Enrique Baquela - Andrés Redchuk

Optimización Matemática con R. Volumen I: Introducción al modelado y resolución de problemas

Optimización Matemática con R

Volumen I: Introducción al modelado y resolución de problemas

Enrique Gabriel Baquela

Andrés Redchuk

Madrid, Marzo de 2013

1ª edición, Mayo de 2013
ISBN: 978-84-686-3748-8
Editor Bubok Publishing S.L.
Impreso en España / Printed in Spain

Dedicatorias

A Clementina Augusta, por existir.
A Flavia Sabrina, por ayudarme a ser lo que soy.
A mis padres y hermanos, por estar siempre.
A Isidoro Marín, Horacio Rojo,
Silvia Ramos y Gloria Trovato,
por el empujón inicial.
A mi directora de Doctorado, Ana Carolina Olivera,
y a mi codirectora, Marta Liliana Cerrano
por aceptar el desafío de ayudarme
a formarme en investigación.
A mis amigos y compañeros del GISOI,
por apoyarme desde el principio.
A Juan Cardinalli, por sus apreciaciones
diarias dentro del trabajo de Furgens Research.
A Andrés, por animarme y ser mi socio
en esta aventura.

Enrique Gabriel Baquela

A las tres hijas que Rita me dio
Sophie, Vicky, Érika,
y a los tres ahijados que la vida regaló
Leo, Fede y Alejo.

Andrés Redchuk

Autores

Enrique Gabriel Baquela

Enrique Gabriel Baquela es Ingeniero Industrial por la Universidad Tecnológica Nacional de San Nicolás de los Arroyos (UTN-FRSN), y alumno de Doctorado en la Universidad Nacional de Rosario (UNR).

Es coordinador del Grupo de Investigación en Simulación y Optimización Industrial (GISOI), investigador del departamento de Ingeniería Industrial de UTN-FRSN, profesor de Simulación de procesos y sistemas y de Optimización Aplicada en el mismo centro, y co-fundador de Furgens Research.

Andrés Redchuk

Andrés Redchuk es Ingeniero Industrial por la Universidad de Lomas de Zamora en Buenos Aires, Master en Calidad Total y en Ingeniería Matemática de la Universidad Carlos III de Madrid; Master en Tecnologías de la Información y Sistemas Informáticos y Doctor en Informática y Modelización Matemática de la Universidad Rey Juan Carlos en España.

Andrés Redchuk es Master Black Belt de la Asociación Española para la Calidad, profesor de la Facultad de Ciencias Empresariales de la Universidad Autónoma de Chile, Profesor de la Facultad de Ingeniería de la Universidad Nacional de Lomas de Zamora, miembro del departamento de Estadística e Investigación Operativa de la Universidad Rey Juan Carlos en Madrid, España y Director del Grupo de Investigación en Lean Six Sigma.

Prologo

Los problemas del mundo real son, en general, no lineales enteros mixtos y de gran dimensión. Poder resolver esa clase de problemas es de vital importancia para todo profesional. Encontrar la solución óptima a estos problemas suele ser nuestro principal objetivo, aunque muchas veces nos alcanza con encontrar soluciones con ciertas características, aunque se trate de un resultado subóptimo y conocer la "calidad" de esta solución.

Es conveniente tener disponibles algoritmos eficientes y robustos para la resolución es esa clase de problemas.

El software R, ha demostrado ser muy eficiente para programar e implementar las técnicas estadísticas y los algoritmos de Optimización más tradicionales.

Mediante este libro pretendemos divulgar el desarrollo y la programación de los algoritmos más tradicionales y conocidos de optimización. Además presentamos su implementación en R. Para encontrar soluciones óptimas desarrollamos algoritmos de optimización y para encontrar soluciones cuasi óptimas desarrollamos algoritmos metaheurísticos.

El libro está pensado para estudiantes de carreras universitarias y de cursos de posgrado que quieran trabajar en el campo de la Optimización, de la Programación Matemática, los Métodos Cuantitativos de Gestión y la Investigación Operativa. Presenta herramientas para utilizar a modo de caja negra, como también el diseño de algoritmos específicos.

Los autores

¿Cómo leer este libro?

Referencias

Este libro es principalmente de aplicación práctica. Gran parte de su contenido consiste en código para transcribir directo a la consola de R. Por lo tanto, hemos adoptado una convención respecto a la tipografía para que las partes de este libro sean fácilmente identificables:

Cada vez que el Texto tenga este formato, entonces es texto explicatorio y formará parta del hilo conductor del libro.

```
Texto con este formato, es código R, ya sea para ingresar a
la consola o bien para grabar en un script. Los comentarios,
que se indican con #, se pueden suprimir. R no termina una
sentencia hasta que no cerremos sintácticamente a la misma,
así que podemos escribirlas en varias líneas como a veces,
por cuestión de espacio, aparece en este libro.
```

```
Texto con este formato se usa para mostrar la salida de consola. Es
lo que nos debería devolver R tras ingresar los comandos indicados.
No siempre se incluye la salida de consola, cuando se considera que
su inclusión es redundante la omitimos en este libro.
```

TEXTO CON ESTE FORMATO ES UN FORMULA O MODELO MATEMÁTICO

Como comentario extra, los bloques de código no son independientes unos de otros. Al inicio de cada capítulo tenemos siempre el siguiente código:

```
rm(list=ls())
setwd("D://Proyectos//IOR//scripts")
```

La primera línea borra la memoria, mientras que la segunda línea declarara cual es el directorio de trabajo. Adicionalmente, podemos encontrar este código de borrado repartido por el código de un mismo capítulo.

```
rm(list=ls())
```

Cuando encontramos esta línea, borrando la memoria de trabajo, significa que todo el código posterior a esa sentencia es independiente del anterior, así que podemos trabajar tranquilos. Pero, si buscando por el libro encontramos un código que nos interesa sin la llamada a

rm(list=ls()), debemos buscar la llamada inmediata anterior y tipear todo el código que existe a partir de ella (incluyendo el *rm(list=ls())*).

Estructura del libro

Este volumen tiene dos partes bien diferenciadas, ocupando cada una un 50 por ciento del mismo. La primera, dedicada a la introducción al entorno y al lenguaje R, puede ser omitida por quien ya tenga conocimientos del mismo. Si no es el caso, aconsejamos su lectura y la ejecución de todo el código proporcionado, siguiendo el orden prefijado por el libro. R tiene una sintaxis y una filosofía propia y diferente a la que nos tienen acostumbrados los lenguajes similares a C/C++ y Pascal. Pensado para el uso en problemas matemáticos en general, y estadísticos en particular, tiene una expresividad enorme que ayuda a no alejarnos de los problemas que estamos intentando resolver. Particularmente importante son los conceptos de vectorización y reciclado, a los que el usuario tendrá que acostumbrarse para sacar el máximo provecho de esta herramienta.

Por otro lado, la segunda parte está enfocada a la implementación y ejecución de distintas técnicas algorítmicas para resolución de problemas de programación matemática. Cada capítulo es independiente de los demás, pudiéndose leer en cualquier orden. Sin embargo, sin importar el algoritmo que se requiera utilizar, aconsejamos la lectura del apartado "Optimizando en base a muestras" del capítulo "Algoritmos para programación Lineal", ya que allí se muestra un puente entre el procesamiento estadístico de datos, el armado de modelo lineales y su posterior optimización (que, con un poco de esfuerzo, pueden ser migrados a otros algoritmos de optimización e incluso a problemas no lineales).

Índice de contenido

Introducción

1.1. ¿Qué es un problema de optimización?

Dentro de la investigación operativa, las técnicas de optimización se enfocan en determinar la política a seguir para maximizar o minimizar la respuesta del sistema. Dicha respuesta, en general, es un indicador del tipo “Costo”, “Producción”, “Ganancia”, etc., la cual es una función de la política seleccionada. Dicha respuesta se denomina objetivo, y la función asociada se llama función objetivo.

Pero, ¿qué entendemos cómo política? Una política es un determinado conjunto de valores que toman los factores que podemos controlar a fin de regular el rendimiento del sistema. Es decir, son las variables independientes de la función que la respuesta del sistema.
Por ejemplo, supongamos que deseamos definir un mix de cantidad de operarios y horas trabajadas para producir lo máximo posible en una planta de trabajos por lotes. En este ejemplo, nuestro objetivo es la producción de la planta, y las variables de decisión son la cantidad de operarios y la cantidad de horas trabajadas. Otros factores afectan a la producción, como la productividad de los operarios, pero los mismos no son controlables por el tomado de decisiones. Este último tipo de factores se denominan parámetros.

En general, el quid de la cuestión en los problemas de optimización radica en que estamos limitados en nuestro poder de decisión. Por ejemplo, podemos tener un tope máximo a la cantidad de horas trabajadas, al igual que un rango de cantidad de operarios entre los que nos podemos manejar. Estas limitaciones, llamadas “restricciones”, reducen la cantidad de alternativas posibles, definiendo un espacio acotado de soluciones factibles (y complicando, de paso, la resolución del problema). Observen que acabamos de hablar de “soluciones factibles”. Y es que, en optimización, cualquier vector con componentes apareadas a las variables independientes que satisfaga las restricciones, es una solución factible, es decir, una política que es posible de implementar en el sistema. Dentro de las soluciones factibles, pueden existir una o más soluciones óptimas, es decir, aquellas que, además de cumplir con todas las restricciones, maximizan (o minimizan, según sea el problema a resolver) el valor de la función objetivo.

Resumiendo, un problema de optimización está compuesto de los siguientes elementos:

- Un conjunto de restricciones
- Un conjunto de soluciones factibles, el cual contiene todas las posibles combinaciones de valores de variables independientes que satisfacen las restricciones anteriores.
- Una función objetivo, que vincula las soluciones factibles con la performance del sistema.

1.2. Modelado de sistemas Vs Resolución de modelos

Dentro de la práctica de la optimización, existen dos ramas relacionadas pero diferenciadas entre sí. Por un lado, nos encontramos con el problema de cómo modelar adecuadamente el sistema bajo estudio, y por otro como resolver el modelo. En general, la rama matemática de la IO se ocupa de buscar mejores métodos de solución, mientras que la rama ingenieril analiza formas de modelar sistemas reales.

Un ingeniero, en su actividad normal, se ocupa en general de modelar los sistemas, seleccionar un método de resolución y dejar que la computadora haga el resto. Sin embargo, es conveniente conocer el funcionamiento de cada metodología, a fin de determinar cuál es la más adecuada para cada situación, y modelar el problema de una forma amigable a dicho método.

1.3. Clasificación de problemas de optimización

En base a su naturaleza, hay varias formas de clasificar un problema de optimización. Analizar en qué categoría entra es importante para definir el método de solución a utilizar, ya que no hay un método único para todos los posibles problemas.

Para comenzar, una primera distinción la podemos realizar en base a la continuidad o no de las variables de decisión. Se dice que estamos frente a un problema de “Optimización Continua” cuando todas las variables de decisión pueden tomar cualquier valor perteneciente al conjunto de los reales. Dentro de este tipo de problemas, son de particular importancia los problemas de "Optimización Convexa", en los cuales se debe minimizar una función convexa sujeta a un conjunto solución convexo.

Cuando tanto la función objetivo como las restricciones son lineales, hablamos de un problema de "Optimización Convexa Lineal" o "Programación Lineal". En el caso de trabajar con variables discretas (es decir, que solo puedan tomar valores enteros) nos enfrentamos a un problema de “Optimización Combinatoria”. Por raro que pueda parecer, en general un problema de optimización combinatoria es más complicado de resolver que uno de optimización continua. En el medio tenemos los problemas de “Optimización Mixta” es los cuales algunas variables son continuas y otras son discretas. En la práctica, estos problemas se resuelven en forma más parecida a los problemas combinatorios que a los continuos. Un caso particular de optimización combinatoria es la “Optimización Binaria”, aquella en la cual todas sus variables están restringidas a tomar uno de dos valores (en general, 0 y 1). Este caso es bastante raro de encontrar en la práctica, siendo más habitual encontrar problemas combinatorios con algunas variables binarias (optimización mixta).

Otra clasificación la podemos hacer en base a la naturaleza probabilística del problema. Cuando podemos considerar que todas las variables son determinísticas, estamos ante un problema determinista, en caso contrario nos enfrentamos a un problema estocástico. El modelado y resolución de un problema estocástico es mucho más complejo que el modelado de un problema determinístico.

1.4. Clasificación de métodos de resolución

Los métodos de resolución de problemas de optimización se pueden clasificar en tres tipos diferentes:

- Resolución mediante cálculo
- Resolución mediante técnicas de búsquedas
- Resolución mediante técnicas de convergencia de soluciones

Resolución mediante cálculo

Los métodos de resolución por cálculo apelan al cálculo de derivadas para determinar para qué valores del dominio la función presenta un máximo o un mínimo. Son métodos de optimización muy robustos, pero que requieren mucho esfuerzo de cómputo y que tanto la función objetivo como las restricciones presenten determinadas condiciones (por

ejemplo, que estén definidas, que sean continuas, etc.). En general, no se suele apelar a estos métodos en el mundo de la ingeniería, ya que los problemas no se ajustan a las restricciones de continuidad y tienen demasiadas variables como para que su tratamiento pueda ser eficiente.

Resolución mediante técnicas de búsquedas

Dentro de este apartado, podemos encontrar un gran abanico de técnicas, desde el viejo método de prueba y error hasta las modernas técnicas de programación matemática. En forma genérica, consisten en el siguiente algoritmo:

Seleccionar una solución inicial y hacerla la solución actual:

1) Hacer Mejor Solución = Solución Actual
2) Buscar n soluciones cercanas a la Solución Actual
3) Para cada una de las n soluciones cercanas hacer
    a. Si el valor de la función objetivo de la solución a verificar es mayor (o menor) al valor generado por la solución actual, hacer Mejor Solución = Solución Evaluada
    b. Si Mejor Solución = Solución Actual, Finalizar del procedimiento, en caso contrario Hacer Solución Actual = Mejor Solución y volver a 2)

Dentro de estos métodos tenemos técnicas para abarcar una gran variedad de problemas. Desde técnicas exactas, como la "Programación Lineal" (que se limita solo a problemas con un conjunto solución convexo y función objetivo y restricciones lineales) hasta las técnicas metaheurísticas de solución aproximada como la "Búsqueda Tabú".

Resolución mediante técnicas de convergencia de soluciones

Dentro de este grupo, tenemos las técnicas más recientemente desarrolladas. A diferencia del conjunto anterior, está compuesto casi completamente por técnicas metaheurísticas (o sea, que los resultados van a ser aproximadamente óptimos). Estos métodos se basan en generar una gran cantidad de soluciones, determinar cuáles son las "mejores" y, a partir de ellas, generar un nuevo conjunto de soluciones a analizar, repitiendo el proceso hasta que las soluciones generadas

converjan en una (o sea, hasta la iteración en la cual todas las soluciones generadas tengan un valor de función objetivo muy parecido).
Entre las técnicas más conocidas de este grupo, aunque no las únicas, tenemos a todas las versiones de "Algoritmos Genéticos"

1.5. Optimización Convexa

La Optimización Convexa es una rama de las técnicas de Optimización que trata sobre técnicas de minimización de funciones convexas sobre un dominio también convexo.
Definamos primero que es una función convexa. Una función es convexa, o cóncava hacia arriba, si, para todo par de puntos pertenecientes a su dominio, la recta que los une pertenece o está sobre la curva. En otras palabras, una función es convexa si el conjunto de puntos situados sobre su gráfica es un conjunto convexo. Matemáticamente, una función cumple con estas características si, para todo X e Y perteneciente a su dominio, con X<Y, y un todo T tal que 0<T<1, se verifica:

$$f\left(tx+\left(1-t\right)y\right) \;\leq\; t\,f\left(x\right)+\left(1-t\right)f\left(y\right)$$

De más está decir que la función debe estar definida para todos los reales, al igual que su codominio.
Un conjunto de puntos de un espacio vectorial real se denomina conjunto convexo si, para cada par de puntos pertenecientes al conjunto, la recta que los une también pertenece al conjunto. Matemáticamente:

Dado el conjunto C, si para todo (A;B) perteneciente a C y todo T perteneciente a [0;1], se cumple que (1-T)A + TB pertenece a C.
Los problemas de optimización convexa toman la forma de una función objetivo convexa a minimizar y un conjunto de restricciones que cumplen las siguientes condiciones:

- Si son inecuaciones, son de la forma $g(x) \leq 0$, siendo g una función convexa.
- Si son ecuaciones, son de la forma $h(x) = 0$, siendo h una función afín.

En realidad, las ecuaciones son redundantes, ya que se pueden reemplazar por un par de inecuaciones de la forma $hi(x) \leq 0$ y $-hi(x) \leq 0$.

Noten que si la función h(x) no es afín, -hi(x) es cóncava, por lo cual el problema dejaría de ser convexo.
Cumpliendo las condiciones anteriores, nos aseguramos que el problema se puede resolver por los métodos de optimización convexa.

Es fácil darse cuenta que, en el caso de necesitar maximizar una función cóncava, es fácil transformar el problema en uno de minimización de función convexa, haciendo ft = -f. También, si las inecuaciones son de la forma h(x) ≥0 podemos transformarlas a ht(x) ≤0 mediante la transformación ht = -h.

Dentro de los métodos de resolución de estos tipos de problemas podemos nombrar los siguientes:

- Método Simplex (para problemas convexos lineales)
- Métodos de punto interior
- Resolución por elipsoides
- Resolución por subgradientes
- Resolución por proyección de subgradientes

En general, la mayoría de los problemas de optimización con variables reales suelen entrar dentro de esta categoría, por lo cual con el aprendizaje de este tipo de técnicas tenemos gran parte del camino allanado.

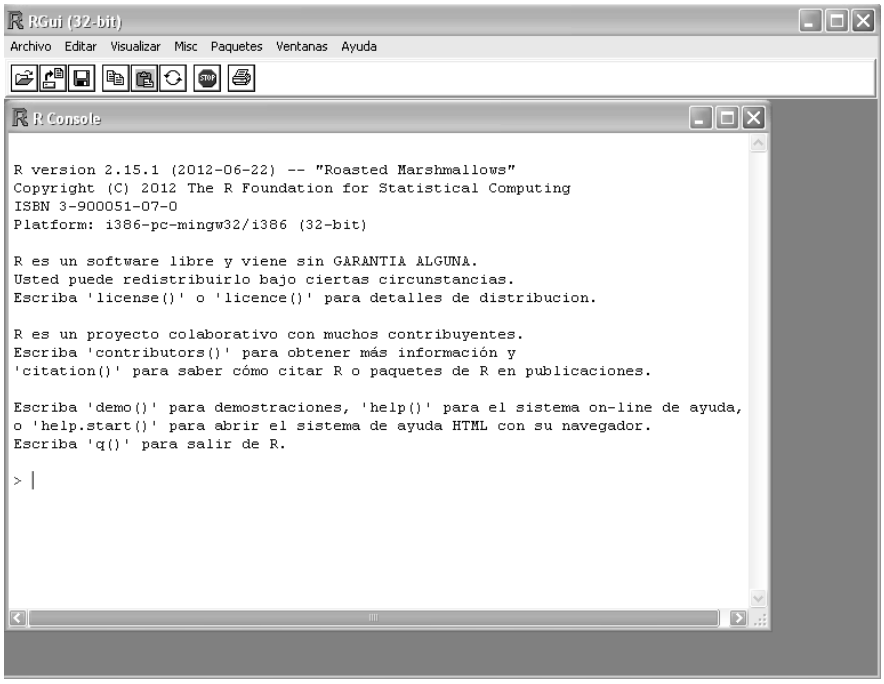
Dentro de los casos especiales, tenemos los siguientes tipos de problemas:

- Programación Lineal: Optimización convexa con función objetiva y restricciones lineales.
- Programación Cuadrática Convexa: Optimización convexa con restricciones lineales pero función objetivo cuadrática.
- Programación No Lineal Convexa: Optimización convexa con la función objetivo y/o las restricciones no lineales.

