5,10,2,6,8,7,9,3,4,4,9,7,8,6,7,7,8,2,7)
# Histograma
hist(z, freq=TRUE,   # Frecuencias absolutas
        col="blue",  # Color de las barras
        density=5,   # Densidad de sombreado de las barras
        main="Histograma de Calificaciones",  # Título general
        xlab="Calificaciones",   # Etiqueta del eje X
        ylab="Frecuencias Absolutas")   # Etiqueta del eje Y
```

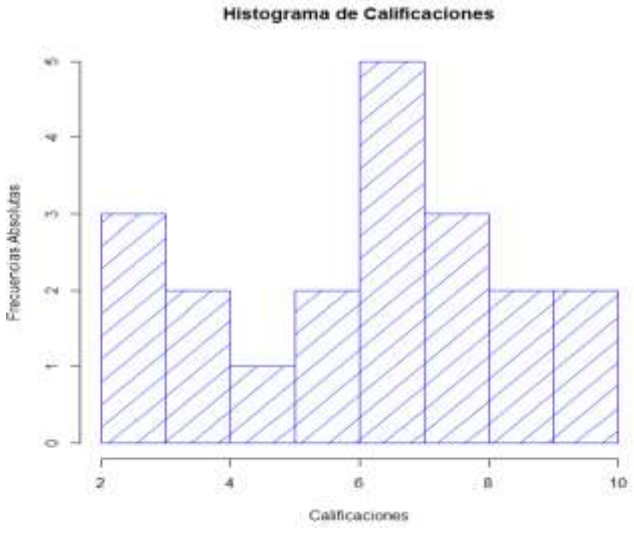


Figura 38. Histograma de calificaciones con frecuencias absolutas.

Al modificar el argumento **`freq = FALSE`** se obtiene un histograma de frecuencias relativas para los mismos datos (figura 39).

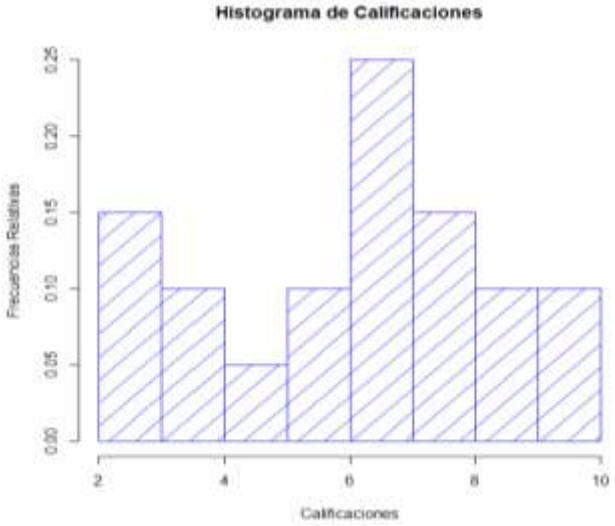


Figura 39. Histograma de calificaciones con frecuencias relativas.

10.4 Diagramas de cajas y bigotes (boxplot)

Este tipo de gráfico se puede obtener de dos formas diferentes: utilizando la función `plot()` y usando la función **`boxplot()`**. La ejemplificación se realiza con el vector de calificaciones.

```
z <- c(10,5,10,2,6,8,7,9,3,4,4,9,7,8,6,7,7,8,2,7)
boxplot(z, main="Diagrama de Cajas y Bigotes")
```

Sale el gráfico de la figura 40:

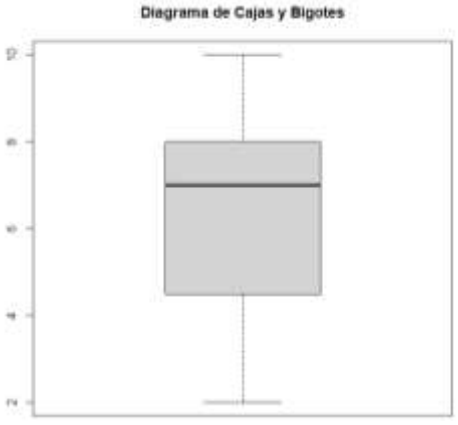


Figura 40. Gráfico de cajas y bigotes.

Recordemos (ver figura 41) las partes de la construcción de este tipo de gráfico:

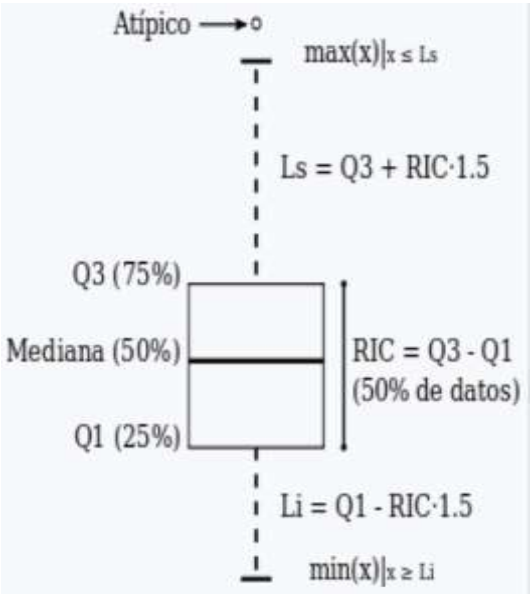


Figura 41. Estructura de un diagrama de cajas y bigotes.

Los círculos pequeños denotan valores muy diferentes a los restantes, llamados outliers o valores atípicos.

En el siguiente ejemplo se construyen tres cajas en el mismo diagrama, correspondientes a una agrupación de la longitud (`Sepal.Length`) de las tres especies florales (variable categórica `Species`) que se cargan del fichero "iris" que trae el R. Este fichero se carga a memoria como un data frame con las cinco columnas:

```
  Sepal.Length Sepal.Width Petal.Length Petal.Width Species
```

El script a ejecutar es:

```
rm(list=ls())
boxplot(Sepal.Length ~ Species, data = iris,
        xlab= "Especies", ylab = "Longitud del Sépalo(cm)",
        main = "Medidas de especies en Iris", boxwex = 0.5)
```

El gráfico de salida se muestra en la figura 42:

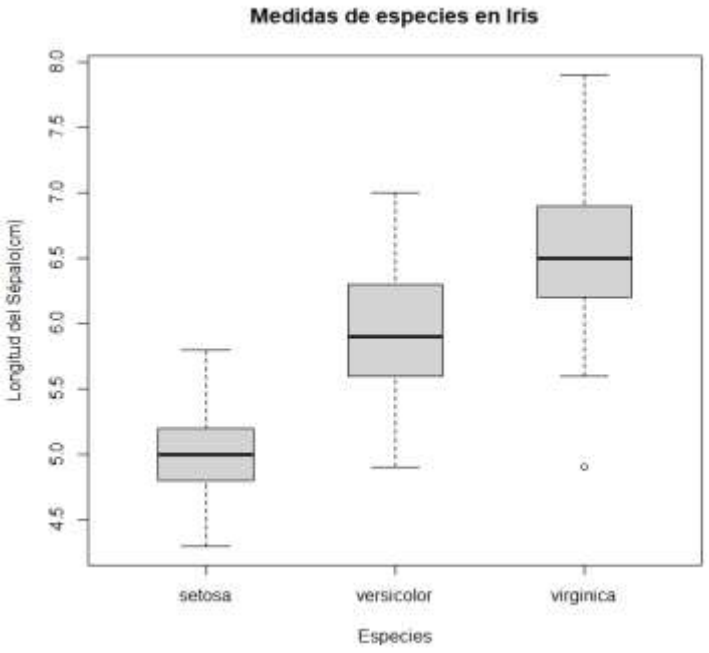


Figura 42. Gráfico de cajas de las especies de flores en la fuente de datos "iris".

Si en lugar de usar la función **`boxplot()`** se usa la función **`plot()`**, se debería de escribir el siguiente script:

```
data(iris)
 fac<-iris$Species
 y <- iris$Sepal.Length
 plot(fac, y, xlab= "Especies", ylab = "Longitud del Sépalo(cm)")
```

cuya salida coincide con la mostrada en la figura 42.

10.5 Gráficos de curvas continuas (función `curve`)

Al inicio de esta unidad se ploteó la función sen(x) con ayuda de la función **`plot()`**, veremos ahora como hacerlo con la función **`curve()`**, la cual dibuja una curva que corresponde a una función dada como primer argumento, en el intervalo especificado por los argumentos (**`from`**, **`to`**).

Script de entrada en R

```
curve(sin(x), from=0, to=4*pi, col="red", lwd=2, xaxt="n")
axis(side=1, xaxp = c(0, 4*pi, 8), pos=c(0,-0.5))
abline (a=0, b=0, col="blue") # Para dibujar línea azul (ver abajo)
```

La función **`abline (a=0, b=0, col="blue")`** traza una línea azul que intercepta al eje Y en 0 (valor de **`a`**) y tiene pendiente 0 (valor de **`b`**).

La función **`axis`** permite dibujar un eje cartesiano junto con etiquetas de valores acompañantes. Con **`axis(side=1, xaxp = c(0, 4*pi, 8), pos=0)`** se indica dibujar un eje de abcisa etiquetado por debajo (`side=1`), marcas de etiquetas indicadas en `xaxp` y que intercepta al eje de las ordenadas en 0 (`pos=0`). Observar que el argumento `xaxt` de `curve()` debe tener el valor ="n", en caso contrario saldrían dos marcas de etiquetas.

El argumento **`xaxp`** es un vector de la forma **`c(x1,x2,n)`**, donde **`x1`** y **`x2`** son las abcisas de los extremos del intervalo de dibujo y **`n`** la cantidad de subintervalos en que se quiere dividir este. De forma similar existe el argumento **`yaxp`** para marcas en el eje de las ordenadas.

La curva se muestra en la figura 43.

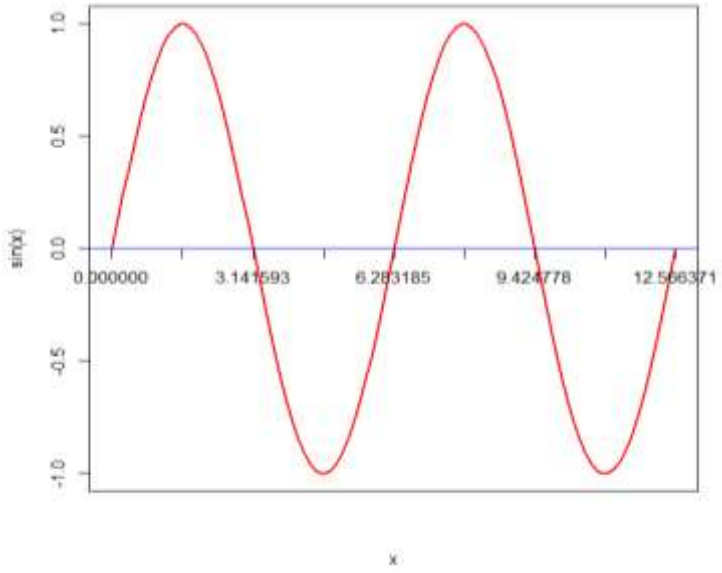


Figura 43. Gráfico de la función seno usando la función curve().

Si se desea enmarcar la figura en una zona rectangular del plano, esta debe ser especificada a través de los parámetros `xlim` y `ylim`, como se muestra en el siguiente script:

Script de entrada en R

```
curve(sin(x), xlim=c(0,2*pi), ylim=c(-1,1), col="red", lwd=2, xaxt="n")
axis(1, xaxp = c(0, 2*pi, 4), pos=0)
abline (a=0, b=0, col="blue") # Para dibujar eje X
```

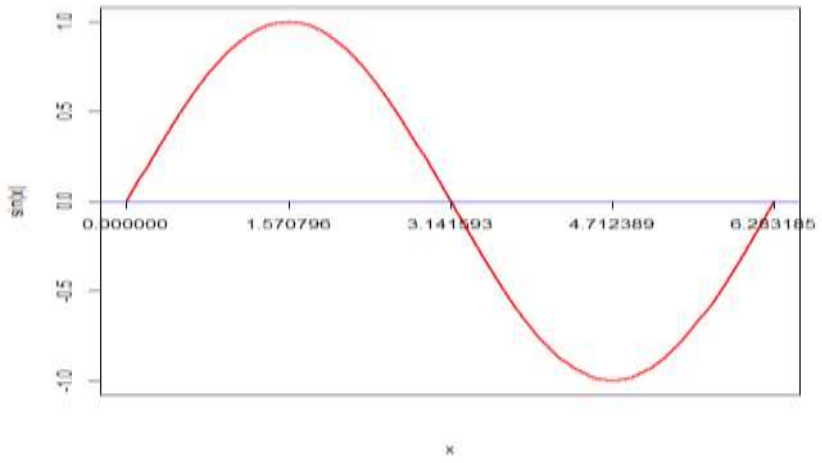


Figura 44. Gráfico de la función seno enmarcado en un rectángulo.

Si se desea resaltar puntos en el gráfico, se puede lograr con la función **`points()`**, la cual se muestra a continuación:

Script de entrada en R