Introducción al uso de R

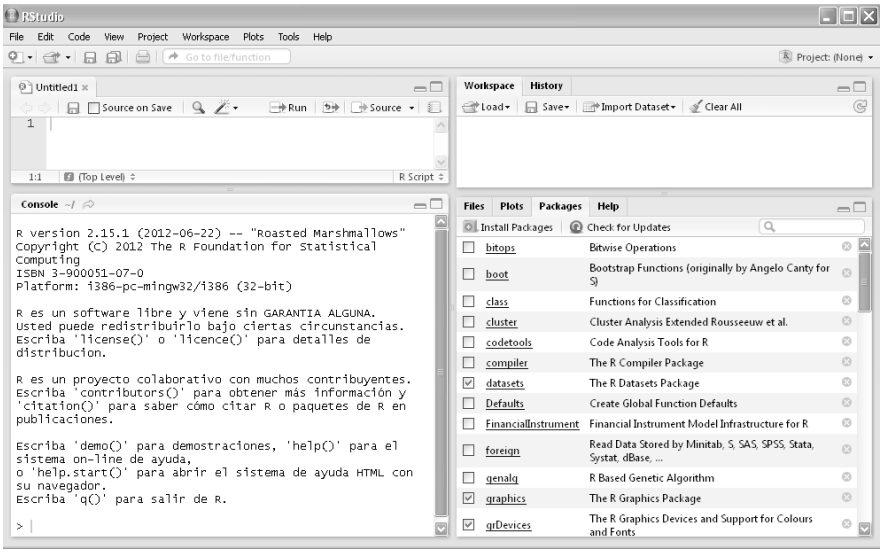
2.1 Actividades previas.

Antes de comenzar a trabajar, tenemos que instalarnos el software R. El mismo se puede descargar desde la página de CRAN (http://cran.r-project.org/). Una vez instalado, su aspecto es el siguiente:



La interfaz de R es una consola de comandos convencional, no viene con ningún IDE más avanzado. Pero podemos utilizar algunas de las muchas IDEs preparadas para R. En este libro vamos a trabajar con RStudio, pero no es de ninguna manera necesario tenerlo instalado. Todo R se puede manejar con comandos en la consola, así que podemos usar la consola básica, RStudio, o cualquier otra IDE sin perder el hilo del libro. De hecho, solo en algunas gráficas de este libro (principalmente algún que otro cuadro de diálogo en la ejecución del comando *View()*) se apreciará la

interfaz de RStudio. En el Anexo I hay un pequeño tutorial mostrando las características de esta IDE.



Amen a tener instalado el sistema, es necesario crear una estructura de directorios para trabajar con R. Creemos una estructura como la que sigue:

Directorio de trabajo

|____scripts

|____sources

|____data

Como dijimos antes, la interacción con R será mediante sentencias. Así que debemos ingresar el texto escrito en formato de código y pulsar enter para ejecutarlo. Si lo escribimos en forma incompleta (por ejemplo, nos olvidamos de cerrar un paréntesis), R se queda esperando a que terminemos la sentencia, tras lo cual debemos volver a pulsar enter. Si la construcción es sintácticamente incorrecta, la consola devuelve error.

2.2. Utilizando R

No hay mejor forma que aprender un nuevo lenguaje de programación y un nuevo software que usándolo. Así que vamos a empezar a trabajar con R. Suponemos que ya lo tenemos instalado, así que abrimos la consola de R o el IDE que estamos utilizando. Primero, limpiemos la memoria y definamos nuestro directorio de trabajo:

```
rm(list=ls())
setwd("D://Proyectos//IOR//scripts")
```

Donde debemos reemplazar "D://Proyectos//IOR//" por la ruta hacia el directorio para alojar nuestros proyectos en R. Referenciamos al subdirectorio script ya que en él es donde guardaremos el código que generemos en R.
Expliquemos qué hemos escrito. La segunda línea es fácil, "*setwd()*" es la abreviatura de "Set Working Directory" y lo que hace es definir a un directorio como directorio de trabajo (o sea, que cada vez que guardemos o leamos un archivos, a menos que escribamos una ruta, las operaciones se hacen en dicho directorio).
Si usamos *getwd()* ("Get Working Directory"):

```
getwd()
[1] "D://Proyectos//IOR//scripts"
```

Obtenemos la ruta del directorio de trabajo. Con *dir()* podemos ver su contenido:

```
dir()
```

Volvamos al comando "rm(list=ls())". El mismo se compone de una función anidada dentro de otra. *ls()* devuelve una lista con el contenido de la memoria de trabajo. *rm()* borra de la memoria los objetos que le indiquemos. Entonces *rm(list=ls())* borra todos los objetos que se encuentren en la memoria de trabajo.
Por ejemplo, creemos algunos objetos:

```
aux01 <- 1
aux02 <- 2
aux03 <- 3
```

Si quiero ver los objetos en memoria:

```
ls()
[1] "aux01" "aux02" "aux03"
```

Si quiero borrar "aux02":

```
rm(aux02)
```

Vuelvo a ejecutar *ls()* para ver si se borró:

```
ls()
[1] "aux01" "aux03"
```

Borremos nuevamente todos los objetos

```
rm(list=ls())
```

Y verifiquemos si están borrados:

```
ls()
character(0)
```

Los comentarios se declaran mediante una almohadilla "*#*"

```
# Esto es un comentario
```

Los objetos se crean mediante asignación de valores. El operador de asignación es "<-"[1]:

```
aux01 <- "Variable Auxiliar 01"
```

Y podemos consultar su valor simplemente llamándola:

```
aux01
[1] "Variable Auxiliar 01"
```

En general, si la variable contiene datos y no sabemos qué cantidad, debemos usar *View()* (ojo, R es sensible a mayúsculas y minúsculas):

```
View(aux01)
```

	x
1	Variable Auxiliar 01

Fíjense que pasa si hacemos *view(aux01)*:

[1] En realidad, también se puede emplear el operador "=", pero está desaconsejado. Las buenas prácticas dejan el "<-" como operador de asignación para variables, y limitan el "=" a la asignación de valores a parámetros de funciones. En este libro se seguirá dicho criterio.

```
view(aux01)
Error: no se pudo encontrar la función "view"
```

Nos tira error, porque para R "*view*" es diferente a "*View()*" (R es case sensitive[2]). Este comando nos ayuda bastante porque, de tener una variable con muchos datos, el mostrarlos invocándola directamente nos desplazaría la consola, mientras que con "*View()*" los datos se muestran en una nueva ventana.

R viene con su propia ayuda incluida, a la que se accede de múltiples maneras.
Si queremos ver la ayuda de la función *View*:

```
help(View)
```

O bien:

```
?View()
```

Si queremos ver el manual de ayuda:

```
help.start()
```

Podemos buscar alguna palabra con *help.search()*:

```
help.search("normal")
```

Buscar algún comando por parte de su nombre con *apropos()*:[3]

```
apropos("chi")
```

También, si los paquetes[4] lo permiten, podemos ver demostraciones de su uso mediante la función *demo()*. Por ejemplo, veamos la demostración del paquete "graphics":

```
demo(graphics)
```

[2] Diferencia mayúsculas de minúsculas cuando le damos instrucciones.

[3] OK, hubiera quedado mejor algo como "help.apropos", pero es lo que hay.

[4] Paquetes es el nombre que se le da en R a las bibliotecas empaquetadas. No son simplemente módulos que se cargan, sino que vienen con su propia ayuda, se instalan en el path de R, entre otros.

R es un lenguaje orientado a objetos. Cada elemento en memoria es un objeto, y si queremos acceder a alguno hay que asignarlo a alguna variable:

```
myVar01 <- 15
myVar01
[1] 15
```

Los objetos residen en memoria, las variables son meras referencias al objeto. Recordemos esto, la variable no es el objeto.
Cuando decimos que cada elemento en memoria es un objeto, nos referimos a todos: datos, funciones, gráficos, etc. Todo puede ser manipulado mediante variables. Cada objeto, indefectiblemente, tiene que pertenecer a una clase. R es un lenguaje de tipado dinámico y fuerte. O sea, puedo definir un objeto asignándole valores, que R se encarga de asignarle el tipo más apropiado (según su forma de ver las cosas), pero el mismo se mantiene inmutable el resto de la vida de dicho objeto.
De todas las clases disponibles, con las que más vamos a tratar son con las clases de datos. Los cuatro tipos básicos de datos son: "scalar", "factor", "character" y "logical".

"logical" es un tipo de dato simple, que permite guardar el valor de un bit. Es el clásico booleano de otros lenguajes de programación. Borremos la memoria de trabajo y pongámonos a trabajar con este tipo de datos:

```
rm(list=ls())

varBoolean01 <- TRUE
varBoolean02 <- FALSE
varBoolean03 <- TRUE

varBoolean01
[1] TRUE

varBoolean02
[1] FALSE

varBoolean03
[1] TRUE
```

Con *class()* chequeamos la clase a la que corresponde el objeto referenciado por la variable:

```
class(varBoolean01)
[1] "logical"
```

Obviamente, podemos realizar operaciones booleanas:

```
!varBoolean01                    # Negación
[1] FALSE
```

```
varBoolean01 && varBoolean02     # Operación AND
[1] FALSE

varBoolean01 || varBoolean02     # Operación OR
[1] TRUE
```

Podemos hacer operaciones más complejas también:

```
!((varBoolean01 || varBoolean02) && (varBoolean03))
```

Podemos utilizar números mediante la clase "scalar". En realidad nunca tratamos con esta clase directamente, sino que operamos con sus clases hijas, "numeric" y "complex":

```
varNumeric01 <- 135
varNumeric02 <- -20574
varNumeric03 <- 12/27
varNumeric04 <- pi * 2
varNumeric05 <- 3.4

varComplex01 <- 12 + 2i
varComplex02 <- 134/245 - 3.1i
varComplex03 <- 946/24i
varComplex04 <- 123.45 + (2 * pi) * 1i

class(varNumeric01)
[1] "numeric"

class(varComplex04)
[1] "complex"
```

Puedo realizar todas las operaciones que uno espera poder hacer con números:[5]

```
varNumeric06 <- varNumeric01 + varNumeric03
varNumeric06

varNumeric01 + varNumeric03
varNumeric01 - varNumeric03
varNumeric01 * varNumeric03
varNumeric01 / varNumeric03 # División real
varNumeric02 ^ 2

varNumericInteger01 <- 10
varNumericInteger02 <- 7

varNumericInteger01 %% varNumericInteger02   # Módulo
```

[5] Bueno, es bastante obvio, ¿no? Pero había que decirlo.

```
varNumericInteger01 %/% varNumericInteger02 # División entera
```

Observen un detalle. Si realizo una operación y la asigno a una variable, su resultado es un objeto referenciado por esa variable, pero la consola no lo muestra, debo llamar al objeto para ver el resultado. Pero si realizo el cálculo sin realizar ninguna asignación, se crea el objeto, se muestra su valor en la consola (el resultado del cálculo) pero es borrado inmediatamente de la memoria de trabajo.

Volviendo al tema de los cálculos, también es posible realizar comparaciones, las cuales nos devuelven un objeto del tipo "logical":[6]

```
varNumeric01 > varNumeric03
auxVar01 <- varNumeric01 > varNumeric03
class(auxVar01)

varNumeric01 > varNumeric03
varNumeric01 >= varNumeric03
varNumeric01 == varNumeric03
varNumeric01 < varNumeric03
varNumeric01 <= varNumeric03
```

Y como las comparaciones son valores "logical", podemos realizar operaciones binarias:

```
(varNumeric01 >= varNumeric03) || (varNumeric01 <= varNumeric02)
(varNumeric01 >= varNumeric03) && (varNumeric01 <= varNumeric02)
!(varNumeric01 > varNumeric03)
```

"character" es un tipo de datos que nos permite almacenar cadenas de texto:

```
myCharacter01 <- "Mi primer cadena de texto"
myCharacter02 <- "b"
myCharacter03 <- "1222"

class(myCharacter01)
[1] "character"
```

Tenemos algunas funciones especiales para trabajar con cadenas de texto:

[6] Debido a que el signo “=” se usa para asignar valores a parámetros de funciones, el operador para chequear igualdad es “==“.

```
paste(myCharacter03,myCharacter02) # Concatenación
[1] "1222 b"

substr(myCharacter01,1,10) # Recorte
[1] "Mi primer "

sub("cadena de texto", "String",myCharacter01) # Sustitución
[1] "Mi primer String"
```

Para cualquiera de estas clases, tenemos funciones *is.type()* y *as.type()*[7]:

```
is.numeric(myCharacter01)
is.numeric(varNumeric01)

varNumeric01 + myCharacter03  # Esto genera un error
varNumeric01 + as.numeric(myCharacter03)
```

is.numeric(), por ejemplo, devuelve TRUE si la variable es numérica y FALSE en caso contrario, mientras que *as.numeric()* nos devuelve el contenido de la variable pasada como parámetro en formato numérico (si la conversión es posible, obviamente).

El último tipo de datos es el "factor", utilizado para representar variables categóricas:[8]

```
varPosiblesValores <- c("Alto", "Medio", "Bajo")
varFactor01 <- factor("Alto", levels=varPosiblesValores)
varFactor01
[1] Alto
Levels: Alto Medio Bajo


varFactor02 <- factor(c("Alto", "Alto", "Bajo", "Medio"),
levels=varPosiblesValores)
varFactor02
[1] Alto  Alto  Bajo  Medio
Levels: Alto Medio Bajo
```

Los factores se construyen mediante la función *factor()* ya que es necesario, además del valor que tomé el factor (el primer parámetro), definirle los posibles valores que podía tomar (el parámetro “levels”).

Los factores tienen asociadas algunas funciones interesantes:

---

[7] Donde type se refiere al nombre de alguna de estas clases.

[8] En este ejemplo introducimos el comando “c”, abreviatura de concatenate. Es uno de los más usados en R, y permite unir varios datos en una única variable vectorial. A cada dato se puede acceder luego por subíndices.

```
levels(varFactor02)
[1] "Alto"  "Medio" "Bajo"

table(varFactor02)
varFactor02
 Alto Medio  Bajo
    2     1     1
```

Y aunque no tiene *as.factor* (en vez de eso se usa *factor*), si tiene *is.factor*:

```
is.factor(varFactor02)
```

Los tipos de datos anteriores (y cualquier objeto) puede tomar el valor NA. Este valor indica “no asignación” y se caracteriza por ser indefinido (es decir, no vale ni cero, ni falso, ni "", es indeterminado).
No es un tipo de dato, pero funciona parecido en algunas cosas:

```
aux01 <- NA
aux01
is.na(aux01)
```

Si intentamos averiguar su clase, devuelve "logical", sin importar que pueda contener la variable:

```
aux01<-"prueba"
class(aux01)
aux01<-NA
class(aux01)
```

Similar al NA, existe el valor NaN (Not a Number), que es el devuelto por algunas operaciones que no tengan sentido:

```
0/0
Inf/Inf
```

Como NA tiene su *is.na()*, NaN tiene su *is.nan()*:

```
is.nan(0/0)
is.nan(1/0)
```

Vimos antes que usamos el valor “Inf”. “Inf” y “-Inf” son otros dos valores especiales que identifican al infinito positivo y negativo respectivamente:

```
1/0
[1] Inf

-1/0
[1] -Inf
```

Existen las funciones *is.finite()* e *is.infinite()* asociadas a estos valores:

```
is.finite(1/0)
is.finite(1/Inf)
is.infinite(2^Inf)
```

Ahora bien, estos tipos de datos, así como están, la verdad que sirven muy poco. En el uso diario, los empleamos como ladrillos para estructuras más complejas. De los datos complejos, la base de todo R es el vector. Un vector se crea con la función *c()* (abreviatura de concatenate)

```
myNumericVector <- c(1,345.6, pi,1/3)
myNumericVector
[1]   1.0000000 345.6000000   3.1415927   0.3333333
```

Un vector es un conjunto de objetos ordenados del mismo tipo:

```
class(myNumericVector)
[1] "numeric"
```

Un vector puede tener cualquier cantidad de elementos, incluso solo uno, pero todos deben ser del mismo tipo básico. Se accede a los mismos mediante sub-índices:

```
myNumericVector[3]
[1] 3.141593
```

Cuidado, R comienza a indexar a partir de 1, no de 0[9]. Puedo ampliar un vector a discreción con la función *c()*:

```
myNumericVector <- c(myNumericVector, 123, 2*pi/3,NA , 10,-
23.9, NA ,-14/13)
myNumericVector
[1]    1.0000000  345.6000000    3.1415927    0.3333333  123.0000000
2.0943951
 [7]           NA  10.0000000 -23.9000000           NA  -1.0769231
```

Y acceder a valores por rango de subíndices:

```
myNumericVector[2:5]
[1] 345.6000000   3.1415927   0.3333333 123.0000000

myNumericVector[-1]
```

[9] Pese a que esto nos encuentra desacostumbrados, es una muy buena idea que los índices comiencen por "1". Por ejemplo, si calculamos la cantidad de elementos de un vector, el índice del último elemento del vector se corresponde con este número.

```
[1]  345.6000000     3.1415927     0.3333333  123.0000000     2.0943951
NA
 [7]  10.0000000 -23.9000000           NA  -1.0769231
```

```
myNumericVector[-1:-2]
[1]    3.1415927    0.3333333  123.0000000    2.0943951           NA
10.0000000
[7] -23.9000000           NA  -1.0769231
```

Tenemos algunas funciones que se aplican a vectores. Por ejemplo, *length()* nos devuelve el tamaño del vector, y *order()* los índices del vector ordenados crecientemente según los valores de cada componente:

```
length(myNumericVector)
[1] 11
```

```
order(myNumericVector)
[1]  9 11  4  1  6  3  8  5  2  7 10
```

Usando la función *order()*, podemos devolver los elementos de un vector ordenados:

```
myNumericVector[order(myNumericVector)]
[1] -23.9000000   -1.0769231    0.3333333    1.0000000    2.0943951
3.1415927
 [7]  10.0000000 123.0000000 345.6000000          NA          NA
```

Los vectores en R tienen una peculiaridad muy especial y es que muchas de sus operaciones están vectorizadas. La vectorización es un mecanismo que permite aplicar una misma función o expresión a todos los elementos de un vector, sin necesidad de construir un procedimiento para iterar. Por ejemplo, si quiero obtener todos los valores no negativos del vector

```
myNumericVector>=0
[1]   TRUE   TRUE   TRUE   TRUE   TRUE   TRUE     NA   TRUE  FALSE     NA
FALSE
```

```
which(myNumericVector>=0)
[1] 1 2 3 4 5 6 8
```

```
myNumericVector[which(myNumericVector>=0)]
[1]    1.0000000  345.6000000    3.1415927    0.3333333  123.0000000
2.0943951
[7]  10.0000000
```

```
myNumericVector[myNumericVector>=0]
[1]    1.0000000  345.6000000    3.1415927    0.3333333  123.0000000
2.0943951
[7]          NA  10.0000000          NA
```

Observen que si filtro sin usar el *which()*, me devuelve también los valores que no puede evaluar (los NA en este caso).
Si quiero devolver todos los valores que no sean NA:

```
myNumericVector[which(!is.na(myNumericVector))]
[1]     1.0000000  345.6000000     3.1415927     0.3333333  123.0000000
2.0943951
[7]  10.0000000 -23.9000000  -1.0769231
```

Lo más interesante, es que podemos realizar asignaciones a los elementos del vector que cumplen con la condición dada en el filtro:

```
myNumericVector02 <- myNumericVector
myNumericVector02[which(myNumericVector02>10)] <- 0
myNumericVector02
[1]    1.0000000    0.0000000    3.1415927    0.3333333    0.0000000
2.0943951
 [7]          NA  10.0000000 -23.9000000          NA  -1.0769231


myNumericVector03 <-
(myNumericVector02+5)[(myNumericVector02 +5) >2]
myNumericVector03
[1]   6.000000   5.000000   8.141593   5.333333   5.000000   7.094395
NA
 [8] 15.000000         NA  3.923077
```

Asociado al concepto de vectorización está el de reciclado (recycling en inglés), gracias al cual podemos hacer cosas como:

```
myNumericVector/10
[1]   0.10000000  34.56000000   0.31415927   0.03333333  12.30000000
0.20943951
 [7]          NA  1.00000000 -2.39000000          NA -0.10769231

myNumericVector - 20
[1]  -19.00000  325.60000  -16.85841  -19.66667  103.00000  -17.90560
NA
 [8] -10.00000 -43.90000         NA -21.07692
```

Esto funciona porque R transforma a 10 y 20 en vectores de igual longitud que myNumericVector:

```
myNumericVector/c(10,10,10,10,10,10,10,10,10,10,10)
myNumericVector04 <-
  myNumericVector/c(10,10,10,10,10,10,10,10,10,10,10)
```

Obviamente, esto funciona con vectores numéricos, no tiene sentido en vectores de caracteres.
Puedo generar secuencias y grabarlas en mi vector:

```
myNumericVector04 <- seq (1:10)
```

```
myNumericVector04
[1]  1  2  3  4  5  6  7  8  9 10

myNumericVector04 <- seq (from= 1, to= 10, by=2)
myNumericVector04
[1] 1 3 5 7 9

myNumericVector04 <- rep(1:4, 2)
myNumericVector04
[1] 1 2 3 4 1 2 3 4

myNumericVector04 <- rep(1:4, each=2)
myNumericVector04
[1] 1 1 2 2 3 3 4 4

myNumericVector04 <- rep(1:4, each=2, len=5)
myNumericVector04
[1] 1 1 2 2 3

myNumericVector04 <- rep(1:4, each=2, times=3)
myNumericVector04
[1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4


myNumericVector04 <- rep(myNumericVector04, times=2)
myNumericVector04
[1] 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
[33]1 1 2 2 3 3 4 4 1 1 2 2 3 3 4 4
```

Ahora bien, los vectores son un tipo de dato interesante, pero son unidimensionales. ¿Qué pasa si quiero manejar datos de altura discriminados por sexo?:[10]

```
altura <- rnorm(10, 1.70, 0.1)
altura
[1] 1.651319 1.779054 1.692859 1.637679 1.669193 1.699384 1.810102
[8]1.822464 1.911979 1.685002

sexo <- sample(factor(c("M","F")), size=10, replace=TRUE)
sexo
[1] M M F F F M M F F
Levels: F M
```

Si me aseguro que los subíndices de cada vector sean equivalentes, para saber el sexo y la altura del tercer elemento muestreal:

```
sexo[3]
[1] F
Levels: F M
```

[10] Para generar los valores utilicé la función *rnorm*, que genera valores aleatorios con distribución normal, y *sample*, que sirve para realizar muestreos de variables discretas.

```
altura[3]
[1] 1.692859
```

Puede ser algo incómodo, ¿no?[11] Probemos usar un "data.frame":

```
datosPoblacion <- data.frame(sexo, altura)
datosPoblacion
   sexo   altura
1     M 1.651319
2     M 1.779054
3     F 1.692859
4     F 1.637679
5     F 1.669193
6     F 1.699384
7     M 1.810102
8     M 1.822464
9     F 1.911979
10    F 1.685002
```

Mucho mejor, ¿no? Accedamos al tercer elemento:

```
datosPoblacion[3,]
  sexo   altura
3    F 1.692859
```

O mostremos solo la altura del tercer elemento:

```
datosPoblacion[3,2]
[1] 1.692859
```

También puedo mostrar todas las alturas:

```
datosPoblacion[,2]
[1] 1.651319 1.779054 1.692859 1.637679 1.669193 1.699384 1.810102
[8]1.822464 1.911979 1.685002
```

Si quiero, puedo ver las alturas “con título”:

```
datosPoblacion[2]
     altura
1  1.651319
2  1.779054
3  1.692859
4  1.637679
5  1.669193
6  1.699384
7  1.810102
8  1.822464
9  1.911979
10 1.685002
```

Cuando el data.frame es grande, conviene usar *View()*:

[11] Y sobre todo propenso a errores.

```
View(datosPoblacion)
```

	sexo	altura
1	M	1.651319
2	M	1.779054
3	F	1.692859
4	F	1.637679
5	F	1.669193
6	F	1.699384
7	M	1.810102
8	M	1.822464
9	F	1.911979
10	F	1.685002

También puedo acceder a las columnas por nombre utilizando el signo "$":

```
datosPoblacion$altura
[1] 1.651319 1.779054 1.692859 1.637679 1.669193 1.699384 1.810102
[8]1.822464 1.911979 1.685002

datosPoblacion$altura[3]
[1] 1.692859
```

Cada columna, y cada fila, son vectores hechos y derechos, así que puedo aplicarles filtros:

```
datosPoblacion[datosPoblacion$sexo == "M",]
  sexo   altura
1    M 1.651319
2    M 1.779054
7    M 1.810102
8    M 1.822464
```

Observen la coma al final del filtro. Le indica a R que no queremos filtrar por columnas (en realidad, les estamos pasando un filtro nulo). Puedo ampliar un data.frame cuando quiera, por ejemplo, ahora vamos a agregarle un campo "peso":

```
datosPoblacion$peso <- rnorm(10, 80, 15)
View(datosPoblacion)
```

	sexo	altura	peso
1	F	1.676572	65.17624
2	M	1.897857	84.37898
3	F	1.771568	93.21675
4	M	1.728978	76.59352
5	F	1.675330	77.48651
6	F	1.888984	95.62467
7	M	1.766966	90.38411
8	F	1.710781	79.98631
9	F	1.796912	70.57039
10	M	1.681548	45.71884

Ni siquiera hubo que declararlo, solo poner datos. También funciona:[12]

```
datosPoblacion <- cbind(datosPoblacion, rnorm(10, 50, 15))
View(datosPoblacion)
```

	sexo	altura	peso	rnorm(10, 50, 15)
1	F	1.676572	65.17624	43.19132
2	M	1.897857	84.37898	38.27734
3	F	1.771568	93.21675	67.61725
4	M	1.728978	76.59352	38.63001
5	F	1.675330	77.48651	21.53811
6	F	1.888984	95.62467	69.04719
7	M	1.766966	90.38411	53.62271
8	F	1.710781	79.98631	19.74455
9	F	1.796912	70.57039	38.69197
10	M	1.681548	45.71884	46.87654

Qué problema, no tenemos nombre para la nueva columna:

```
names(datosPoblacion)
[1] "sexo"              "altura"            "peso"
[4] "rnorm(10, 50, 15)"

names(datosPoblacion) <-c(names(datosPoblacion[1:3]),"edad")
names(datosPoblacion)
[1] "sexo"   "altura" "peso"   "edad"
```

[12] *cbind* es equivalente a la función *c* pero para data frames. *cbind* concatena columnas a nuestro data frame. También existe *rbind*, que concatenas filas a nuestro data frame.

Claro, pero para mirar el conjunto de datos no necesito R. Los podría mirar desde cualquier otro software. ¿Por qué no calculo estadísticos descriptivos?[13]:

```
summary(datosPoblacion)
sexo      altura          peso             edad
 F:6   Min.   :1.675   Min.   :45.72   Min.   :19.74
 M:4   1st Qu.:1.689   1st Qu.:72.08   1st Qu.:38.37
       Median :1.748   Median :78.74   Median :40.94
       Mean   :1.760   Mean   :77.91   Mean   :43.72
       3rd Qu.:1.791   3rd Qu.:88.88   3rd Qu.:51.94
       Max.   :1.898   Max.   :95.62   Max.   :69.05
```

Muy lindo, pero difícil para trabajar. Si quiero solo la media del peso:

```
mean(datosPoblacion$peso)
[1] 77.91363
```

¿Y si quiero un vector con la media de todos los campos?:[14]

```
sapply(datosPoblacion, mean)
    sexo   altura     peso     edad
      NA  1.75955 77.91363 43.72370
Mensajes de aviso perdidos
In mean.default(X[[1L]], ...) :
  argument is not numeric or logical: returning NA
```

¿Y el su desvío estándar?:[15]

```
sapply(datosPoblacion, sd)
       sexo      altura        peso        edad
         NA  0.08222817 14.92228094 16.56140722
Mensajes de aviso perdidos
In var(as.vector(x), na.rm = na.rm) : NAs introducidos por coerción
```

Quizás me interesaría saber cuánto suman las columnas:

```
colSums(datosPoblacion[,2:4])
  altura     peso     edad
 17.5955 779.1363 437.2370
```

[13] Para profundizar en el uso de estadísticos descriptivos en R, y acerca de R en general, un buen libro de consultas es “Introducción a R”.

[14] Fíjense que, por más que hay un campo que no es numérico, por lo cual no se le puede calcular la media, R calcula la media para todos los otros campos e informa que campo no se pudo calcular.

[15] Resumiendo, *sapply()* nos permite aplicar funciones a cada campo (columna) de un data frame.

O calcular la frecuencia absoluta para alguna variable:

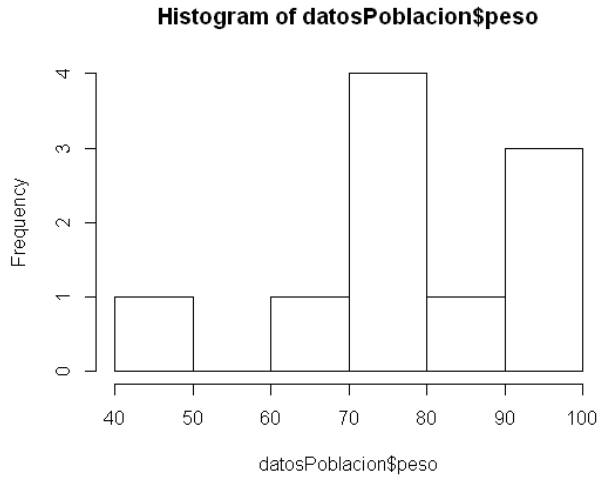
```
table(datosPoblacion$sexo)
F M
6 4
```

Ahora bien, saber la media de la altura es interesante, ¿pero será la misma por cada sexo?:[16]

```
tapply(datosPoblacion$altura, datosPoblacion$sexo, mean)
       F        M
1.753358 1.768837
```

Ahora, muy lindo todo, ¿por qué no hacemos algunos gráficos?:

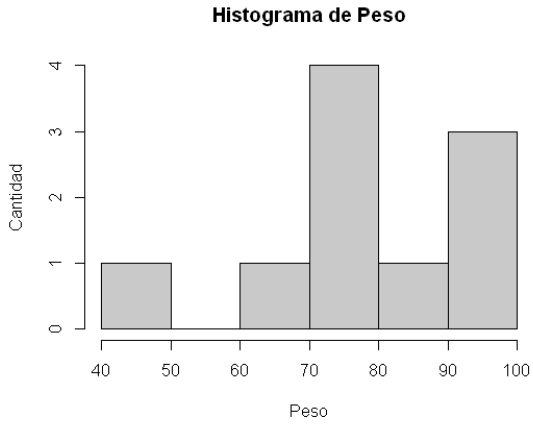
```
hist(datosPoblacion$peso)
```



Muy feo, ¿no?:

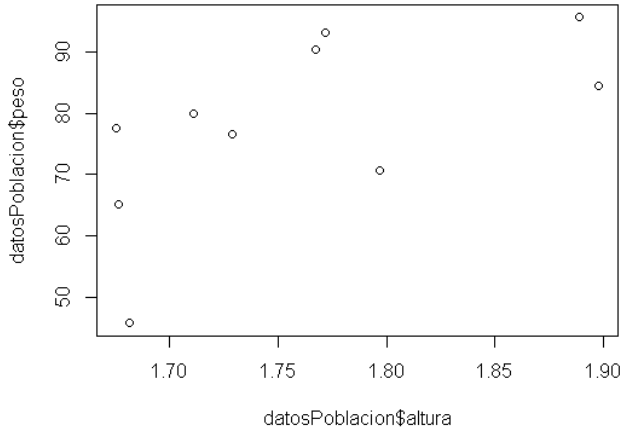
```
hist(datosPoblacion$peso, col = "cyan", main="Histograma de
Peso", xlab = "Peso", ylab = "Cantidad", freq = TRUE)
```

[16] La función *tapply()* nos permite aplicar una función (el tercer parámetro) a una columna de un data frame (el primer parámetro) discriminando los datos según el valor de otra columna del data frame (el segundo parámetro). En este caso, calculamos la media de la altura discriminando por sexo.



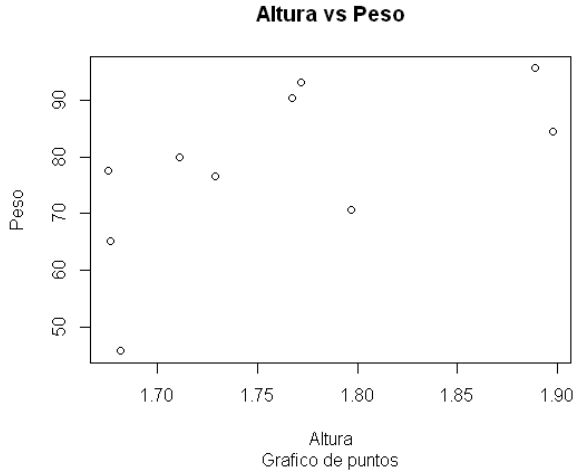
¿Estarán relacionadas la altura y el peso?

```
plot(datosPoblacion$altura, datosPoblacion$peso)
```



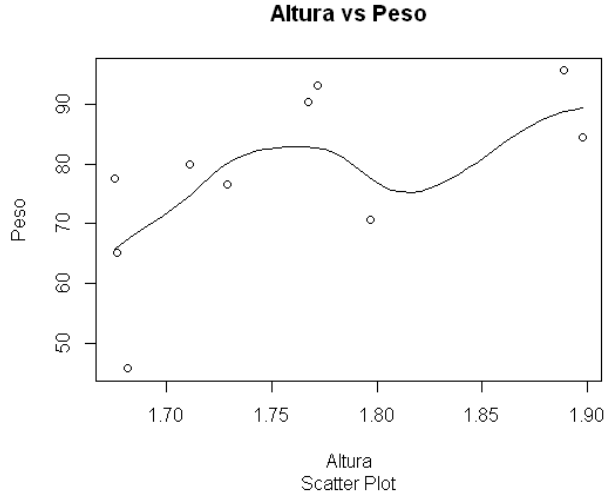
Bueno, podemos mejorarlo:

```
plot(datosPoblacion$altura, datosPoblacion$peso, xlab =
"Altura", ylab = "Peso", main = "Altura vs Peso", type =
"p", sub = "Grafico de puntos")
```



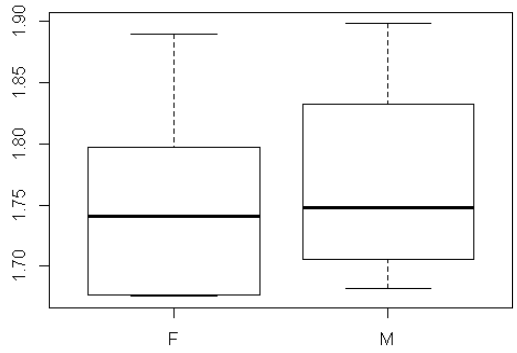
También podemos hacer:

```
scatter.smooth(datosPoblacion$altura,    datosPoblacion$peso,
xlab = "Altura", ylab = "Peso", main = "Altura vs Peso",
type = "p", sub = "Scatter Plot")
```



¿Y si graficamos la altura por sexo?:

```
boxplot(datosPoblacion$altura ~ datosPoblacion$sexo)
```



Deberíamos poder guardar nuestro data frame para usarlo en el futuro:

```
write.csv(datosPoblacion, "..//data//datosPoblacion.csv")
```

Fíjense que R permite usar referencias relativas a directorios, por lo cual no importa donde tenemos ubicada a nuestra carpeta, sino que mantengamos la estructura adecuada.