```
curve(sin(x), xlim=c(0,2*pi), ylim=c(-1,1), col="red", lwd=2, axt="n")
abline(a=0, b=0, col="black") # Para dibujar eje X
axis(1, xaxp = c(0, 2*pi, 4), pos=0)
xseno <- seq(0, 2*pi, pi/2)
yseno <- sin(xseno)
points(xseno, yseno, type="p", pch=16, col="blue")  #Plotea los puntos
```

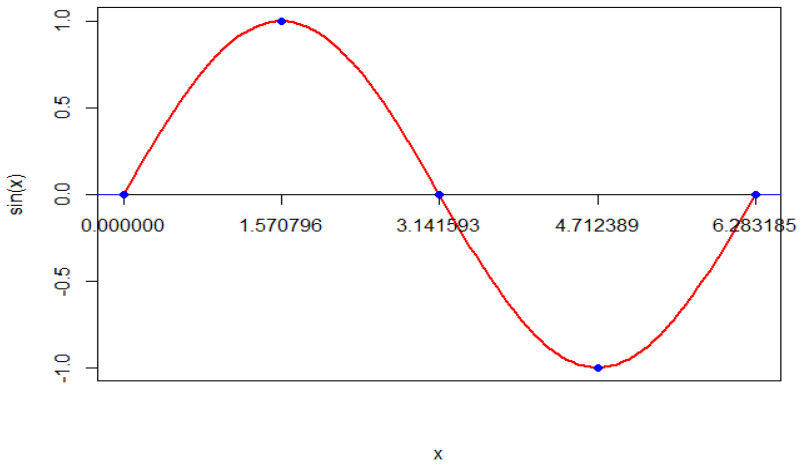


Figura 45. Función seno con puntos resaltados.

Si se desean graficar los puntos y los puntos y líneas entre ellos, esto se haría de la siguiente forma, en la cual se muestran ambos gráficos en una misma ventana con el empleo de la función **`layout()`**:

Script en R

```
# Divide la ventana en una fila y dos columnas
layout(matrix(c(1,2), 1, 2, byrow = TRUE))
layout.show(2)   # Muestra las dos subventanas

# Crea condiciones para plotear los puntos en la ventana izquierda
plot(0:2*pi, -1:1, xlab="x", ylab="y", type = "n")

# Preparando coordenadas de los puntos a mostrar
xseno <- seq(0, 2*pi, pi/2)
yseno <- sin(xseno)

# Plotea los puntos de ventana izquierda
points(xseno,yseno, type="p", pch=16, col="blue")

# Crea condiciones para plotear los puntos la ventana derecha
plot(0:2*pi, -1:1, xlab="x", ylab="y", type = "n")
# Plotea los puntos de la ventana derecha y los une con segmentos
points(xseno,yseno, type="b", pch=16, col="blue")

# Retornando a la pantalla completa
layout(matrix(c(1,1), 1, 1, byrow = TRUE))
```

La salida que se obtiene se muestra en la figura 46:

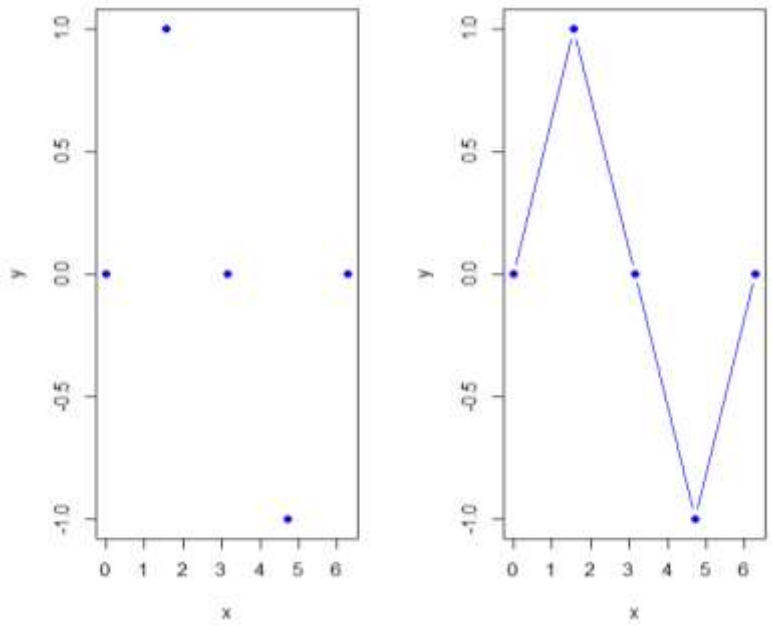


Figura 46. Ploteo de puntos (izquierda), puntos y líneas (derecha).

Otra forma de combinar gráficos, además de `layout()`, es con el empleo de la función `par()`, que posee, entre otros, los argumentos **`mfrow`** y **`mfcol`** para crear una matriz de `nrows` filas y `ncols` columnas con `c(nrows, ncols),` que sirve de cuadriculado para los ploteos. El siguiente ejemplo, basado en el data frame de transporte empleado para elaborar la figura 25, combina dos gráficos en una ventana:

```
......
info <- data.frame(participantes, transporte)
tt <- table(info$transporte)
par(mfrow=c(1,2))
pie(tt, col=rainbow(4, alpha=0.5), radius = 1,
        main="Gráfico de pastel")
barplot(tt,col=c("red","green","blue","violetred"), horiz = T,
        main="Grafico de barras")
```

Saldría la ventana:

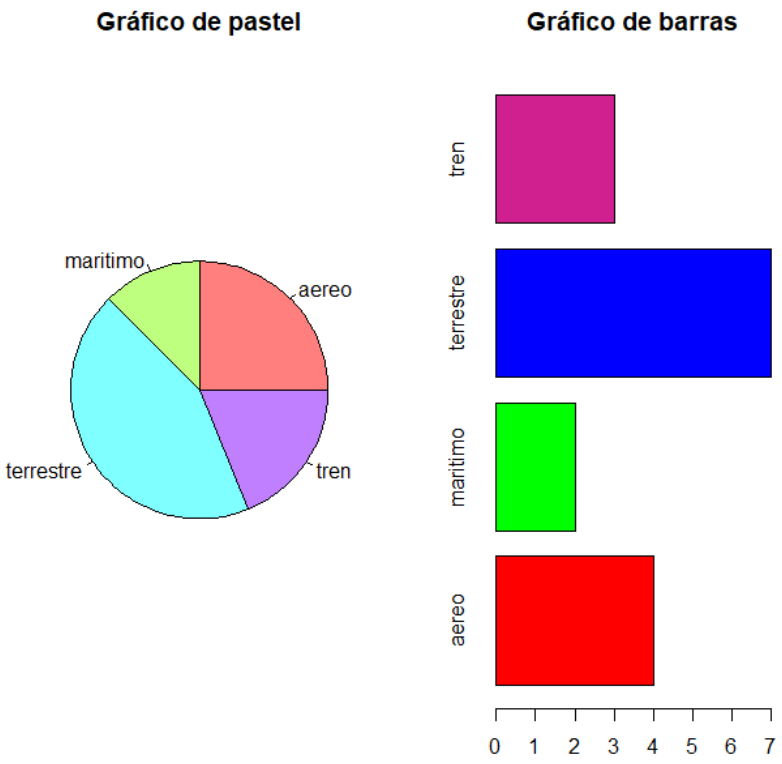


Figura 47. Ploteo de un gráfico de pastel (izquierda), gráfico de barras (derecha).

Para retornar a la pantalla completa:

```
par(mfrow=c(1,1))
```

Esto equivale a `layout(matrix(c(1,1), 1, 1, byrow = TRUE))`.

10.6 Complementos de las figuras

Existen diversas funciones para agregar componentes a la región de ploteo de figuras existentes, entre ellas:

`points(x, y, ...)`: agrega puntos al ploteo actual.

`lines(x, y, ...)`: agrega segmentos de recta al ploteo actual.

`text(x, y, labels, pos, ...)`: agrega etiquetas en la posición (**`x`**, **`y`**) en el gráfico. El argumento **`labels`** es un vector de enteros o caracteres. La etiqueta **`labels[i]`** se plotea

junto al punto (**x[i]**, **y[i]**); en dependencia del valor dado a **pos:** 1 abajo, 2 a la izquierda, 3 arriba y 4 a la derecha. Por defecto **labels** es **1:length(x)**.

abline(a, b, ...): agrega la recta ***y* = *a* + *bx***.

abline(h=y, ...): agrega una línea horizontal de ordenada **y**.

abline(v=x, ...): agrega una línea vertical de abcisa **x**.

polygon(x, y, ...): dibuja un polígono definido por los vértices en (**x**, **y**), según orden de aparición.

segments(x0, y0, x1, y1, ...): dibuja el segmento de recta entre los puntos dados.

arrows(x0, y0, x1, y1, ...): igual al anterior, pero dibuja flechas.

symbols(x, y, circles, squares, rectangles, stars, thermometers, boxplots,...): dibuja círculos, cuadrados, estrellas, rectángulos, termómetros, etc.

legend(x, y, leg, ...) : dibuja un cuadro aclaratorio, con las leyendas en **leg**.

Estas funciones tienen múltiples argumentos opcionales que especifican color, tamaño, y otras características a ser usadas.

10.7 Exportación e impresión de gráficos

Los gráficos que se han desarrollado pueden ser guardados en diversos formatos, para luego ser reusados en disímiles análisis que así lo requieran. Para ello se sigue como esquema el de programación de comandos de impresión o el empleo de posibilidades que brindan algunas GUI´s como RStudio.

Para el primer esquema, supongamos el histograma de 100 valores que siguen una distribución uniforme en el intervalo [1,50]. Al ejecutar:

```
hist(runif(100, min = 1, max = 50))
```

aparecerá una ventana gráfica parecida a:

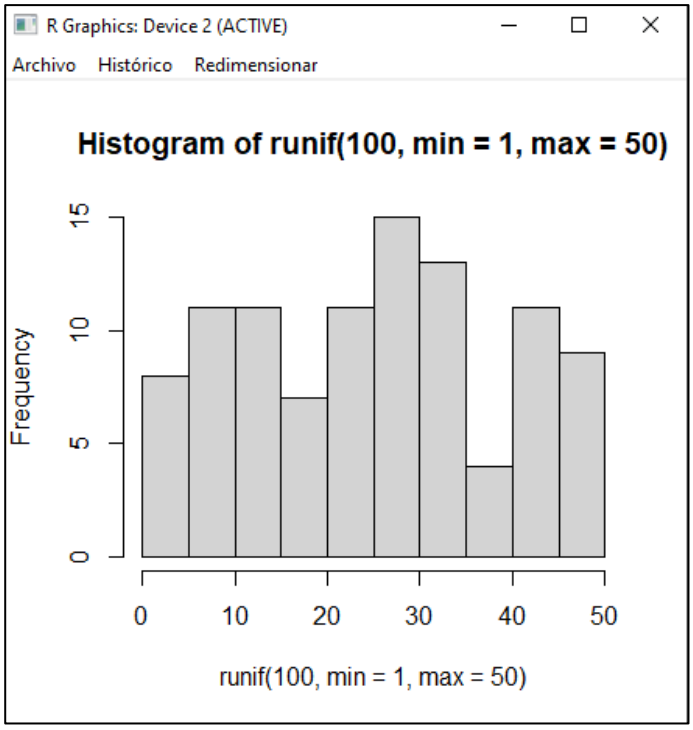


Figura 48. Histograma a guardar.

Una forma de salvar la figura en la ventana gráfica es usando la función:

```
savePlot(filename, type, device, ...)
```

donde:

`filename`: nombre (camino) del fichero donde se guardará la figura.

`type`: tipo de figura ("wmf", "emf", "png", "jpg", "jpeg", "bmp", "tif", "tiff", "ps", "eps", "pdf").

`device`: número de dispositivo de ventana, por defecto el dispositivo actual dev.cur().

`...`: otros parámetros.

Entonces si a las líneas anteriores de código le agregamos:

```
savePlot(filename="miHistograma.png", type="png")
```

se guardará en el directorio de trabajo activo un fichero con nombre `miHistograma.png` que contendrá la figura que aparece en el dispositivo gráfico mostrado arriba.

Existen otras funciones que permiten guardar los gráficos en diversos formatos, como `postscript()`, `pdf()`, `pictex()`, `xfig()`, `bitmap()`, `bmp()`, `jpeg()`, `png()`, `tiff()`. Ellas exigen al menos tres acciones: especificar el nombre del fichero destino y otras características de la impresión, dibujar el gráfico y por último llamar `dev.off(0)`.