Probemos ahorra borrar nuestro data frame:

```
rm(datosPoblacion)
datosPoblacion
Error: objeto 'datosPoblacion' no encontrado
```

Como vemos, el objeto "datosPoblacion" no está más en nuestra memoria. Vamos a leerlo y cargarlo en memoria:

```
datosPoblacion <- read.csv("..//data//datosPoblacion.csv")
View(datosPoblacion)
```

2.3. Acerca del trabajo con matrices en R

Borremos la memoria de trabajo y carguemos el paquete para trabajar con matrices:

```
rm(list=ls())
```

```
library(Matrix)
```

Creemos ahora nuestra primera matriz:[17]

```
myMatriz <- matrix(c(1,2,3,4,5,6), nrow = 2, ncol = 3, byrow
= TRUE)

myMatriz
     [,1] [,2] [,3]
[1,]    1    2    3
[2,]    4    5    6
```

Podemos nombrar a las filas y columnas:

```
nombreFilas <- c("Fila01", "Fila02")

nombreColumnas <- c("Col01", "Col02", "Col03")

dimnames(myMatriz) <- list(nombreFilas, nombreColumnas)

myMatriz
       Col01 Col02 Col03
Fila01     1     2     3
Fila02     4     5     6
```

Accedo a sus valores usando la notación de "[ ]":

```
myMatriz[2,3]       # Fila 2, columna 3
[1] 6

myMatriz[2,]        # Segunda fila
Col01 Col02 Col03
    4     5     6

myMatriz[,3]        # Tercera columna
Fila01 Fila02
     3      6
```

Si pongo un solo índice:

```
myMatriz[3]
[1] 2
```

Me devuelve la celda "n", contando primero toda la primera columna, después la segunda, etc. [18]O sea[19]:

---

[17] El primer parámetro de *matrix()*, el vector “c(1,2,3,4,5,6)” contiene a los elementos de la matriz. Los mismos se ordenan de acuerdo al valor del parámetro “byrow”. Si vale TRUE, los elementos se cargan completando filas y pasando luego a las columnas. Prueben generar la misma matriz pero con el parámetro “byrow” tomando el valor FALSE, verán que la misma se llena por columnas.

[18] Si, sin importar con qué valor de “byrow” la creé.

```
n <- 4

myMatriz[n] == myMatriz[nrow(myMatriz)- (n %%
nrow(myMatriz)) , (n-1) %/% nrow(myMatriz)+1]


n <- 5

myMatriz[n] == myMatriz[nrow(myMatriz)- (n %%
nrow(myMatriz)) , (n-1) %/% nrow(myMatriz)+1]


n <- 2

myMatriz[n] == myMatriz[nrow(myMatriz)- (n %%
nrow(myMatriz)) , (n-1) %/% nrow(myMatriz)+1]


n <- 6

myMatriz[n] == myMatriz[nrow(myMatriz)- (n %%
nrow(myMatriz)) , (n-1) %/% nrow(myMatriz)+1]
```

Como tipo de dato, tiene sus métodos is.matrix y as.matrix:

```
is.matrix(myMatriz)

myVector <-c(1,2,3)

is.vector(myVector)

is.matrix(myVector)

is.matrix(as.matrix(myVector))
```

matrix es un tipo básico disponible sin necesidad de cargar el paquete "Matrix", pero este permite disponer de toda la funcionalidad para el trabajo con matrices:

```
myOtraMatriz <- matrix(10 + seq(1:28), 4, 7, byrow = TRUE)

myOtraMatriz
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]   11   12   13   14   15   16   17
[2,]   18   19   20   21   22   23   24
[3,]   25   26   27   28   29   30   31
[4,]   32   33   34   35   36   37   38
```

[19] El operador "==" sirve, como ya vimos, para realizar comparaciones, devolviendo un valor del tipo "logical".

```
myOtraMatrizXCol <- matrix(10 + seq(1:28), 4, 7, byrow =
FALSE)
```

```
myOtraMatrizXCol
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]   11   15   19   23   27   31   35
[2,]   12   16   20   24   28   32   36
[3,]   13   17   21   25   29   33   37
[4,]   14   18   22   26   30   34   38
```

```
myOtraMatriz + myOtraMatrizXCol
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]   22   27   32   37   42   47   52
[2,]   30   35   40   45   50   55   60
[3,]   38   43   48   53   58   63   68
[4,]   46   51   56   61   66   71   76
```

```
myTercerMatriz <- matrix(seq(1:14),7,2,byrow = TRUE)
```

```
myTercerMatriz
     [,1] [,2]
[1,]    1    2
[2,]    3    4
[3,]    5    6
[4,]    7    8
[5,]    9   10
[6,]   11   12
[7,]   13   14
```

```
myOtraMatriz %*% myTercerMatriz # Producto matricial
     [,1] [,2]
[1,]  742  840
[2,] 1085 1232
[3,] 1428 1624
[4,] 1771 2016
```

Puedo ampliar matrices:[20]

```
myOtraMatrizAmpliada <- cbind(c(-1,-2,-3,-4), myOtraMatriz)
```

```
myOtraMatrizAmpliada
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8]
[1,]   -1   11   12   13   14   15   16   17
[2,]   -2   18   19   20   21   22   23   24
[3,]   -3   25   26   27   28   29   30   31
[4,]   -4   32   33   34   35   36   37   38
```

Puedo aplicar vectorización en una matriz:

```
myOtraMatriz
```

```
myOtraMatriz + 10
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]   21   22   23   24   25   26   27
```

[20] Nuevamente podemos ver el uso de *cbind()*

```
[2,]   28   29   30   31   32   33   34
[3,]   35   36   37   38   39   40   41
[4,]   42   43   44   45   46   47   48
```

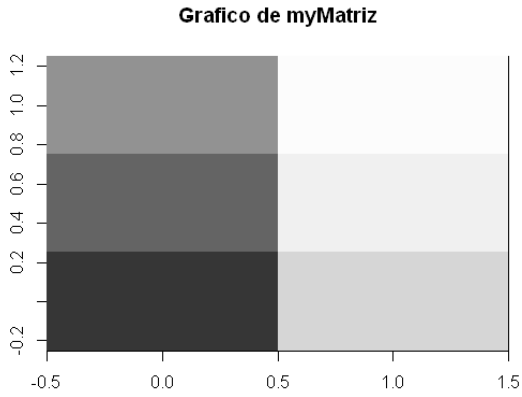
```
myOtraMatriz * 10
     [,1] [,2] [,3] [,4] [,5] [,6] [,7]
[1,]  110  120  130  140  150  160  170
[2,]  180  190  200  210  220  230  240
[3,]  250  260  270  280  290  300  310
[4,]  320  330  340  350  360  370  380
```

¿Qué pasa si hago un summary?:[21]

```
summary(myMatriz)
     Col01          Col02          Col03
 Min.   :1.00   Min.   :2.00   Min.   :3.00
 1st Qu.:1.75   1st Qu.:2.75   1st Qu.:3.75
 Median :2.50   Median :3.50   Median :4.50
 Mean   :2.50   Mean   :3.50   Mean   :4.50
 3rd Qu.:3.25   3rd Qu.:4.25   3rd Qu.:5.25
 Max.   :4.00   Max.   :5.00   Max.   :6.00
```

Podemos hacer cosas otras interesantes también:

```
print(image(myMatriz, main = "Grafico de myMatriz"))
```



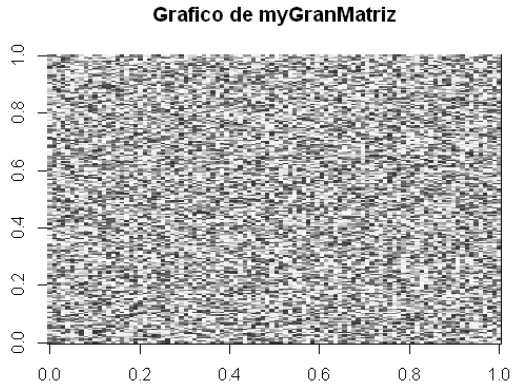
Claro, no sirve para nada lo que hicimos, pero pensemos en esto[22]:

[21] Observen como *summary()* (y muchas otras funciones de R) se adapta a la clase del objeto sobre la cual la aplico.

[22] *print* es una función para imprimir en un dispositivo de salida (la pantalla por defecto) lo que le pasemos por parámetro. *image* genera una imagen representativa de un conjunto de datos. *runif* genera números aleatorios de distribución uniforme (es similar a *rnorm*, salvo que ésta aplica a distribuciones normales). En este ejemplo "graficamos" la distribución de valores numéricos en

```
myGranMatriz <- matrix(rep(seq(1:1000), times = 20),100,200,
byrow = TRUE)
print(image(myGranMatriz, main = "Grafico de myGranMatriz"))
```


```
myGranMatriz <- matrix(runif(100*200,0,50) ,100,200, byrow =
TRUE)
print(image(myGranMatriz, main = "Grafico de myGranMatriz"))
```



Podemos calcular la inversa de una matriz:

```
myMatrizCuadrada <- matrix(seq(1:4),2,2,byrow=TRUE)
```

---

la matriz. A priori, lo podríamos utilizar para diferenciar visualmente ruido aleatorio de alguna otra estructura subyacente.

```
myMatrizCuadrada
     [,1] [,2]
[1,]    1    2
[2,]    3    4
```

```
solve(myMatrizCuadrada)
     [,1] [,2]
[1,] -2.0  1.0
[2,]  1.5 -0.5
```

La función *solve()* tiene otro uso interesante. Se puede utilizar para resolver sistemas de ecuaciones[23]:

```
matrizCoeficientes <- matrix(c(1,2,3,-1,-4,-10, 0,3,1), 3,
3, byrow=TRUE)

vectorTermIndep <- c(0, -1, -25)

vectorSolucion <- (solve(matrizCoeficientes,
vectorTermIndep))

vectorSolucion
[1] 10.157895 -9.263158  2.789474
```

```
matrizCoeficientes %*% vectorSolucion == vectorTermIndep
      [,1]
[1,] FALSE
[2,] FALSE
[3,]  TRUE
```

¿Qué pasó?, ¿el producto no debería haber sido igual al vector de los términos independientes?:

```
matrizCoeficientes %*% vectorSolucion - vectorTermIndep
              [,1]
[1,] -4.440892e-16
[2,]  2.664535e-15
[3,]  0.000000e+00
```

Lo que sucede es que, como cualquier otro método numérico, los resultados son aproximaciones. Debemos tener esto en cuenta al utilizar automáticamente los resultados de cualquier cálculo.

Con las matrices, podemos usar la función *apply()* para calcular funciones por columna o fila:

[23] En su forma de uso más general, *solve* resuelve ecuaciones de la forma a%*%x==b, con x y b pudiendo ser vectores o matrices.

```
apply(z, 2, mean) # Promedio por columnas
[1] 0.5 1.0

apply(z, 1, mean) # Promedio por filas
[1] 1.0 0.5
```

Tenemos a nuestra disposición también los productos externos. El resultado de este es una matriz cuyo vector de dimensión es igual a la concatenación de los vectores de dimensiones de los dos vectores a multiplicar, y su contenido consiste en todas las multiplicaciones posibles entre los elementos de los dos elementos:

```
z %o% y

, , 1, 1

     [,1] [,2]
[1,]    1    1
[2,]    0    1

, , 2, 1

     [,1] [,2]
[1,]    1    1
[2,]    0    1

, , 1, 2

     [,1] [,2]
[1,]    0    0
[2,]    0    0

, , 2, 2

     [,1] [,2]
[1,]    1    1
[2,]    0    1

, , 1, 3

     [,1] [,2]
[1,]    1    1
[2,]    0    1

, , 2, 3

     [,1] [,2]
[1,]    1    1
[2,]    0    1
```

Para hacer lo mismo podemos utilizar la función *outer()*:

```
outer(z,y,"*")
```

E incluso podemos reemplazar la operación "*" por cualquier operación:

```
outer(z,y,"+")
outer(z,y,"-")
```

2.4. Arrays

Los arrays son generalizaciones de los vectores y las matrices. Un vector se puede utilizar como un array multidimensional mediante la función *dim()*. Un array se compone de un vector de datos y un vector de dimensión, este último es el que se declara con *dim()*:

```
z<-numeric(3*5*10) # Creo un vector de 150 elementos,
                   # todos con valor cero
dim(z)<-c(3,5,10)
z
```

```
, , 1

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

, , 2

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

, , 3

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

, , 4

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

, , 5

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

, , 6

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

, , 7

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

, , 8

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
```

```
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

, , 9

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0

, , 10

     [,1] [,2] [,3] [,4] [,5]
[1,]    0    0    0    0    0
[2,]    0    0    0    0    0
[3,]    0    0    0    0    0
```

Con esto digo que "z" es un vector con tres dimensiones, con 3 componentes en la primera, 5 en la segunda y 10 en la tercera. Observen que fue necesario declarar la longitud, o la cantidad de componentes, del vector (150 = 3*5*10) con el comando dim(), y luego pasarle el vector de cómo se deben distribuir las componentes entre las dimensiones (c(3,5,10)):

```
z[1,4,7]<-25
z[1,4,7]
[1] 25

z[1,4,]
[1]  0  0  0  0  0  0 25  0  0  0

z[1,,]
     [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9] [,10]
[1,]    0    0    0    0    0    0    0    0    0     0
[2,]    0    0    0    0    0    0    0    0    0     0
[3,]    0    0    0    0    0    0    0    0    0     0
[4,]    0    0    0    0    0    0   25    0    0     0
[5,]    0    0    0    0    0    0    0    0    0     0
```

También podría haberlo declarado con la función array(), donde el primer parámetro es el vector de datos y el segundo el de dimensión:

```
x<-c(1:10)
y<-c(5,2)
z<-array(x,y)
z
     [,1] [,2]
[1,]    1    6
[2,]    2    7
[3,]    3    8
[4,]    4    9
[5,]    5   10
```

Los arrays nos permiten trabajar con subarrays:

```
x<-array(c(1,5,4,2,2,1), c(3,2))
```

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

z[x] # Pido que devuelva los elementos (1,2), (5,2) y (4,1)
[1]  6 10  4

z[x]<-0
z
     [,1] [,2]
[1,]    1    0
[2,]    2    7
[3,]    3    8
[4,]    0    9
[5,]    5    0
```

2.5. Listas

Las "list" son colecciones ordenadas de otros objetos (incluso de otras listas). Estos objetos pueden ser de cualquier tipo y disimiles entre sí:

```
myList <- list(nombre="Pedro", valorLogico=TRUE,
               unNumero=3, unVector=c(4,7,9),
               unaMatriz=matrix(c(1,0,1,1), nrow=2))

myList
$nombre
[1] "Pedro"

$valorLogico
[1] TRUE

$unNumero
[1] 3

$unVector
[1] 4 7 9

$unaMatriz
     [,1] [,2]
[1,]    1    1
[2,]    0    1
```

¿No les parece conocido esto?. Es que los data.frame son un caso especial de listas en el cual cada elemento es un vector de la misma longitud.
Volviendo a las listas, se puede acceder a cada elemento mediante su nombre:

```
myList$unaMatriz
     [,1] [,2]
[1,]    1    1
[2,]    0    1
```

O bien mediante dobles corchetes"[[ ]]" referenciando al índice del elemento:

```
myList[[5]]
     [,1] [,2]
[1,]    1    1
[2,]    0    1
```

Con corchetes simples accedo la sublista:

```
myList[5]
$unaMatriz
     [,1] [,2]
[1,]    1    1
[2,]    0    1
```

Observen la diferencia, con corchetes simples devolvemos un objeto tipo lista (incluye el nombre de la componente por ejemplo), pero con dobles obtenemos el objeto almacenado:

```
class(myList[1])
[1] "list"

class(myList[[1]])
[1] "character"
```

Puedo crear la lista sin nombre y asignárselos después:

```
myList<- list("Juan", "25", FALSE)
myList
[[1]]
[1] "Juan"

[[2]]
[1] "25"

[[3]]
[1] FALSE

names(myList)<- c("nombre", "numero", "booleano")
myList
$nombre
[1] "Juan"

$numero
[1] "25"

$booleano
[1] FALSE
```

length() me permite conocer el tamaño de la lista:

```
length(myList)
[1] 3
```

Puedo concatenar listas también:

```
myList2 <- list(matrix(c(1,0,1,1), nrow=2),
                matrix(c(1,1,1,0), nrow=2))
myList3 <- c(myList, myList2)
myList3
$nombre
[1] "Juan"

$numero
[1] "25"

$booleano
[1] FALSE

[[4]]
     [,1] [,2]
[1,]    1    1
[2,]    0    1

[[5]]
     [,1] [,2]
[1,]    1    1
[2,]    1    0
```

Un poco más de R

3.1. Utilizando funciones en R

R tiene muchas funciones predefinidas, pero muchas veces es útil definir nuestras propias funciones. Definir funciones en R es muy fácil:

```
mySuma <- function(x,y) {
       x+y
}
```

Acabamos de definir una función que se llama *mySuma* y que acepta dos parámetros, "x" e "y". El resultado que devuelve es la suma de los dos parámetros pasados al invocarla:

```
mySuma(10,25)
[1] 35
```

En R no hace falta el uso de un comando del tipo "return" para indicar que valor debe devolver la función. En R, cualquier función devuelve el último objeto que se llame o invoque durante su ejecución. En el caso de la función *mySuma*, el resultado de la suma.
Si quiero ver cómo está formada mi función, la llamo sin parámetros:

```
mySuma
function(x,y) {
    x+y
}
```

Las funciones en R son muy interesantes. A efectos prácticos, son un objeto más del sistema, así que las puedo asignar a variables:

```
algunaFuncion <- mySuma

algunaFuncion(2,3)
[1] 5

algunaFuncion
function(x,y) {
    x+y
}
```

Entonces, puedo hacer una función que tome como parámetros otras funciones:[24]

```
myResta <- function(x,y) {
  x-y
}
myOperacion <- function(x,y,cualquierOtraFuncion){
  cualquierOtraFuncion (x,y)
}
myOperacion(2,3,mySuma)
myOperacion(2,3,myResta)
```

Fijémonos que la función *myOperacion* toma el tercer parámetro (la función a aplicar) y lo ejecuta con el primer y segundo parámetro.
Es interesante que, gracias a la vectorización, aunque no lo pensamos, funcionan igual cosas como estas:

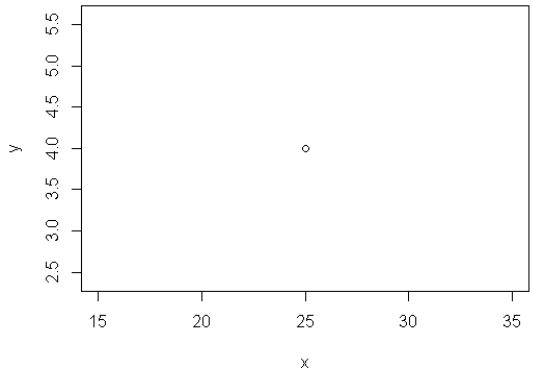
```
secuencia <- seq(1:20)
mySuma(secuencia, 100)
[1] 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116
[17] 117 118 119 120
```

¡¡¡La función aplica vectorización y reciclado en forma automática!!!!, sumó 100 a cada elemento del vector “secuencia”.

Hasta ahora nuestra función es "muy matemática" pero el concepto de función en R, como en casi cualquier otro lenguaje de programación, está pensado desde el punto de vista algorítmico.

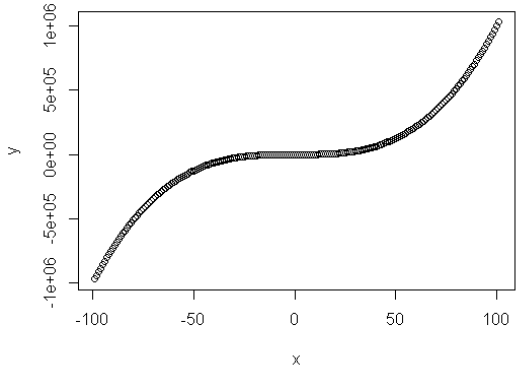
```
myGraficador <- function(x,y){
  plot(x,y)
}
myGraficador(25,4)
```

[24] Básicamente, así funcionan *sapply()* y *tapply()*.



Que es una función algorítmica. Mediante vectorización:

```
myGraficador(seq(0:200)-100, (seq(0:200)-100)^3)
```

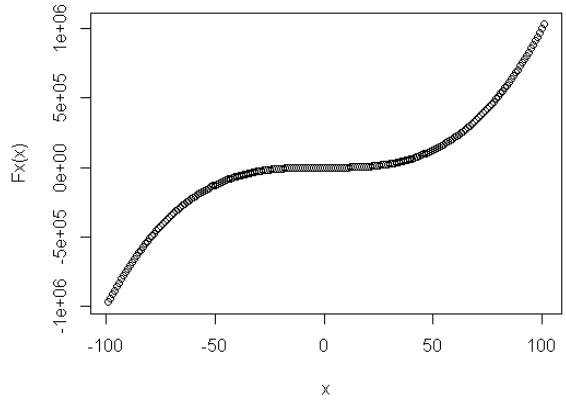


Si lo pensamos, *myGraficador()* es una función horrible. Le tenemos que pasar los valores de las abscisas y de las ordenadas para graficar, pero no le podemos pasar un conjunto de valores y una función para aplicarles. Arreglémoslo un poco:

```
myGraficadorDeFunciones <- function(x, Fx){
  plot(x,Fx(x))
}
```

Listo, tenemos un graficador que, pasándole el dominio ("x") y una función a aplicar ("Fx") genera la gráfica correspondiente en 2D. Definamos ahora una función cúbica:

```
myCubica <- function(x){
  x^3
}
myGraficadorDeFunciones(seq(0:200)-100, myCubica)
```



Sigue siendo muy fea, ya que a myCubica() no le puedo pasar parámetros. Redefinámosla:

```
myCubica <- function(x, coefa, coefb, coefc, coefd){
  coefa*(x^3) + coefb*(x^2) + coefc*x + coefd
}
myGraficadorDeFunciones(seq(0:200)-100, myCubica)
Error en coefa * (x^3) : coefa' esta perdido
```

¿Qué pasó?. Nos tiró un error porque le agregamos nuevos parámetros a myCubica() y no los estamos pasando desde myGraficadorDeFunciones(). Redefinamos esta última:

```
myGraficadorDeFunciones <- function(x, Fx, ...){
  plot(x,Fx(x, ...))
}
```

Con "..." le decimos a R que myGraficadorDeFunciones puede tomar un número indeterminado y no obligatorio de parámetros adicionales a los que si declaramos obligatoriamente (x y Fx), y con el mismo operador podemos hacer uso de ellos dentro de la función. Veamos su ejecución:

```
myGraficadorDeFunciones(seq(0:200)-100,  myCubica,  -3,20,0,-
4)
```

Las funciones en R permiten el pasaje de parámetros nombrados. Por ejemplo, definamos esta función:

```
potencia <- function(base, exponente){
  base^exponente
}
```

Dada esta función, la puedo llamar de la manera convencional, pasando los parámetros en orden:

```
potencia(10,2)
[1] 100
```

Pero no tiene por qué ser así, puedo pasarle los parámetros desordenados siempre y cuando los pase con nombre:[25]

```
potencia(exponente=2, base=10)
[1] 100
```

También puedo definir una función con valores de parámetros por defecto, que hacen opcional el ingreso de los mismos:

```
potencia <- function(base, exponente=2){
  base^exponente
}
```

Si la llamo sin declararle el exponente:

```
potencia(base=10)
[1] 100
```

[25] Observemos el detalle que para pasar valores a parámetros si uso el operador "=", en vez del operador "<-"

Y, ya que base es el primer parámetro, puedo omitir su nombre:[26]

```
potencia(10)
[1] 100
```

3.2. Guardando nuestras funciones

Obviamente, si tenemos que escribir las funciones cada vez que las queremos usar, su uso sería bastante limitado. R nos permite definir scripts, archivos de texto plano con extensión ".R" que contienen código R. Así que para grabar nuestras funciones y tenerlas disponibles cuando queramos, basta crear un archivo de texto, escribir ahí las definiciones de las funciones, y guardarlo con extensión ".R" Por ejemplo, creemos el archivo "funcionesPrueba.R" con cualquier editor de texto (o desde el panel "Source" del RStudio) [27]y escribamos lo siguiente:

```
potencia <- function(base, exponente=2){
  base^exponente
}

raiz <- function(base, indice=2){
  base^(1/indice)
}
```

Guardemos el archivo en la carpeta "sources" de nuestro directorio de trabajo, y ejecutemos el siguiente código:

```
rm(list=ls())
character(0)
source("..\\sources\\funcionesPrueba.R")
ls()
```

[26] Igual tengamos cuidado con esto. En general, conviene declarar, cuando se crea una función, todos los parámetros obligatorios primero y después los parámetros con valores por defecto. Pero ante la duda, al momento de llamar a la función, nombremos el parámetro.

[27] En estas cosas vemos como conviene utilizar un IDE integrado como RStudio en vez de solamente la consola básica de R.

```
[1] "potencia" "raiz"
```

Llamamos a *rm()* para asegurar que no tenemos ninguna función con este nombre creada, con source*()* leemos el archivo recién creado, y con ls() mostramos los objetos disponibles en la sesión (apareciendo las nuevas funciones). Probemos usar alguna:

```
raiz(10,2)
[1] 3.162278
```

Excelente, funcionan correctamente.

3.3. Armando scripts

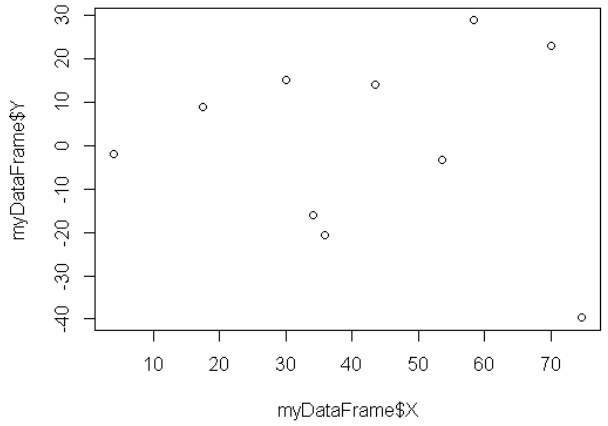
El archivo que creamos solo tenía cargado definiciones de funciones, pero también puedo alojar allí código que quiero que se ejecute al leer el archivo de script. Creemos en nuestro directorio script el archivo "scriptPrueba.R" y carguémosle el siguiente contenido:

```
# Script de prueba en R
# Creamos un data.frame
myDataFrame <- data.frame(runif(10,0,100), runif(10,-50,50))
names(myDataFrame) <- c("X","Y")
plot(myDataFrame$X, myDataFrame$Y)
View(myDataFrame)
```

Ahora ejecutemos el comando *source()*:[28]

```
source("scriptPrueba.R")
```

[28] Como mi directorio de trabajo es "script", no es necesario pasarle ninguna ruta, basta con el nombre del archivo.



	X	Y
1	74.701114	-39.728437
2	35.819991	-20.666453
3	17.464645	8.962565
4	34.191171	-16.073070
5	53.537047	-3.202527
6	29.998742	14.988526
7	58.320702	28.838195
8	43.490644	13.963918
9	4.039695	-1.901415
10	69.998986	23.034759

Gráficos con R

4.1. Trabajando con el paquete graphics

Vamos a presentar una pequeña introducción al uso de gráficos con R. Para ello, vamos a necesitar el archivo de datos "datosPoblacionMaraton.csv"[29]

Arranquemos nuestro directorio de trabajo:

```
setwd("D://Proyectos//IOR//scripts")
```

Y limpiemos la memoria de trabajo:

```
rm(list=ls())
```

Comencemos cargando el archivo "datosPoblacionMaraton.csv" en la variable "dPM":

```
dPM <- read.csv("..//data//datosPoblacionMaraton.csv")
```

Es una buena práctica revisar el data frame cargado antes de ponernos a trabajar. Veamos su estructura:

```
str(dPM)
```

```
'data.frame':      6000 obs. of  7 variables:
 $ X                            : int  1 2 3 4 5 6 7 8 9 10 ...
 $ sexo                         : Factor w/ 2 levels "F","M": 1 2 2
1 1 2 2 2 1 2 ...
 $ altura                       : num  1.69 1.71 1.71 1.71 1.89 ...
 $ peso                         : num  77.9 79.5 79.6 79.8 88.1 ...
 $ edad                         : num  45.2 47.1 41.6 50.6 43.2 ...
 $ frecuenciaSemanalEntrenamiento: int  3 4 0 4 3 2 2 3 3 4 ...
 $ tiempo21KMMin                : num  98.1 79.8 160 82.5 122.2 ...
```

Veamos un poco algunos registros:

```
head(dPM)
```

```
  X sexo   altura     peso     edad frecuenciaSemanalEntrenamiento
```

[29] Se puede descargar desde "http://www.modelizandosistemas.com.ar/p/optimizacion-con-r.html". Una vez descargado, hay que copiarlo en la carpeta "data" de nuestro proyecto.

```
1 1    F 1.686767 77.86853 45.23844                                3
2 2    M 1.707745 79.50636 47.10703                                4
3 3    M 1.708544 79.56911 41.55911                                0
4 4    F 1.710859 79.75118 50.57101                                4
5 5    F 1.891673 88.08078 43.18842                                3
6 6    M 1.843027 83.90220 39.16556                                2
  tiempo21KMMin
1      98.14615
2      79.82995
3     159.99911
4      82.46038
5     122.19544
6     136.86943
```

Podemos hacer una visualización completa también:

```
View(dPM)
```

6000 observations of 7 variables

	X	sexo	altura	peso	edad	frecuenciaSemanalEntrenamiento	tiempo21KMMin
1	1	F	1.686767	77.86853	45.23844	3	98.14615
2	2	M	1.707745	79.50636	47.10703	4	79.82995
3	3	M	1.708544	79.56911	41.55911	0	159.99911
4	4	F	1.710859	79.75118	50.57101	4	82.46038
5	5	F	1.891673	88.08078	43.18842	3	122.19544
6	6	M	1.843027	83.90220	39.16556	2	136.86943
7	7	M	1.690187	78.13413	32.36071	2	110.35485
8	8	M	1.752911	76.44902	46.47301	3	114.72397
9	9	F	1.871720	86.35375	47.52661	3	123.78147
10	10	M	1.875104	86.64532	42.24352	4	99.32822
11	11	M	1.681953	70.84323	34.35950	2	122.53565
12	12	M	1.970735	95.10433	27.52454	4	96.13051

Displayed 1000 rows of 6000 (5000 omitted)

OK, ya vimos cómo está formado el conjunto de datos, familiaricémonos con su contenido:

```
summary(dPM)
```

```
       X         sexo          altura           peso             edad      
 Min.   :   1   F:2995   Min.   :1.482   Min.   : 56.31   Min.   :12.15  
 1st Qu.:1501   M:3005   1st Qu.:1.668   1st Qu.: 74.82   1st Qu.:38.18  
 Median :3000            Median :1.719   Median : 78.30   Median :42.49  
 Mean   :3000            Mean   :1.741   Mean   : 79.02   Mean   :42.55  
 3rd Qu.:4500            3rd Qu.:1.805   3rd Qu.: 82.56   3rd Qu.:46.99  
 Max.   :6000            Max.   :2.177   Max.   :114.81   Max.   :66.94  

 frecuenciaSemanalEntrenamiento  tiempo21KMMin   
 Min.   :0.00                   Min.   : 60.27  
 1st Qu.:1.00                   1st Qu.:101.10  
 Median :2.00                   Median :119.31  
 Mean   :2.25                   Mean   :121.44  
 3rd Qu.:3.00                   3rd Qu.:139.68  
 Max.   :4.00                   Max.   :201.68  
```

Bueno, comencemos a trabajar. El último campo del data frame es el tiempo que tardó cada maratonista en recorrer los 21KM. En general, se

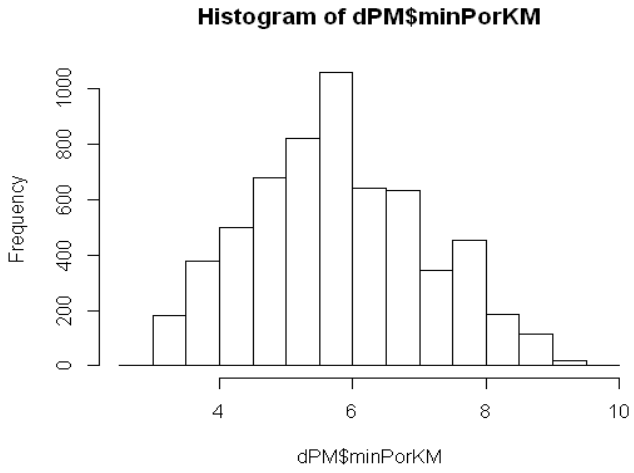
suele utilizar como medida de performance el tiempo (en minutos) que se tarda en recorrer 1KM, así que creemos ese campo:

```
dPM$minPorKM <- dPM$tiempo21KMMin/21
```

Veamos ahora como se distribuyeron las velocidades:

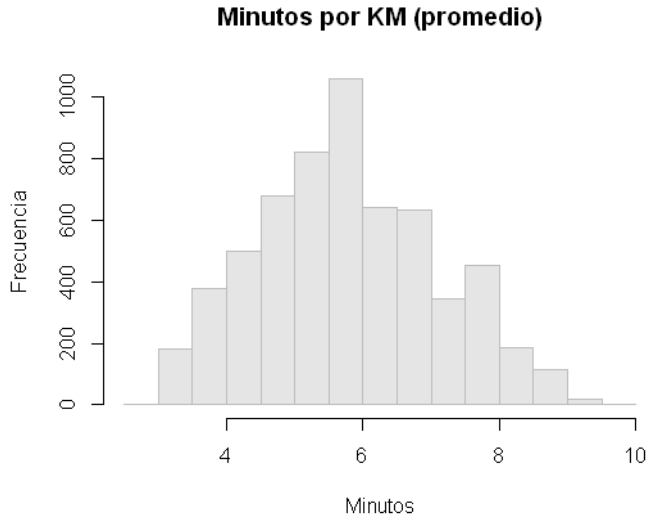
```
summary(dPM$minPorKM)
hist(dPM$minPorKM)
```

```
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  2.870   4.814   5.682   5.783   6.651   9.604
```



Mejoremos un poquito el gráfico

```
hist_minPorKM <- hist(dPM$minPorKM, col = "gray90",
border="gray", main = "Minutos por KM (promedio)", xlab =
"Minutos", ylab = "Frecuencia", axes = TRUE)
```



Fíjense que guardamos el histograma en una variable. Y es que, como decíamos en anteriores capítulos, casi todo en R es un objeto, y todos los objetos se pueden guardar en variables. Si averiguamos su clase:

```
class(hist_minPorKM)
```

```
[1] "histogram"
```

Como todo objeto, podemos analizar su estructura:

```
str(hist_minPorKM)
```

```
List of 7
 $ breaks     : num [1:16] 2.5 3 3.5 4 4.5 5 5.5 6 6.5 7 ...
 $ counts     : int [1:15] 2 182 379 498 676 819 1056 641 633 342
...
 $ intensities: num [1:15] 0.000667 0.060667 0.126333 0.166 0.225333
...
 $ density    : num [1:15] 0.000667 0.060667 0.126333 0.166 0.225333
...
 $ mids       : num [1:15] 2.75 3.25 3.75 4.25 4.75 5.25 5.75 6.25
6.75 7.25 ...
 $ xname      : chr "dPM$minPorKM"
 $ equidist   : logi TRUE
 - attr(*, "class")= chr "histogram"
```

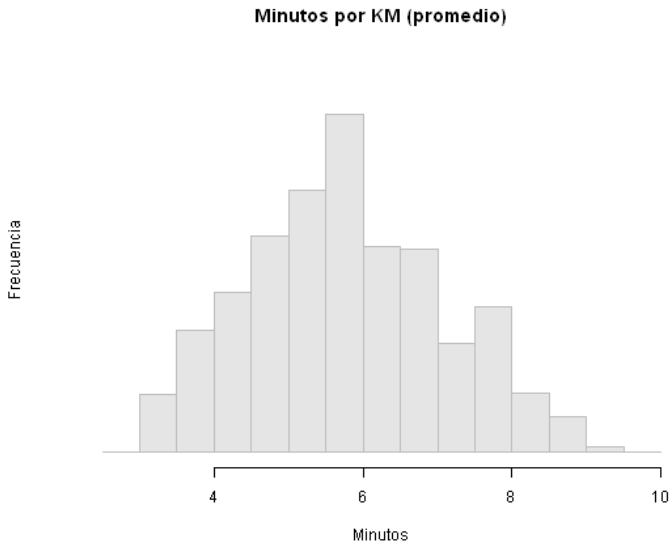
Y claro, sabiendo su estructura podemos utilizarla para otras cosas:

```
View(data.frame(hist_minPorKM$mids, hist_minPorKM$counts))
```

	hist_minPorKM.mids	hist_minPorKM.counts
1	2.75	2
2	3.25	182
3	3.75	379
4	4.25	498
5	4.75	676
6	5.25	819
7	5.75	1056
8	6.25	641
9	6.75	633
10	7.25	342
11	7.75	453
12	8.25	186

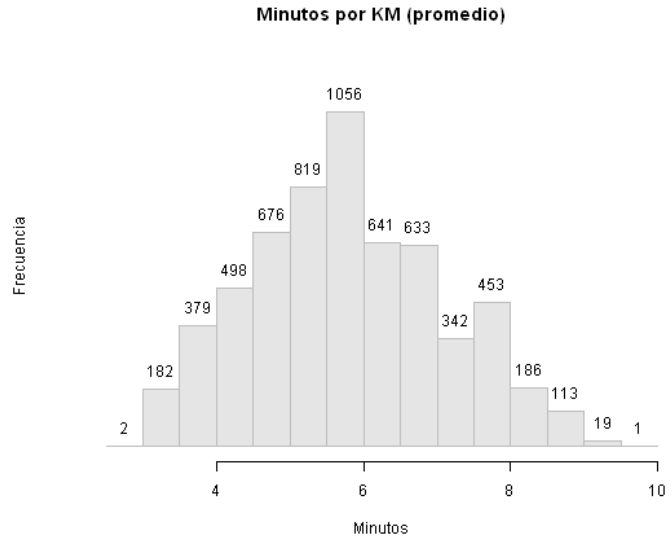
Pero sigamos mejorando el gráfico. Vamos a recrearlo un poco. Ojo cuando utilicemos la función *par()*, guardemos antes los valores originales y restaurémoslos después de usarlos:

```
par_cex <- par("cex") # Guardamos el valor de "cex"
par(cex=0.75) # Reducimos la proporción de la fuente
par_yaxt <- par("yaxt") # Guardamos el valor de "yaxt"
par(yaxt="n") # Quitamos el eje Y
par_bty <- par("bty") # Guardamos el valor de "bty"
par(bty = "n") # Sacamos el borde del gráfico


hist_minPorKM <- hist(dPM$minPorKM, col = "gray90",
border="gray",  main = "Minutos por KM (promedio)", xlab =
"Minutos", ylab = "Frecuencia", axes = TRUE, labels = FALSE,
ylim = c(0,1200))

# Revertimos los cambios, devolviendo a las variables los
valores originales.
par(cex=par_cex)
par(yaxt=par_yaxt)
par(bty = par_bty)
```



par permite acceder a valores globales de configuración de gráficos. Si no revertimos los cambios, el próximo gráfico saldrá con la misma configuración.
Agreguemos los valores de frecuencia:

```
text(hist_minPorKM$mids, hist_minPorKM$counts,
hist_minPorKM$counts, adj=c(.5,-1), cex = 0.75)
```



El primer parámetro pasado a *text()* le dice la posición en X en la que tiene que escribir, el segundo la posición en Y, el tercero el texto a escribir, el cuarto el desplazamiento respecto de los valores X e Y, y el quinto reduce el tamaño de la letra. Como quiero poner el valor que corresponde a cada barra encima de esta, X = hist_minPorKM$mids (el centro de cada barra) e Y = hist_minPorKM$counts (la altura de la barra).

Superpongámosle ahora un polígono de frecuencia. Primero, calculo el ancho de clase del histograma:

```
hist_minPorKM.anchoClase <-
hist_minPorKM$mids[2]- hist_minPorKM$mids[1]
```

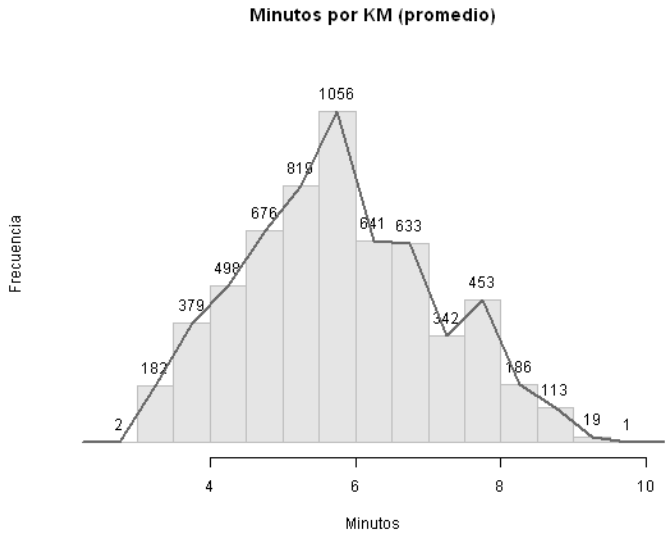
Calculo los vectores de coordenadas horizontales y verticales:

```
xx  <-  c(  hist_minPorKM$mids[1]-hist_minPorKM.anchoClase,
hist_minPorKM$mids,
tail(hist_minPorKM$mids,1)+hist_minPorKM.anchoClase )
yy <- c(0, hist_minPorKM$counts, 0)
```

El vector "xx" tiene como primer y último elementos valores, por afuera del rango de graficación del histograma (al primer elemento le resto un acho de clase, y al último, obtenido con *tail()*, le sumo un ancho de clase), el resto de los valores son los centros de cada clase del histograma. El vector "yy" inicia y termina con 0, y el resto de los valores son la frecuencia absoluta de cada clase del histograma.
Agrego el polígono al histograma:

```
lines(xx, yy, lwd=2, col = "royalblue")
```

Donde “lwd” es el ancho de la línea.

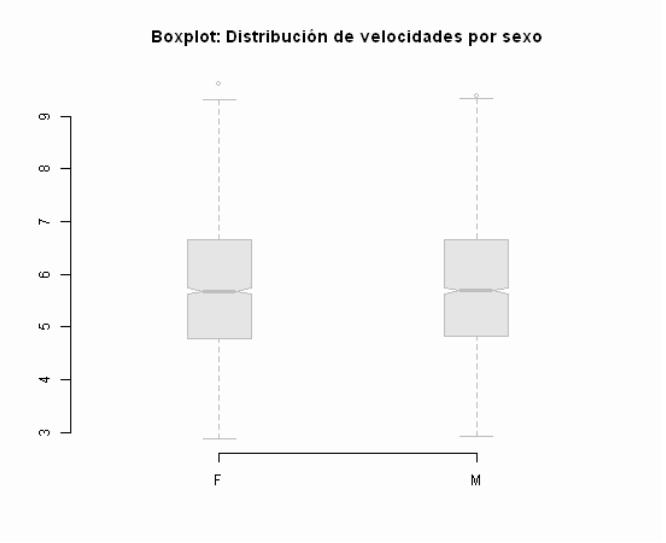


Volvamos ahora a nuestro data frame. ¿Cuantos valores cayeron por debajo de la media?:

```
length(dPM$minPorKM[dPM$minPorKM < mean(dPM$minPorKM)])
```

Tendrá alguna relación el sexo con la velocidad promedio:

```
par_cex <- par("cex")
par(cex=0.75)
par_bg <- par("bg")
par(bg = "lightyellow") # Color del fondo
par_yaxt <- par("yaxt")
par(yaxt="s") #Agrego el eje y
par_bty <- par("bty")
par(bty = "n")

box_minVsSexo <- boxplot(minPorKM ~ sexo, data = dPM, col =
"gray90", border = "grey",  notch=TRUE, boxwex=0.25 )
title("Boxplot: Distribución de velocidades por sexo")

par(cex=par_cex)
par(bg = par_bg)
par(yaxt = par_yaxt)
par(bty = par_bty)
```

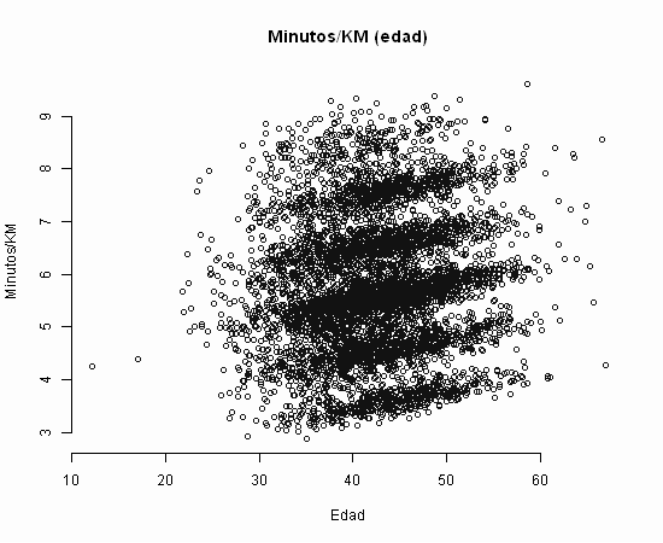


Observen que restauré los valores de *par*. Como con cualquier otro objeto, puedo consultar la variable en la que almacené mi boxplot:

```
str(box_minVsSexo)
```

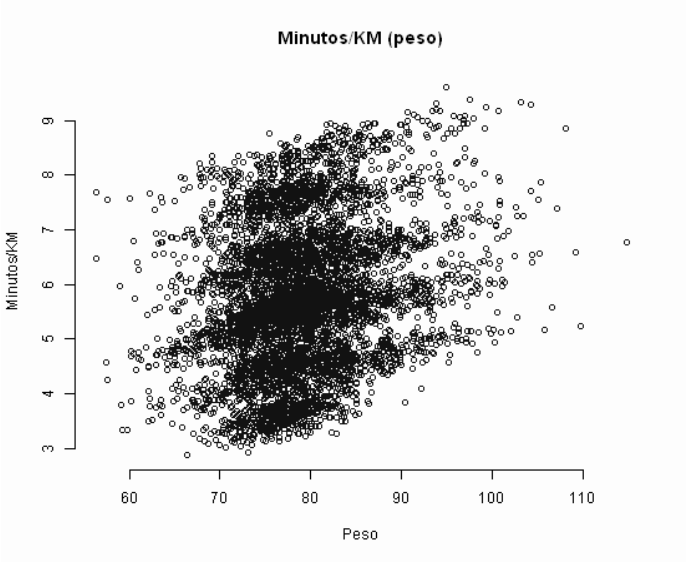
Posiblemente la velocidad esté relacionada con la edad:

```
par_cex <- par("cex")
par(cex=0.75)
par_bg <- par("bg")
par(bg = "lightyellow")
par_bty <- par("bty")
par(bty = "n")

plot_minVsEdad <- plot(dPM$edad, dPM$minPorKM, main =
"Minutos/KM (edad)", xlab="Edad", ylab="Minutos/KM",
col="blue")

par(cex=par_cex)
par(bg = par_bg)
par(bty = par_bty)
```



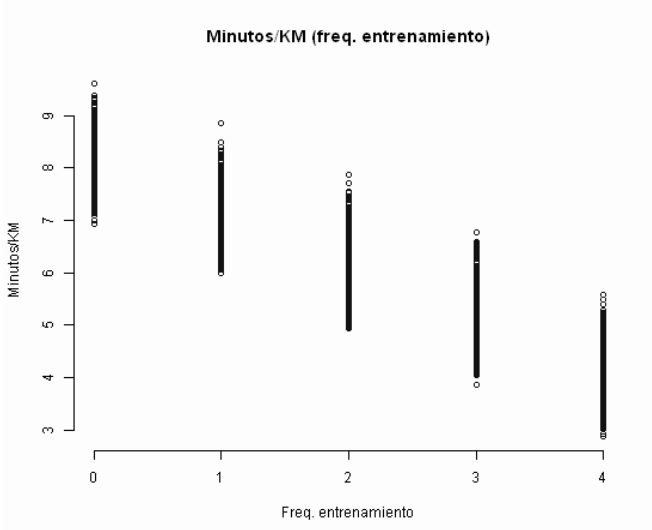
¿Y con el peso?:

```
par_cex <- par("cex")
par(cex=0.75)
par_bg <- par("bg")
par(bg = "lightyellow")
par_bty <- par("bty")
par(bty = "n")

plot_minVsPeso <-plot(dPM$peso, dPM$minPorKM, main =
"Minutos/KM (peso)", xlab="Peso", ylab="Minutos/KM",
col="blue")

par(cex=par_cex)
par(bg = par_bg)
par(bty = par_bty)
```



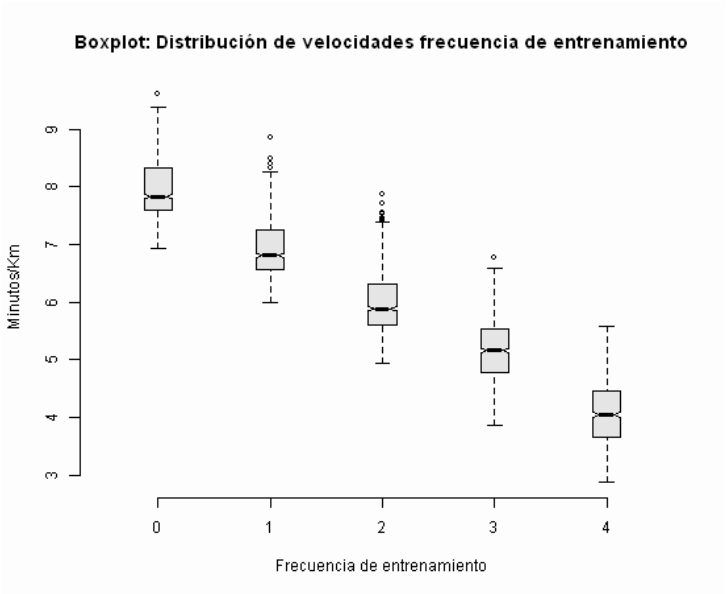
¿Y cómo varía con la frecuencia de entrenamiento?:

```
par_cex <- par("cex")
par(cex=0.75)
par_bg <- par("bg")
par(bg = "lightyellow")
par_bty <- par("bty")
par(bty = "n")

plot_minVsFreq <-plot(dPM$frecuenciaSemanalEntrenamiento,
dPM$minPorKM, main = "Minutos/KM (freq. entrenamiento)",
xlab="Freq. entrenamiento", ylab="Minutos/KM", col="blue")

par(cex=par_cex)
par(bg = par_bg)
par(bty = par_bty)
```



Un gráfico medio feo, mejor un boxplot:

```
par_cex <- par("cex")
par(cex=0.75)
par_bg <- par("bg")
par(bg = "lightyellow")
par_yaxt <- par("yaxt")
par(yaxt="s")
par_xaxt <- par("xaxt")
par(xaxt="s")
par_bty <- par("bty")
par(bty = "n")

box_minVsFrecuencia <- boxplot(minPorKM ~
frecuenciaSemanalEntrenamiento, data = dPM, col = "gray90",
notch=TRUE, boxwex=0.25 )
title(main="Boxplot: Distribución de velocidades frecuencia
de entrenamiento")
title(xlab="Frecuencia de entrenamiento")
title(ylab="Minutos/Km")

par(cex=par_cex)
par(bg = par_bg)
par(yaxt = par_yaxt)
par(xaxt = par_xaxt)
par(bty = par_bty)
```



Aparentemente hay alguna vinculación entre la velocidad promedio, la frecuencia de entrenamiento y las características fisiológicas. Armemos un modelo de regresión con la función *lm*():

```
lm.vel <- lm(dPM$minPorKM ~ dPM$peso + dPM$edad +
dPM$frecuenciaSemanalEntrenamiento)
summary(lm.vel)

Call:
lm(formula = dPM$minPorKM ~ dPM$peso + dPM$edad +
dPM$frecuenciaSemanalEntrenamiento)

Residuals:
     Min       1Q   Median       3Q      Max
-0.86541 -0.32015 -0.02829  0.32447  0.88141

Coefficients:
                                    Estimate Std. Error t value
Pr(>|t|)
(Intercept)                        3.2833367  0.0663475   49.49
<2e-16 ***
dPM$peso                           0.0521247  0.0006957   74.92
<2e-16 ***
dPM$edad                           0.0122465  0.0007130   17.18
<2e-16 ***
dPM$frecuenciaSemanalEntrenamiento -0.9511652  0.0036234 -262.50
<2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.3587 on 5996 degrees of freedom
Multiple R-squared: 0.9255,  Adjusted R-squared: 0.9254
F-statistic: 2.482e+04 on 3 and 5996 DF,  p-value: < 2.2e-16
```

Así como los gráficos son objetos, los modelos de ajuste de datos son también objetos:

```
class(lm.vel)
```

Como cualquier objeto, puedo acceder a su estructura:

```
str(lm.vel)
View(lm.vel$coefficients)
```

	row.names	x
1	(Intercept)	3.28333667
2	dPM$peso	0.05212468
3	dPM$edad	0.01224653
4	dPM$frecuenciaSemanalEntrenamiento	-0.95116523

```
class(lm.vel$coefficients)
```

```
[1] "numeric"
```

Vemos que "`coefficients`" es un vector en el cual la primera componente es el termino independiente, y el resto los coeficientes estimados para la función de velocidad. Sabiendo esto, puedo usar el modelo para predecir valores. Supongamos que quiero saber la velocidad promedio que alcanzará una persona de 75 kg de peso, 25 años de edad y una frecuencia de entrenamiento de 4 días semanales:

```
lm.vel$coefficients[1] + lm.vel$coefficients[2]*75 +
lm.vel$coefficients[3]*25 + lm.vel$coefficients[4]*4
```

Mejor aún, puedo hacerlo así:

```
prediccion_vel <- sum(lm.vel$coefficients * c(1,75,25,4))
prediccion_vel
```

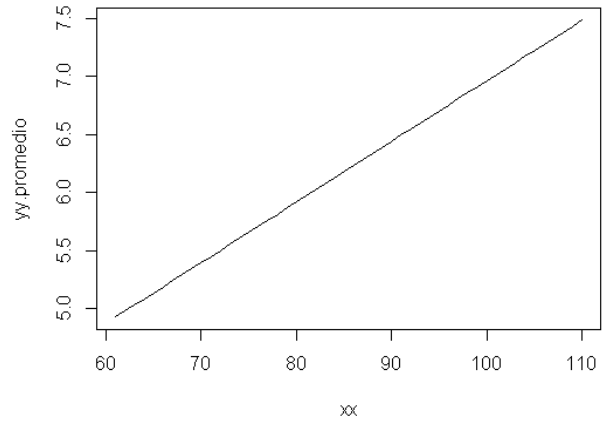
El "1" en la primer componente del vector c(1,75,25,4) sirve para mantener el termino independiente del modelo en un valor fijo (ya que lo multiplico por la constante).
Claro, tiene mucha dispersión mi población, quizás pueda armar un intervalo de predicción:

```
residuos.sd_vel <- sd(lm.vel$residuals)
prediccion_vel_interval <- c(prediccion_vel -
2*residuos.sd_vel, prediccion_vel + 2*residuos.sd_vel)

prediccion_vel_interval
```

```
[1] 2.977059 4.411321
```

Supongamos que queremos graficar la influencia del peso dada una frecuencia de entrenamiento fija (2) y una edad también fija (30), para el rango de peso comprendido entre 61 kg y 110 kg. Creo una variable denominada “xx” para almacenar los pesos, y una familia de variables “yy” para almacenar la velocidad:

```
edad <- 30
frec <- 2
xx <- seq(1:50)+60
yy.promedio <- lm.vel$coefficients[1] +
lm.vel$coefficients[2]*xx + lm.vel$coefficients[3]*edad +
lm.vel$coefficients[4]*frec
yy.min <- yy.promedio - 2*residuos.sd_vel
yy.max <- yy.promedio + 2*residuos.sd_vel

plot(xx, yy.promedio, type = "l")
```



Donde tenemos un gráfico para buscar la velocidad promedio que alcanzará un maratonista respecto de su peso. Obviamente, en un gráfico muy feo, queda para el lector mejorarlo un poco.

4.2. Trabajando con layouts

Vamos a ver ahora como graficar más de un gráfico en, valga la redundancia, el mismo gráfico. Continuando trabajando con los datos almacenados en nuestra variable “dPM”, vamos a graficar, en la misma

ventana, dos “plot chart” que muestren la relación entre "minPorKM" y "peso", y minPorKM" y "edad".

Para comenzar, guardémonos la configuración de todos los parámetros de *par()* antes de hacer algún cambio:

```
par.original <- par(no.readonly = TRUE)
```

Hasta ahora veníamos guardando cada atributo de *par()* en forma individual, pero en realidad es más práctico guardar todos los parámetros, y restaurar toda la configuración original. Con el código anterior, nos aseguramos que guardamos toda la configuración gráfica actual.

Usemos “mfrow” para separar nuestro gráfico en un layout de 2 filas y 1 columna:

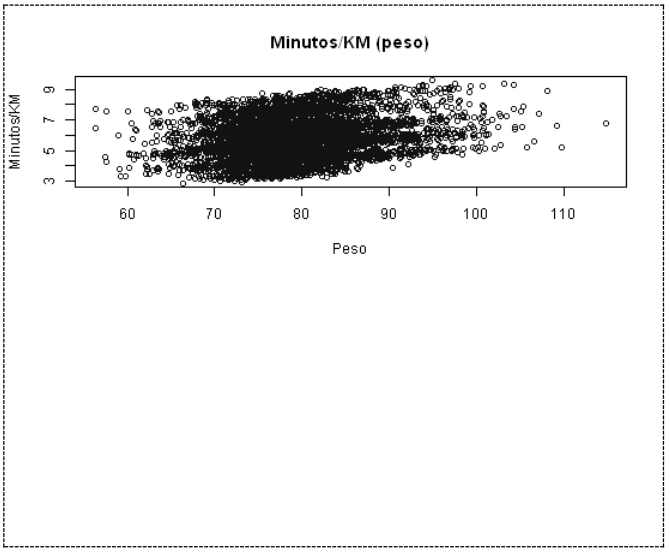
```
par(mfrow=c(2,1))
```

Al parámetro “mfrow” se le debe pasar un vector de dos componentes, donde la primera es la cantidad de filas y la segunda la cantidad de columnas. Ajustemos también el tipo de letra:

```
par(cex=0.75)
```

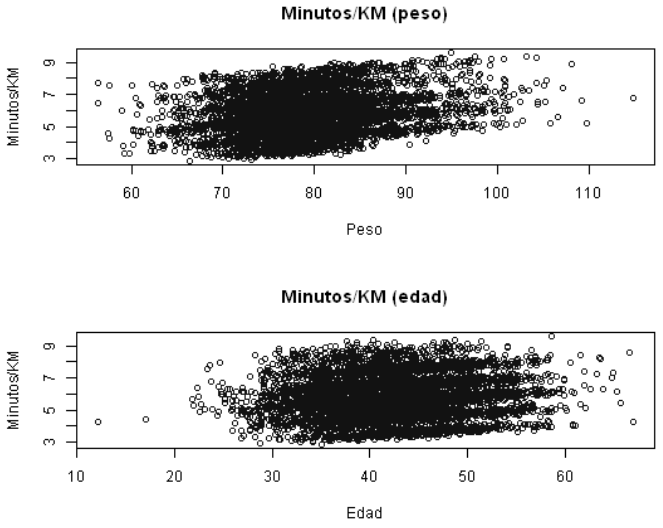
Dibujemos ahora el primer plot:

```
plot_minVsPeso <- plot(dPM$peso, dPM$minPorKM, main =
"Minutos/KM (peso)", xlab="Peso", ylab="Minutos/KM",
col="blue")
```



Observen como la mitad del gráfico es espacio vacío. Dibujemos ahora el segundo:

```
plot_minVsEdad <- plot(dPM$edad, dPM$minPorKM, main =
"Minutos/KM (edad)", xlab="Edad", ylab="Minutos/KM",
col="blue")
```

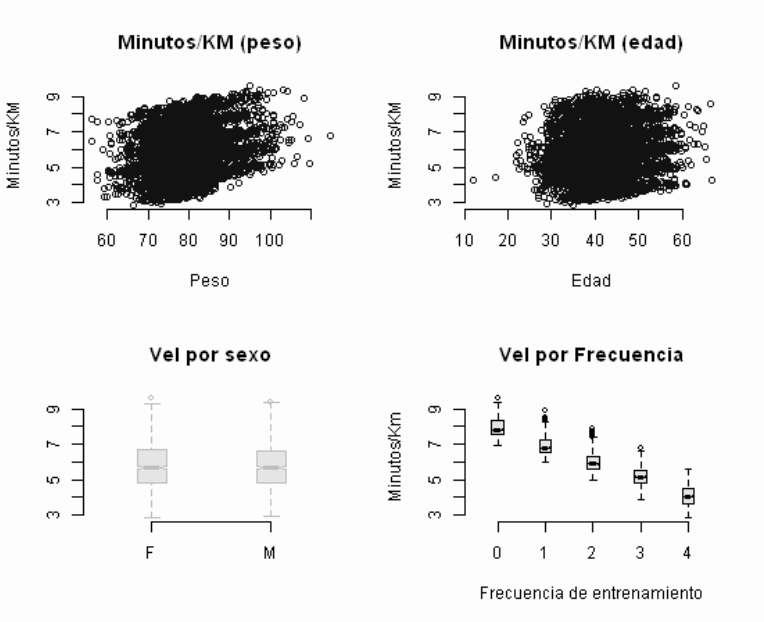


Restauremos ahora la configuración de *par():*

```
par(par.original)
```

Podemos hacer diseños un poco más complejos:

```
par.original <- par(no.readonly = TRUE)

par(mfrow=c(2,2))

par(cex=0.75)

par(bg = "lightyellow")

par(bty = "n")



plot_minVsPeso <- plot(dPM$peso, dPM$minPorKM, main =
"Minutos/KM (peso)", xlab="Peso", ylab="Minutos/KM",
col="blue")

plot_minVsEdad <- plot(dPM$edad, dPM$minPorKM, main =
"Minutos/KM (edad)", xlab="Edad", ylab="Minutos/KM",
col="blue")

box_minVsSexo <- boxplot(minPorKM ~ sexo, data = dPM, col =
"gray90", border = "grey",  notch=TRUE, boxwex=0.25 )

title("Vel por sexo")



box_minVsFrecuencia <- boxplot(minPorKM ~
frecuenciaSemanalEntrenamiento, data = dPM, col = "gray90",
notch=TRUE, boxwex=0.25 )

title(main="Vel por Frecuencia")

title(xlab="Frecuencia de entrenamiento")

title(ylab="Minutos/Km")



par(par.original)
```



Un poco más a alto nivel, puedo utilizar la función *layout()*.[30] Primero, armo un histograma (sin graficarlo) de los campos edad y peso del data.frame dPM:

```
hist.edad <- hist(dPM$edad, plot=FALSE)

hist.peso <- hist(dPM$peso, plot=FALSE)
```

Calculo los rangos de visualización

```
top <- max(c(hist.peso$counts, hist.edad$counts))

edad.range <- c(0,max(dPM$edad))

peso.range <- c(0,max(dPM$peso))
```

Calculo los parámetros gráficos

```
grafico.ancho <- c(3,1) # Ancho relativos de las dos
columnas

grafico.alto <- c(1,3) # Alto relativo de las dos filas
```

[30] El siguiente ejemplo está adaptado de http://gallery.r-enthusiasts.com/RGraphGallery.php?graph=78

```
grafico.nroFilas <- 2

grafico.nroColumnas <- 2
```

Armo la matriz de graficación. La misma es una matriz en la cual cada componente está asociada a la componente correspondiente del layout del gráfico y su valor indica cuál de los gráficos se graficaran en esa coordenada. Para ello, primero defino la secuencia de graficación:

```
grafico.secuenciaGraficacion <- c(2,0,1,3)
```

Estamos diciendo que, dada una matriz de 2 x 2, en (1,1) cae el gráfico número 2, en (1,2) no se grafica nada, en (2,1) va el primer grafico que hagamos y en (2,2) se dibuja el tercero. Ahora si armo la matriz:

```
matrizGraficacion <-
matrix(grafico.secuenciaGraficacion,grafico.nroFilas,
grafico.nroColumnas,byrow=TRUE)
```

Genero el layout:

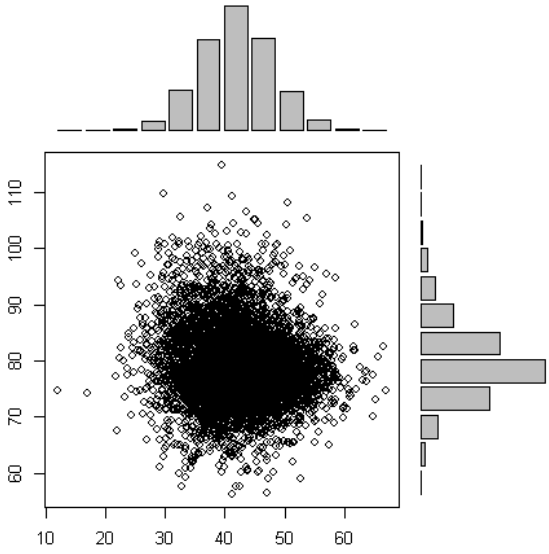
```
layout(matrizGraficacion, grafico.ancho, grafico.alto,
respect = TRUE)
```

Creo los gráficos, con "mar" defino los márgenes

```
par(mar=c(3,3,1,1))

plot(dPM$edad, dPM$peso, xlab = "edad", ylab="peso")

par(mar=c(0,3,1,1))

barplot(hist.edad$counts, axes=FALSE,)

par(mar=c(3,0,1,1))

barplot(hist.peso$counts, axes=FALSE,horiz=TRUE)
```

Restauro *par()*:

```
par(par.original)
```


110
100
90
80
70
60
10
20
30
40
50
60

Implementando nuestras primeras búsquedas

5.1. Acerca de las búsquedas por fuerza bruta

Vamos a comenzar nuestro enfoque en la optimización utilizando R para ejecutar el que quizás es el algoritmo más intuitivo de todos, la fuerza bruta. Es decir, vamos a recorrer una por una todas las alternativas posibles y evaluar cuál es la mejor solución. Supongamos que deseamos encontrar el mínimo de la función $Y=(X+10)^2 – 20$ cuando X solo puede tomar valores enteros, acotados entre -100 y 100. Definimos primero la función de evaluación:

```
funcionCuad <- function(x){
  (-x+10)^2-20
}
```

Y a continuación el algoritmo de fuerza bruta:

```
fuerzaBruta <- function(fn, cotaMin, cotaMax){
  menorValor<- fn(cotaMin)
  solucion <- cotaMin
  for (i in (cotaMin+1):cotaMax){
    valorActual <- fn(i)
    if(valorActual < menorValor){
      menorValor <- valorActual
      solucion <- i
    }
  }
  resultado <- data.frame(solucion, menorValor)
  names(resultado) <- c("Solución", "Valor Objetivo")
  resultado
```

```
}

fuerzaBruta(funcionCuad, -100, 100)

  Solución Valor Objetivo
1       10            -20
```

Bueno, funciona bien, pero el código no aprovecha la expresividad de R[31]. Sabiendo que podemos usar vectorización y reciclado, que implementa los bucles "for" mucho más rápido, podemos hacer algo como esto:

```
fuerzaBruta <- function(fn, cotaMin, cotaMax, maximizar =
FALSE){

  dominio <- seq(cotaMin, cotaMax)

  evaluacion <- fn(dominio)



  indice <- order(evaluacion, decreasing = maximizar)[1]

  resultado <- data.frame(dominio[indice],
evaluacion[indice])

  names(resultado) <- c("Solución", "Valor Objetivo")

  resultado

}

fuerzaBruta(funcionCuad, -100, 100, FALSE)

  Solución Valor Objetivo
1       10            -20
```

Mucho más eficiente y, además, puede maximizar si así lo deseamos[32]. El problema anterior es muy sencillo, podemos darnos el lujo de aplicar fuerza bruta. Obviamente, este enfoque está preparado para problemas de optimización combinatoria con un número finito de soluciones. Por definición, si existen infinitas soluciones posibles, el tiempo de cómputo necesario para evaluarlas todas también es infinito. En caso de

[31] En Python hubiéramos usado la frase "el código no es muy pithonico". Intenté armar una frase equivalente para R, pero "eRrico" queda muy feo.

[32] Y todavía se podría simplificar mucho más el código.

optimización continua, tenemos que fijar un nivel de granularidad máximo (o sea, cual es la magnitud a partir de la cual las variaciones nos parecen indiferentes). Adaptemos nuestro ejemplo para el caso continuo:

```
fuerzaBruta <- function(fn, cotaMin, cotaMax,
maximizar=FALSE, granularidad = 1){

  dominio <- seq(from= cotaMin, to= cotaMax, by=
granularidad)

  evaluacion <- fn(dominio)



  indice <- order(evaluacion, decreasing = maximizar)[1]

  resultado <- data.frame(dominio[indice],
evaluacion[indice])

  names(resultado) <- c("Solución", "Valor Objetivo")

  resultado

}

fuerzaBruta(funcionCuad, -100, 100, FALSE, 0.01)
```

```
  Solución Valor Objetivo
1       10            -20
```

Si quisiéramos armar una versión que trabaje con múltiples variables de decisión, podríamos armar algo como esto:

```
fuerzaBruta <- function(fn, cotaMin, cotaMax,
maximizar=FALSE, granularidad = 1){

  for (i in 1:length(cotaMin)){

    if (i==1){

      auxDominio <- list(seq(from= cotaMin[i], to=
cotaMax[i], by= granularidad))

    }

    else{
```

```
      auxList <- list(seq(from= cotaMin[i], to= cotaMax[i],
by= granularidad))

      auxDominio <- c(auxDominio, auxList)

    }

  }

  dominio <- expand.grid(auxDominio)

  evaluacion <- fn(dominio)



  indice <- order(evaluacion, decreasing = maximizar)[1]

  resultado <- data.frame(dominio[indice,],
evaluacion[indice,])

  names(resultado) <- c(paste(rep("Variable", times =
length(cotaMin)), seq(1:length(cotaMin))), "Valor Objetivo")

  resultado

}
```

Esta función requiere algo más de explicación. La estructura es muy similar a la versión de *fuerzaBruta()* que veníamos utilizando, pero con algunas diferencias sútiles. Veamos primero como trabaja. La función "fn" debe ser una función de una variable, pudiendo ser esta variable vectorial. O sea, si quiero optimizar un problema de dos variables, debería modelarlas mediante un vector de dos componentes. Las variables cotaMin y cotaMax, entonces, deben ser vectores de la misma dimensión que el vector que acepta la función fn. Por esta naturaliza vectorial, ya no podemos construir el dominio mediante un simple cálculo de secuencia. En vez de eso, utilizamos una variable denominada "auxDominio" la cual es una lista de tantos elementos como variables de decisión haya. La misma se rellena en el bucle "for" inicial, en la primer iteración declarándola como lista y en el resto ampliándola mediante *c()*. Elegimos usar una lista y no un data frame ya que el data frame tiene la restricciones que todos los campos deban tener la misma cantidad de registros, algo que no necesariamente se cumple. Por último, una vez generada la lista completa, en la cual cada elemento es una sublista con el dominio de cada variable independiente, usamos *expand.grid()* para

generar un data frame con el producto cartesiano de las mismas. Veamos en detalle ejecutando el siguiente código:

```
   # Prueba de funcionamiento del bucle for. Se inyectan los
valores de los parámetros.

  cotaMin <- c(-10,-5,-1)

  cotaMax <- c(5,5,3)

  granularidad <- 1

  for (i in 1:length(cotaMin)){

    if (i==1){

      auxDominio <- list(seq(from= cotaMin[i], to=
cotaMax[i], by= granularidad))

    }

    else{

      auxList <- list(seq(from= cotaMin[i], to= cotaMax[i],
by= granularidad))

      auxDominio <- c(auxDominio, auxList)

    }

  }

  dominio <- expand.grid(auxDominio)

  auxDominio

  [[1]]
 [1] -10  -9  -8  -7  -6  -5  -4  -3  -2  -1   0   1   2   3   4   5

[[2]]
 [1] -5 -4 -3 -2 -1  0  1  2  3  4  5

[[3]]
[1] -1  0  1  2  3


  dominio

   Var1 Var2 Var3
1   -10   -5   -1
2    -9   -5   -1
3    -8   -5   -1
4    -7   -5   -1
5    -6   -5   -1
6    -5   -5   -1
```

```
7    -4   -5   -1
8    -3   -5   -1
9    -2   -5   -1
10   -1   -5   -1
11    0   -5   -1
12    1   -5   -1
13    2   -5   -1
14    3   -5   -1
15    4   -5   -1
16    5   -5   -1
17  -10   -4   -1
18   -9   -4   -1
19   -8   -4   -1
20   -7   -4   -1
…
```

Como vemos, el bucle for inicial genera todo el conjunto de soluciones a evaluar[33]. El resto de la nueva función de *fuerzaBruta()* es muy similar a la versión anterior, salvo que al momento de devolver los resultados hay que poner varios nombres de variables independientes, que se hacen concatenando con *paste()* una repetición del término "Var" construida con *rep()* y el número de la variable generado con *seq()*. Probemos de ejecutarlo con nuestro problema de optimizar la función *funcionCuad()*:

```
fuerzaBruta(funcionCuad, -100, 100)
```

```
     Variable 1 Valor Objetivo
Var1         10            -20
```

Y probemos ahora con una función de dos variables:

```
funcion2Var <- function(x){

  x[1]*x[1] + 2*x[2]*x[2]



}

fuerzaBruta(funcion2Var, c(-100,-100), c(100,100))
```

```
      Variable 1 Variable 2 Valor Objetivo
20201          0          0              0
```

Grabando la función *fuerzaBruta()* a un script, podemos resolver por este método cualquier problema con variables acotadas en ambos sentidos.

[33] Por cuestiones de espacio se omitieron la mayoría de los resultados.

El gran problema con esta función es que, potencialmente, no termina nunca. Así que deberíamos definirle un parámetro con el tiempo máximo de cómputo, para que alcanzado este, termine de evaluar soluciones.
Pero con esto anulamos la principal característica de la fuerza bruta, la de revisar todos los casos. Así que vamos a solucionar esto con el segundo algoritmo más intuitivo de todos, la búsqueda aleatoria.[34]

5.2. Búsqueda aleatoria

Los algoritmos de búsqueda aleatorias se basan en el mismo principio que la fuerza bruta, pero en vez de evaluar todas las soluciones, evalúan un subconjunto seleccionado al azar. Mientras más grande sea este subconjunto, más cercano a la fuerza bruta estará este algoritmo (y más certera será la solución óptima encontrada).

Implementar un algoritmo de este tipo no requiere mucho más trabajo que el algoritmo de fuerza bruta:

```
busquedaAleatoria <- function(fn, cotaMin, cotaMax,
maximizar=FALSE, cantMuestras = 500){

  for (i in 1:length(cotaMin)){

    if (i==1){

      dominio <- data.frame(runif(cantMuestras, cotaMin[i],
cotaMax[i]))

    }

    else{

      dominio <- cbind(dominio, runif(cantMuestras,
cotaMin[i], cotaMax[i]))

    }

  }

  evaluacion <- fn(dominio)
```

[34] Obviamente, estas afirmaciones son muy subjetivas.