```
pdf(file = "f1.pdf", width = 8, height = 10)
hist(runif(100, min = 1, max = 50))
dev.off()
```

El gráfico no aparece en pantalla, pero se puede comprobar que en el directorio de trabajo activo está el fichero f1.pdf con el gráfico solicitado, de apariencia igual al gráfico de la figura 48.

Un gráfico puede ser guardado en varios formatos también, usando el menú del dispositivo gráfico activo. Por ejemplo, en la figura 49 se muestra la secuencia de pasos a seguir para guardar un gráfico. Algunas opciones pueden lograrse también dando clic derecho sobre la figura.

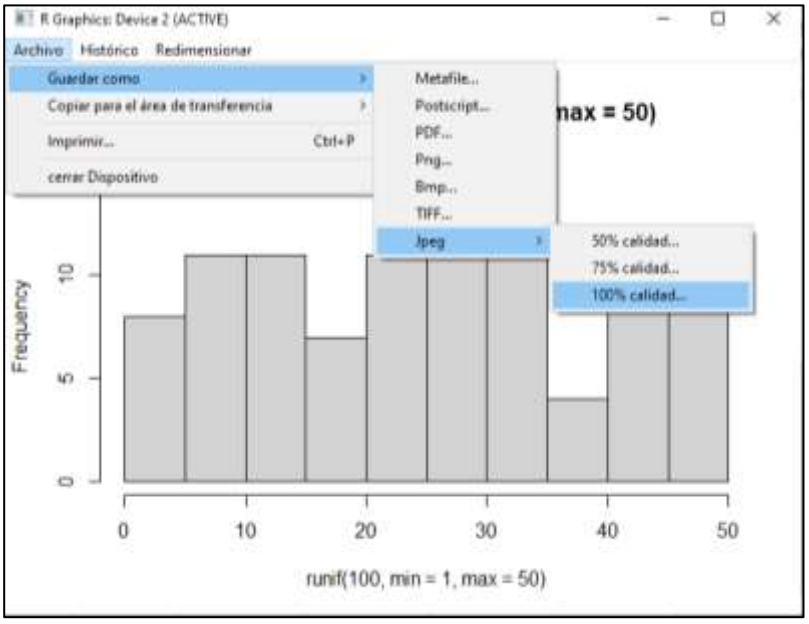


Figura 49. Opciones de salvado en menú del dispositivo actual.

Ejercicios

1. Estudie los símbolos empleados para los puntos y los colores.
2. Genere el fichero "EstudianteEdad.txt" con los datos correspondientes a la edad de 12 estudiantes, los cuales se muestran a continuación:

Estudiante	1	2	3	4	5	6	7	8	9	10	11	12
Edad	4	5	5	6	6	6	7	7	8	9	10	11

   Represente gráficamente la edad mediante un diagrama de barras.
3. Escribir y graficar la función de distribución acumulada de los datos que aparecen en el fichero "EstudianteEdad.txt" que creó en el ejercicio 2.
4. Estudiarse los parámetros de la función `hist()` y crear una función para calcular las marcas de clases y pasar esta función como valor del argumento `breaks`. Con los datos referentes al peso de 50 estudiantes, los cuales se muestran en la siguiente secuencia de datos, construya el histograma para la variable peso, a partir de formar exactamente 6 clases.

71.5	80.0	70.0	81.0	91.0	99.0	85.0	92.0	99.5	91.0
71.0	82.2	101.0	95.0	75.5	120.0	115.0	93.5	92.0	94.0
96.0	86.5	100.0	89.0	95.2	102.0	90.0	98.0	100.0	102.5
87.0	88.0	93.0	89.5	83.0	103.0	94.1	90.7	96.0	105.0
96.0	98.0	108.0	99.0	100.0	109.0	99.0	100.0	99.0	110.0

5. Dibujar la función:

$$f(x) = \begin{cases} 3x + 2 & x \leq 3 \\ 2x - 0.5x^2 & x > 3 \end{cases}$$

   en el intervalo [0, 6].
6. Para los datos del fichero "iris" del sistema base de R, represente gráficamente el ancho del sépalo contra su longitud.

7. Los datos siguientes corresponden al tiempo necesario para procesar 25 trabajos en una CPU. Representar gráficamente estos datos mediante un histograma de frecuencias relativas.

1.17	1.61	1.16	1.38	3.53	0.15	2.41	0.71	0.02	1.59	0.92	0.75	2.59
1.23	3.76	1.94	0.96	4.75	0.19	0.82	0.47	2.16	2.01	3.07	1.40	

8. La siguiente tabla contiene datos sobre adicción alcohólica de mujeres y hombres. Expréselos en un diagrama de barras agrupado y en otro apilado.

	Hombre	Mujer
Nunca bebe	43	92
Bebe ocasionalmente	49	14
Muy bebedor	24	6

9. Construya un diagrama de cajas y bigotes con los datos del ejercicio 7.
10. Cada puntuación en el conjunto siguiente se clasifica en los intervalos 60-69, 70-79, 80-89 y 90-100. Dibuje un histograma para las puntuaciones suministradas, en los intervalos especificados.

    74 89 80 93 64 67 72 70 66 85 89 81 81 71 74 82 85 63 72 81 81 95 84

    81 80 70 69 66 60 83 85 98 84 68 90 82 69 72 81 88

11. Para los siguientes datos dibuje un diagrama de cajas e interprételo:

    40 52 55 60 70 75 85 85 90 90 92 94 94 95 98 100 115 125 125

12. A cada uno de los 40 estudiantes de un grupo se le preguntó su deporte favorito, he aquí sus respuestas:

    Beisbol; Ninguno; Beisbol; Futbol; Beisbol; Futbol; Futbol; Ninguno; Futbol; Baloncesto; Baloncesto; Futbol; Baloncesto; Beisbol; Futbol; Futbol; Beisbol;

Ajedrez; Futbol; Futbol; Beisbol; Ajedrez; Baloncesto; Volibol; Ajedrez; Baloncesto; Futbol; Ninguno; Futbol; Voleibol; Futbol; Beisbol; Baloncesto; Futbol; Futbol; Futbol; Futbol; Beisbol; Beisbol; Futbol.

a) Dibuje un diagrama de pastel de frecuencias absolutas y otro de frecuencias relativas.

b) Dibuje un diagrama de barras.

Unidad 11. Otros temas de interés en la programación con R

11.1 Paquetes en R

Los **paquetes de R** son colecciones de funciones y conjuntos de datos encapsulados, que permiten incrementar y mejorar lo existente en el sistema base. Normalmente se acompañan con documentación del paquete y de sus componentes.

La información básica de un paquete se brinda en su fichero **DESCRIPTION**, donde se resume que hace el paquete, sus autores, fecha de elaboración, versión, tipo de licencia para su empleo y dependencias del paquete (necesidades de otros paquetes para que pueda funcionar).

Un **repositorio** es un lugar donde se encuentran alojados paquetes, y le permiten a un usuario su instalación. Aunque una institución puede tener su propio repositorio local, es común que se mantengan en línea con acceso a cualquiera. Algunos de los repositorios más populares de paquetes de R son:

- CRAN: es el reposotorio official de paquetes de R. Constituye una red de servidores de web y ftp mantenida por la comunidad R de todo el mundo y coordinada por la fundación R. Para que un paquete sea publicado aquí debe pasar varias pruebas que aseguren que el paquete sigue las políticas de la CRAN.
- Bioconductor: es un repositorio específico para código abierto en bioinformática. Al igual que la CRAN, tiene sus propios procesos de admisión y revisión y tiene una comunidad muy activa, que celebra anualmente varias reuniones y conferencias.
- Github: no es un repositorio específico de R, pero es uno de los más populares para proyectos de código abierto. Da buenas facilidades de compartir y colaborar con otros desarrolladores. No posee procesos de revisión.

Al iniciarse el programa R se cargan por defecto ciertos paquetes básicos. Con el comando **`search()`** se desplegará una lista con todos los paquetes cargados. Sin embargo, a veces es necesario cargar otros paquetes para realizar ciertos análisis, o llamar a algunas funciones "no básicas". Esto se hace a través del comando **`library(`**`nombre del paquete`**`)`**.

Algunos de los paquetes básicos (se cargan por defecto) son:

- **base**: contiene las funciones básicas de R.
- **stats**: paquete para realizar pruebas estadísticas clásicas.
- **graphics**: paquete gráfico de R
- **grDevices**: dispositivos gráficos de R y soporte para puntos y fuentes.
- **methods**: clases y métodos formales.
- **utils**: paquete de herramientas de R.

Algunos paquetes específicos (requieren instalación previa) que acompañan a la instalación de R son:

- **datasets**: paquete con conjuntos de datos de R.
- **boot**: funciones bootstrap.
- **cluster**: funciones para análisis de clúster extendido.
- **parallel**: soporte para la computación paralela en R.
- **foreign**: funciones que facilitan la lectura de datos creados en SAS, Minitab, SPSS, etc.

Una **librería** es una carpeta en la memoria secundaria. Cuando se instala R se crea una librería del sistema (carpeta con las carpetas de cada librería).

11.1.1 Instalación de paquetes

Una vía para instalar paquetes en R es usando el repositorio CRAN, conocido el nombre del paquete, usando el comando **`install.packages("nombre.paquete")`**. Varios paquetes

pueden ser instalados a la vez, pasando como parámetro un vector de nombres de paquetes, por ejemplo, `install.packages(c("foreign", "MASS"))`.

Otras operaciones de manipulación de paquetes son:

- Busca o recupera los detalles de los paquetes instalados: **`installed.packages()`**.
- Desinstalar un paquete: **`remove.packages("nombre.paquete")`**.
- Chequear cuales paquetes requieren ser actualizados: **`old.packages()`**.
- Actualizar un paquete: **`update.packages()`**.

También puede usarse una interfaz usuaria gráfica para instalar paquetes, como RStudio o RGui. En RStudio se encuentra en la opción **Tools → Install Package**, apareciendo una ventana (ver figura 50) donde se escoge el tipo de paquete a instalar:

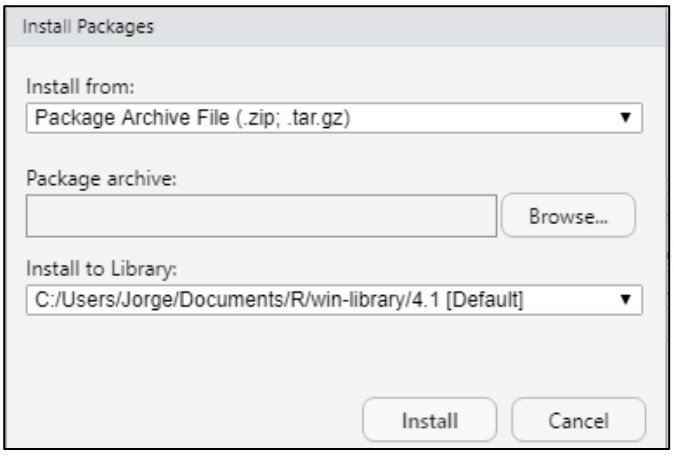


Figura 50. Ventana para seleccionar el tipo de paquete a instalar.

Usando RGui se encuentra en el menú “Packages” (ver figura 51).

El comando usado en el código de programa para cargar un paquete es **`library()`**, que refiere al lugar donde el paquete se encuentra.

Si se desea cargar el paquete **`foreign`** se debe escribir en el script: `library(foreign).`

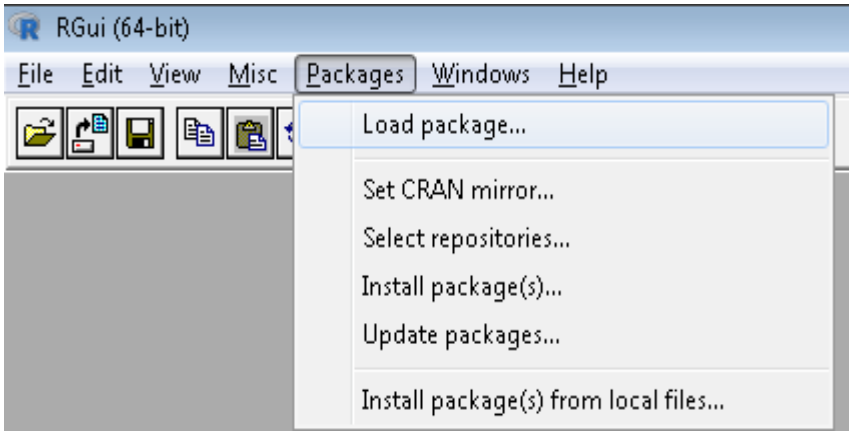


Figura 51. Ventana para seleccionar el tipo de paquete a instalar.

11.1.2 Elaboración de paquetes en R

Se usará como ejemplo explicativo la elaboración de un paquete con nombre `ExpRep` dentro de la carpeta `Experiment_Repetitions` de la torre `E:`, o sea, en el camino `E:/Experiment_Repetitions`, compuesto por las funciones `"Poisson_Theorem.R"`, `"Local_Theorem.R"` e `"Integral_Theorem.R"`.

Resulta necesario que los ejecutables de R y RTools sean accesibles a través de la variable de entorno **path**.

Para situar la carpeta donde se creará el paquete se ejecuta el comando R:

```
setwd("E:/Experiment_Repetitions")
```

Para crear el paquete se usa la función **`package.skeleton()`**, cuyos principales parámetros son:

```
package.skeleton(name="ExpRep", path="e:/Experiment_Repetitions",
  code_files=c("Poisson_Theorem.R","Local_Theorem.R","Integral_Th
  eorem.R"))
```

Note el empleo del parámetro **`code_files`** para señalar cuales archivos integrarán el paquete, los cuales se escribirán con su extensión **.R** y estarán en el camino determinado por el valor del parámetro path.

Al ejecutar `package.skeleton()`, se crea una estructura de paquete dentro de la carpeta **ExpRep** con el siguiente contenido:

- Carpeta man: con los esquemas de documentación de cada función y del paquete en general, en ficheros con extensión **.Rd**.
- Carpeta R: código fuente de cada función dentro del paquete.
- Fichero DESCRIPTION: credenciales generales del paquete.
- Fichero NAMESPACE: contiene información acerca de funciones importadas y exportadas. Este fichero no se modificará manualmente.
- Fichero Read-and-delete-me, con indicaciones que una vez leídas puede ser borrado.

Los ficheros **.Rd** deben ser modificados para dar al paquete y sus funciones la documentación adecuada.

Para construir el paquete se emplea el comando **`R CMD build nombre.paquete`** en una línea del editor de comandos del sistema (cmd). Para nuestro ejemplo:

```
e:\Experiment_Repetitions>R CMD build ExpRep
```

con lo cual se genera el fichero compactado **ExpRep_1.0.tar.gz**.

Para el chequeo del paquete con vista a subirlo a la CRAN se emplea el comando **`R CMD check nombre.fichero.compactado`**:

```
e:\Experiment_Repetitions>R CMD check ExpRep_1.0.tar.gz
```

Con ello se crea la carpeta **ExpRep.Rcheck** que recoge, entre otras, la bitácora de chequeo, manual de usuario del paquete y sus funciones (**ExpRep-manual.pdf**), escrito con resultados de ejemplos corridos, etc.

Para instalarlo:

- si se trabaja directamente sobre el R:

```
e:\Experiment_Repetitions>R CMD INSTALL ExpRep_1.0.tar.gz
```

- Si se trabaja con R Studio se emplea la opción **Tools → Install Package**, explicada en el epígrafe 11.1.1.

Una vez instalado el paquete se reinicia R usando la opción **Session → Restart R**. Ya una vez situado en un programa que necesite el paquete se escribe `library(ExpRep)`.

11.2 Objetos de tipo expression y ejecución de código en formato de cadena

Una expresión en R puede ser vista como código R (incluso dispuesto en varias líneas) que puede ser ejecutado por el intérprete de R. De hecho, todos los comandos válidos son expresiones; cuando se escribe un comando directamente en el teclado, este es evaluado por R y ejecutado si es válido.

Obsérvese esto con más detalle. En ocasiones se desea ejecutar código que está almacenado en formato de cadena de caracteres:

```
# Cadena cad
cad <- "1+1"
```

Pero una cadena de caracteres cualquiera no es un objeto de tipo `expression`:

```
# Una cadena no es una expresión
is.expression(cad)
[1] FALSE
```

Por tanto, su evaluación con la función `eval()` resulta en la propia cadena:

```
eval(cad)
[1] "1+1"
```

¿Cómo solucionar la ejecución de código en formato cadena? Una posibilidad es usar la función **`parse()`**, la cual retorna un texto analizado sintacticamente, pero no evaluado, convertido a un objeto de tipo `expression,` como se muestra en una versión simplificada:

```
# parse convierte cadenas en expresiones
parsed.cad <- parse(text="1+1")
is.expression(parsed.cad)
[1] TRUE
```

Una vez construida una expresión, puede ser evaluada, empleando la función `eval()`.

```
eval(parsed.cad)
[1] 2
```

Una versión, sin todos sus parámetros, de la función `parse()`, es:

`parse(text = NULL, …)`

donde:

`text`: vector de caracteres, que es el texto a analizar sintácticamente.

`...`: otros parámetros.

Otro ejemplo:

```
for (i in 1:3)
{   eval(parse(text=paste("a",i," <- i",sep="")))
}
```

El script anterior crea en cada iteración del ciclo una expresión consistente en una variable formada por la letra `"a"` y el valor de la variable de control del ciclo i, a la que se asigna el propio valor de i. Si luego se ejecuta el fragmento de código:

```
cat("Valor asignado a a1: ",a1, '\n')
cat("Valor asignado a a2: ",a2, '\n')
cat("Valor asignado a a3: ",a3, '\n')
```

sale como resultado:

```
> cat("Valor asignado a a1: ",a1, '\n')
Valor asignado a a1:  1
> cat("Valor asignado a a2: ",a2, '\n')
Valor asignado a a2:  2
> cat("Valor asignado a a3: ",a3, '\n')
Valor asignado a a3:  3
```

Existe también la función **`str2expression(s)`**, donde **`s`** es un vector de caracteres, que es una versión simplificada equivalente a `parse(text = s, keep.source=FALSE)`, que devuelve un objeto de tipo expression.

Con ello la conversión en expresión de "1+1" y su evaluación se puede expresar de las dos formas siguientes:

```
parsed.cad <- str2expression("1+1")
is.expression(parsed.cad)
eval(parsed.cad)
```

```
eval(str2expression("1+1"))
```

Es posible capturar por acotación cualquier expresión que sea sintácticamente correcta, dando como argumentos a la función `quote()` expresiones, incluso con variables indefinidas:

```
call2 <- quote(sin(x))

call2

[1] sin(x)
```

En `call2`, `sin(x)` usa una variable indefinida x. Si se trata de evaluar directamente, da error:

```
eval(call2)

[1] Error in eval(expr, envir, enclos): object 'x' not found
```

Este error es similar a ejecutar sin(x) sin definir antes a x:

```
sin(x)

[1] Error in eval(expr, envir, enclos): object 'x' not found
```

Pero al usar `eval()` es posible proporcionar una lista para evaluar la expresión dada:

```
eval(call2, list(x = 1))
```

Puede ser útil construir una expresión sin evaluarla, usando la función **`expression()`**, la que luego es posible evaluar con `eval()`, lo que es mostrado en el siguiente script:

Script de entrada en R

```
exp1 <- expression(1 + 0:9)
exp1
mode(exp1)
length(exp1)
eval(exp1)
```

Consola de salida en R

```
> exp1 <- expression(1 + 0:9)
> exp1
expression(1 + 0:9)
> mode(exp1)
[1] "expression"
> length(exp1)
[1] 1
> eval(exp1)
[1]  1  2  3  4  5  6  7  8  9 10
```

La función `expression()` permite como argumento una secuencia de expresiones consecutivas, cada expresión corresponde a un argumento efectivo en el llamado de la función. Por ejemplo, al ejecutar cualesquiera de los dos fragmentos de cógidos siguientes equivalentes:

```
exp4 <- expression({
  print("linea1")
  print("linea2")
})
eval(exp4)
exp5 <- expression(print("linea1"), print("linea2"))
eval(exp5)
```

da el resultado:

```
[1] "linea1"
[1] "linea2"
```

11.3 Uso de Rcpp

Escribir programas en R que interactúan con código escrito en C++ constituye una vía de acelerar la ejecución de esos programas. El código C++ normalmente corre muy rápido, ya que se compila con instrucciones a bajo nivel y por tanto más cerca del nivel de hardware que un lenguaje de scripting como R.

Tal interacción se logra usando el paquete **Rcpp**, que proporciona una API (Application Program Interface) que permite escribir código con alto rendimiento, preservando el poder de la manipulación de datos en R, o reusar código C++ ya existente en proyectos de R.

Para usar Rcpp hay que asegurar que el sistema quede preparado para ejecutar código nativo, por lo que en Windows resulta necesario tener cargado el programa **RTools**, que puede descargarse del sitio oficial de la CRAN. Luego se procede a la instalación del paquete Rcpp, lo mismo usando el comando `install.packages()` o usando las posibilidades que brindan al usuario RGui o RStudio, explicadas en el epígrafe 11.1.1.

A modo de ejemplo escribimos el script:

```
library(Rcpp)
cppFunction('
   double suma_cpp(double x, double y)
   {  double value = x + y;
      return value;
   }
')
suma_cpp(1, 2)
```

cuya ejecución resultaría (solo la llamada de la función suma_cpp):

```
> suma_cpp(1, 2)
[1] 3
```

La función `cppFunction()` es adecuada para ejemplos pequeños. Sin embargo, constituye una mejor práctica de programación tener el codigo C++ en fichero separado con extensión `.cpp` y usar la función **`sourceCpp("camino_fichero_cpp")`**, donde **`camino_fichero_cpp`** es una cadena de caracteres con el camino absoluto o relativo al directorio activo donde encontrar el fichero .cpp a ejecutar.

Para el ejemplo anterior, debemos llevar el código siguiente a un nuevo fichero .cpp.

```
double suma_cpp(double x, double y)
{   double value = x + y;
    return value;
}
```

Este fichero debe sufrir algunas modificaciones:

a) Se debe incluir el fichero **`Rcpp.h`** que da acceso a las funciones presentes en Rcpp, mediante `#include <Rcpp.h>`.

b) El acceso a cualquier función f en Rcpp se logra con la calificación **`Rcpp::f`**. Para tener que evitar el uso continuo del calificador **`Rcpp::`**, basta escribir una vez:

```
using namespace Rcpp;
```

c) Señalizar las funciones que se exportan/usan en R, poniendo como comentario la etiqueta:

```
// [[Rcpp::export]]
```

que proporciona una forma fácil de manipular posibles conflictos de nombres.

Entonces el fichero modificado .cpp correspondiente al ejemplo (`add_cpp1.cpp`) quedaría como:

```
#include <Rcpp.h>
using namespace Rcpp;
// [[Rcpp::export]]
double suma_cpp(double x, double y)
{   double value = x + y;
    return value;
}
```

Un ejemplo de programa en R que usa la función suma_cpp es:

```
rm(list=ls())
library(Rcpp)
sourceCpp("add_cpp1.cpp")
suma_cpp(1, 2)
```

Se ha considerado que el fichero `add_cpp1.cpp` se encuentra en el directorio de trabajo de trabajo actual.

Beneficios adicionales de la separación de ficheros C++ radican en las buenas posibilidades de edición de código C++ que brinda RStudio (**File → New File → C++ File**) que facilita la escritura de fichero modificado de C++ y diagnosticar errores con la opción **Code → Show Diagnostics**.

Bibliografía

Aguilar-Fernández, E. y Andrey-Zamora, J. (2020). Introducción a la estadística descriptiva con R. Editorial Universidad Nacional de Costa Rica.

Braun, W. J. y Murdoch, D. J. (2007). *A First Course in Statistical Programming with R*. Cambridge University Press.

Contreras-García, J.M., Molina-Portillo, E. y Arteaga-Cezón, P. (2010). *Introducción a la programación estadística con R para profesores.* Grupo de Educación Estadística, Universidad de Navarra.

Deitel, P. y Deitel, H. (2017). *C++ How to Program*.10th Edition. Pearson Education Limited.

Gallic, E. (2015). *Logiciel R et programmation. Notes de cours.* Faculté des Sciences Économiques de l'Université de Rennes 1, France.

Lafaye de Micheaux, P., Drouilhet, R. y Liquet, B. (2011). *Le logiciel R*. Springer-Verlag France.

Lander, J. P. (2017). *R for Everyone - Advanced Analytics and Graphics.* Second Edition. Pearson Education, Inc.

Le Roux, N. y Lubbe, S. (2015). *A step by step R tutorial: An introduction into R, applications and programming*. First edition. Free eBook at http://bookboon.com

Matloff, N. S. (2011). *The art of R programming: tour of statistical software design*. No Starch Press, Inc.

Mora, W. (2016). Cómo utilizar R en métodos numéricos. *Revista digital Matemática, Educación e Internet, 16*(2), 1-74. http://tecdigital.tec.ac.cr/revistamatematica.

Mc Grath, R. (2018). *R for Data Analysis*. In Easy Step Limited.

Peng, R. D. (2015). *R Programming for Data Science.* Leanpub.

Perlin, M.S. (2021). *Analyzing Financial and Economic Data with R.* Online edition. https://www.msperlin.com/afedR

Ren, K. (2016). *Learning R Programming*. Packt Publishing.

Rowe, B. (2019). *Modeling Data with Functional Programming in R, Volume 1*. CRC Press.

Santana, J. S. y Farfán, E. M. (2014). *El arte de programar en R: un lenguaje para la estadística*. Instituto Mexicano de Tecnología del Agua. UNESCO. Comité Nacional Mexicano del Programa Hidrológico Internacional.

Schmuller, J. (2018). *R Projects for Dummies*. John Wiley & Sons.

Stack Overflow Documentation (2018 compiled). *R Notes for Professionals*. https://goalkicker.com/RBook

The R Core Team. (2020). *R: A Language and Environment for Statistical Computing Reference Index.* Version 4.0.2.

Venables, W. N., Smith D. M. & the R Core Team. (2020). *An Introduction to R*. Version 4.0.2.

Wickham, H. (2019): *Advanced R*. CRC Press. http://adv-r.had.co.nz

Este libro ha sido concebido para ser utilizado por estudiantes de las carreras de Licenciatura en Matemática, Ciencia de la Computación y afines. Enseña a desarrollar software basado en el sistema base del Lenguaje R y al mismo tiempo brinda herramientas metodológicas para poder extender las potencialidades del lenguaje con el empleo de los múltiples paquetes de datos que se han desarrollado para el mismo y que están a disposición de forma libre en repositorios mundiales. Con un carácter didáctico y detallado en las explicaciones y las soluciones a los ejemplos, puede ser usado como material de consulta en la docencia o investigación en otras disciplinas donde sea necesario el desarrollo de programas basado en big data, su análisis y visualización.

ISBN: 978-959-207-734-8

9 789592 077348

Ediciones UO