```
  indice <- order(evaluacion, decreasing = maximizar)[1]

  resultado <- data.frame(dominio[indice,],
evaluacion[indice,])

  names(resultado) <- c(paste(rep("Variable", times =
length(cotaMin)), seq(1:length(cotaMin))), "Valor Objetivo")

  resultado

}
```

Como vemos, en vez de evaluar las soluciones siguiendo una secuencia, generamos aquí valores aleatorios dentro del rango permitido para cada variable independiente. Los únicos cambios son la eliminación del parámetro granularidad, el cual es reemplazado por la cantidad de muestras, y la modificación del bucle "for" para generar valores aleatorios mediante *runif()*[35] en vez de todas las combinaciones posibles. Como a priori sabemos la cantidad de valores de cada variable a evaluar, y este es el mismo para todas, no necesitamos usar una lista, sino que alcanza con un data frame. A simple vista, el método es bastante ineficaz, ya que el resultado obtenido en cada evaluación no depende de evaluaciones anteriores (es decir, es un algoritmo que no mejora la solución a medida que avanza en su ejecución), además que la solución devuelta depende de los valores que haya tomado el generado de números aleatorios en esa llamada:

```
busquedaAleatoria(funcion2Var, c(-100, -10), c(100,10))

    Variable 1 Variable 2 Valor Objetivo
439  0.8753577   1.173358       3.519791

busquedaAleatoria(funcion2Var, c(-100, -10), c(100,10))

    Variable 1 Variable 2 Valor Objetivo
220  -2.339311   -1.06756       7.751744

busquedaAleatoria(funcion2Var, c(-100, -10), c(100,10))

    Variable 1 Variable 2 Valor Objetivo
114  -1.049637  -1.634462       6.444671
```

[35] La función runif() nos permite generar muestras de una variable aleatoria con distribución uniforme. Requiere de tres parámetros, que en orden son: el número de muestras a generar, la cota inferior de la distribución y la cota superior.

5.3. Búsqueda dirigida por gradientes

El caso opuesto a la búsqueda aleatoria y a la fuerza bruta es la búsqueda dirigida por "gradientes". Aquí, a partir de una solución inicial (que puede ser generada al azar o basarse en alguna heurística particular), avanzamos de iteración en iteración "moviéndonos" en la dirección que mejor mejore nuestra función objetivo.
Si nuestra función a optimizar es diferenciable, esta dirección se puede obtener mediante el gradiente de la función. Entonces, nuestro algoritmo de búsqueda se podría implementar así:

```
busquedaPorGradiente <- function(fn, gradiente,
solucionInicial, precision, paso=1, iteraciones=100,
minimizar=TRUE){

  solActual <- solucionInicial

  for (i in 1:iteraciones){

    # Me muevo en el sentido de la variación que deseo

    if(minimizar==TRUE){

      solNueva <- solActual - paso * gradiente(solActual)

    }

    else{

      solNueva <- solActual + paso * gradiente(solActual)

    }

    #Chequeo si la variación es significativa

    if (sum((abs(solNueva - solActual))<precision)){

      break

    }

    else{

      solActual <- solNueva

    }

  }
```

```
  resultado <- c(solActual, fn(solActual))

  names(resultado) <- c(paste(rep("Variable", times =
length(solActual)), seq(1:length(solActual))), "Valor
Objetivo")

  resultado

}
```

Esta función requiere como parámetros obligatorios la función a optimizar, la función que genera el gradiente de la función a optimizar, la solución inicial a partir de la cual comenzar la búsqueda, y la precisión deseada.
Ya teniendo definida nuestra función de dos variables:

```
funcion2Var <- function(x){
  x[1]*x[1] + 2*x[2]*x[2]

}
```

Podemos armar una función que devuelva el gradiente de dicha función:

```
gradFuncion2Var <-function(x){
  c(2*x[1], 4*x[2])
}
```

Y la podemos probar:

```
busquedaPorGradiente(funcion2Var, gradFuncion2Var, c(-100, -
10), precision = 0.01)
    Variable 1     Variable 2 Valor Objetivo
 -1.000000e+02  -5.153775e+48   5.312280e+97

busquedaPorGradiente(funcion2Var, gradFuncion2Var, c(-100, -
10), precision = 0.01, paso=0.1)
    Variable 1     Variable 2 Valor Objetivo
 -4.056482e-02  -1.719071e-07   1.645505e-03

busquedaPorGradiente(funcion2Var, gradFuncion2Var, c(-100, -
10), precision = 0.01, paso=0.1, iteraciones=10000)
Variable 1     Variable 2 Valor Objetivo
 -4.056482e-02  -1.719071e-07   1.645505e-03
```

Como suele suceder con este algoritmo, el paso debe ser de una magnitud cercana a la precisión que deseamos para llegar a resultados cercanos a algún óptimo local en un número de iteraciones aceptable.

Algoritmos para programación lineal

6.1. Introducción a la Programación Lineal

Los problemas de Programación Lineal son problemas de optimización convexa en el cual tanto la función objetivo como las restricciones son funciones lineales. Si bien su definición parece un tanto restrictiva, en realidad muchos de los problemas estudiados en el mundo real entran dentro de esta categoría, por ejemplo:

- Asignación de rutas de transporte
- Balance de líneas de producción
- Asignación de procesos a equipos de procesamiento
- Definición de mix de productos
- Definición de mezcla de componentes para productos
- Nivelación de recursos

Un problema de programación lineal se caracteriza porque su conjunto de soluciones factibles forma un politopo (o sea, la generalización de un polígono para n dimensiones). Por su propia naturaleza, al estar los gradientes definidos en una solo dirección, los puntos extremos se encuentran en la frontera del politopo, más específicamente en algunos de sus vértices. Esto permitió el desarrollo del muy eficiente Método Simplex para su resolución, que se aprovecha de esta particularidad para buscar soluciones solo en dichos puntos.

6.2. Cómo resolver problemas de PL

Dentro de los paquetes "estándares" que se distribuyen con R, tenemos el paquete "boot". Entre otras, este paquete tiene una función denominada "simplex" que se encarga de aplicar el algoritmo de dicho nombre. Para usarla, primero carguemos el paquete:[36]

```
library(boot)
```

[36] Como vimos anteriormente, los paquetes (código "empaquetado", valga la redundancia) se cargan con *library()*, no con *source()*. Con *source()* solo cargamos código guardado en scripts

La función “simplex” se encarga de resolver problemas de programación lineal. Permite únicamente el uso de solo variables continuas y fuerza las soluciones no negativas. Veamos un ejemplo de uso:

```
coefFuncionZ <- c(1,2,3)

matrizRestriccionesMenorOIgual <-
matrix(c(0,2,3,5,0,3),2,3,byrow=TRUE)

vectorTermIndep <- c(100,100)

mySimplex <- simplex(coefFuncionZ, A1 =
matrizRestriccionesMenorOIgual, b1= vectorTermIndep,
maxi=TRUE )

mySimplex
```

```
Linear Programming Results

Call : simplex(a = coefFuncionZ, A1 =
matrizRestriccionesMenorOIgual,
    b1 = vectorTermIndep, maxi = TRUE)

Maximization Problem with Objective Function Coefficients
x1 x2 x3
 1  2  3


Optimal solution has the following values
x1 x2 x3
20 50  0
The optimal value of the objective  function is 120.
```

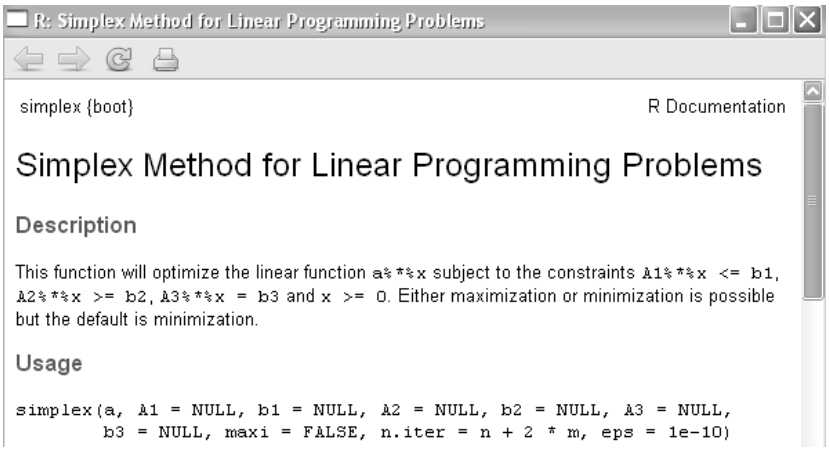
Con el código anterior, resolvimos el siguiente problema:

$$\max \quad z = x_1 + 2x_2 + 3x_3$$

$$\begin{aligned} s.a \qquad & 2x_2 + 3x_3 \leq 100 \\ 5x_1 + \quad & + 3x_3 \leq 100 \\ & x_1, x_2, x_3 \geq 0 \end{aligned}$$

simplex() es una función compleja para utilizar debido al gran número de parámetros que requiere. Veamos su ayuda:

```
?simplex
```



Como vemos, amén del vector "a" de los coeficientes de la función objetivo, tenemos parámetros del tipo matrix para los coeficientes de las restricciones (A1, A2 y A3, dependiendo si son del tipo ≤, ≥ o =, respectivamente), coeficientes para los términos independientes de las restricciones (b1, b2 y b3, con la misma lógica que los An), un parámetro maxi que si vale TRUE el solver se ejecuta maximizando (si se omite o vale FALSE se ejecuta minimizando), y parámetros de cálculo algorítmicos como “n.iter” para el número de iteraciones y “eps” para la precisión decimal al momento de comparar valores.
Entonces, como mínimo, para utilizar esta función, debemos definir un vector de coeficientes para la función objetivo, una o más matrices de coeficientes de restricciones (A1 para las restricciones ≤, A2 para las ≥, y A3 para las =) y el término independiente respectivo para cada tipo de restricción, y opcionalmente una indicación acerca de si queremos minimizar.

Como objeto hecho y derecho, podemos acceder a los atributos del objeto generado por simplex:

```
Class(mySimplex)
mySimplex$soln      # Soluciones
mySimplex$soln[2]   # Solución de la segunda variable
mySimplex$value     # Valor de la función objetivo
mySimplex$slack     # Holgura de cada restricción <=
mySimplex$surplus   # Excedente de cada restricción ≥
```

Este objeto tiene, además, otros atributos relativos al cálculo:

```
mySimplex$a         # Coeficientes de Z con variables básicas
con coeficientes igual a cero

mySimplex$basic     # Los índices de las variables básicas en
la solución encontrada

mySimplex$A         # Matriz de restricciones expresada en
términos de variables no básicas
```

Esta función es de las más básicas para trabajar con problemas de optimización. No trae incorporado análisis de sensibilidad (aunque pueden calcularse), ni permite declarar variables enteras. Además presupone no negatividad, por lo cual si queremos que nuestras variable puedan tomar negativas, tenemos que utilizar el artilugio de separar nuestra variable en dos[37]. Sin embargo, es muy intuitiva en su operación con matrices. La contra más grande que tiene es que es lenta, tarda un tiempo promedio proporcional al cubo de la cantidad de restricciones para hallar una respuesta.

6.3. Otros paquetes de PL: lpSolve

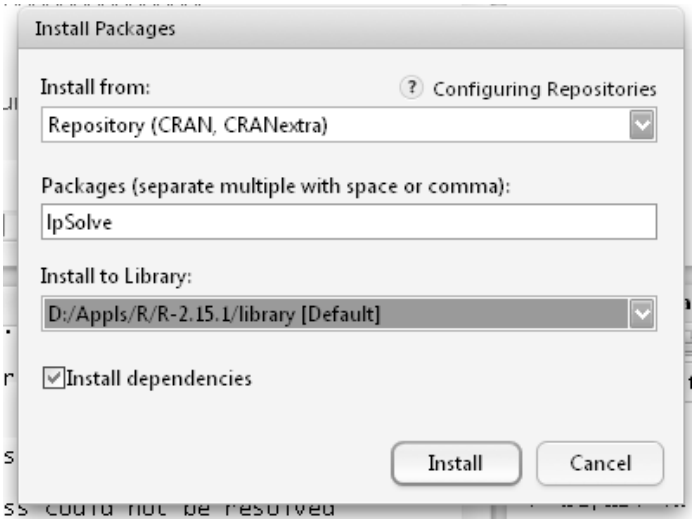
Vamos a trabajar con otro paquete para resolver problemas de optimización lineal. Como no es estándar, tenemos que descargarlo desde CRAN:

```
install.packages("lpSolve")
```

Ahora, seleccionamos la ubicación de algún servidor y listo, ya queda instalado.

En RStudio podría haber ido al menú "Tools", seleccionar el ítem "Install Packages" y escribir "lpSolve" en el cuadro de texto:

[37] O sea, si Xa puede tomar valores negativos, tenemos que utilizar dos variables, Xa1 y Xa2, relacionándolas mediante Xa = Xa1 – Xa2.



Una vez instalado, tenemos que cargarlo al workspace[38] actual con la función "library":

```
library(lpSolve)
```

lpSolve encapsula el acceso al programa también llamado lpSolve[39]. El mismo tiene licencias LGPL y está disponible en forma de bibliotecas. El paquete lpSolve implementa algunas de las funcionalidades del programa lpSolve.

¿Se acuerdan cómo funcionaba la función "simplex" del paquete "boot"?. Teníamos tres matrices de coeficientes de restricciones y tres vectores de términos independientes, uno por cada tipo de restricción (≤, =, ≥). En este paquete las cosas funcionan un poco más simples.
Bueno, resolvamos un problema sencillo con lpSolve.

Primero, definamos los coeficientes de la función objetivo:

```
problem.coef.obj <- c(1, 9, 3)
```

[38] Al espacio de trabajo actual, la sesión que tenemos abierta.

[39] La biblioteca se puede descargar desde: http://lpsolve.sourceforge.net/5.5 para utilizarla en otro lenguaje de programación, pero no es necesaria si solo trabajamos con ella desde R, se instala con el comando *install.packages*

Como vemos, es una función de tres variables. Supongamos que tenemos dos restricciones, una del tipo ≥ y otra del tipo ≤:

```
problem.coef.restr <-  matrix (c(1, 2, 3, 3, 2, 2), nrow=2,
byrow=TRUE)

problem.coef.restr
```

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

En una única matriz defino mis dos restricciones. Como el parámetro "byrow" [40]vale "TRUE", tenemos la primera restricción de coeficientes (1, 2, 3) y la segunda de coeficientes (3,-2,2).

Y ahora defino los signos de mis restricciones en un vector del tipo "character":

```
problem.dir.restr <- c(">=", "<=")

problem.dir.restr
```

```
[1] ">=" "<="
```

Seguimos con los términos independientes:

```
problem.term.restr <- c(1, 15)
```

Maximicemos la función objetivo:

```
lp ("max", problem.coef.obj, problem.coef.restr,
problem.dir.restr, problem.term.restr)
```

```
Success: the objective function is 67.5
```

En cuanto a sintaxis, mucho mejor que usar la función simplex. Además, el algoritmo simplex que implementa la función *lp()* es mucho más veloz.

[40] Recordemos que al crear una matriz, "byrow = TRUE" implica que pasamos los índices de la matriz por filas. En este caso, en una matriz de 2x3, los primeros 3 índices pasados corresponden a la primera fila, y los segundos 3 a la segunda fila.

Como ya sabemos, todo en R es un objeto:

```
mylp <- lp ("max", problem.coef.obj, problem.coef.restr,
problem.dir.restr, problem.term.restr)
mylp
```

```
Success: the objective function is 67.5
```

Y como objeto, podemos acceder a sus miembros. Primero, las soluciones del problema:

```
mylp$solution
```

```
[1] 0.0 7.5 0.0
```

También el valor de Z alcanzado:

```
mylp$objval
```

```
[1] 67.5
```

¿Queremos hacer análisis de sensibilidad?, bueno, hagamos valer TRUE el parámetro "compute.sens":

```
mylp <- lp ("max", problem.coef.obj, problem.coef.restr,
problem.dir.restr, problem.term.restr, compute.sens=TRUE)
```

Y accedamos a los atributos "sens.coef.from" y "sens.coef.from" para ver la sensibilidad de los coeficientes de la función objetivo:

```
mylp$sens.coef.from
```

```
[1] -1e+30  3e+00 -1e+30
```

```
mylp$sens.coef.to
```

```
[1] 1.35e+01 1.00e+30 9.00e+00
```

"compute.sens=TRUE" nos permite hacer análisis de dualidad también:

```
mylp$duals
```

```
[1]   0.0   4.5 -12.5   0.0  -6.0
```

Donde sí tengo “n” variables y “m” restricciones, los primeros m valores corresponden a los valores duales de las restricciones y los siguientes n valores a los duales de las actuales variables de decisión. Puedo ver la sensibilidad de los duales también:

```
mylp$duals.from
```

```
[1] -1.0e+30  1.0e+00 -1.0e+30 -1.0e+30 -1.4e+01
```

```
mylp$duals.to
```

```
[1] 1.0e+30 1.0e+30 5.0e+00 1.0e+30 7.5e+00
```

No me devuelve las holguras, pero las puedo calcular a mano. Primero devuelvo la matriz de las restricciones:

```
mylp$constraints
```

```
              [,1] [,2]
                 1    3
                 2    2
                 3    2
const.dir.num    2    1
const.rhs        1   15
```

La matriz se devuelve transpuesta (es la forma en que lpSolve trabaja internamente). La primera columna es la primera restricción, la segunda es la segunda restricción. La última fila representa a los términos independientes, la anteúltima son referencias al sentido de las restricciones (2 es ≥, 1 es ≤ y 3 =) y los primeros valores los coeficientes de las restricciones. Entonces, podemos hacer:

```
sum(mylp$constraints[1:3,1] * mylp$solution) -
mylp$constraints[5,1]
```

```
const.rhs
       14
```

```
sum(mylp$constraints[1:3,2] * mylp$solution) -
mylp$constraints[5,2]
```

```
const.rhs
        0
```

Todo muy lindo, pero con un poco de ingenio (y mucho trabajo manual) podíamos llegar a tener un análisis de sensibilidad con la función *simplex()* del paquete boot. ¿Qué más podemos hacer con este paquete?. Bueno, supongamos que de las tres variables queremos hacer que la segunda y la tercera sean enteras:

```
mylp <- lp ("max", problem.coef.obj, problem.coef.restr,
problem.dir.restr, problem.term.restr, compute.sens=TRUE,
int.vec=c(2:3))
```

```
mylp$solution
```

```
[1] 0.3333333 7.0000000 0.0000000
```

Con el parámetro "int.vec" puedo pasarle a lp un vector con los índices de las variables enteras. Si quiero ver cuáles eran las variables enteras:

```
mylp$int.count #Me devuelve la cantidad de variables enteras
```

```
[1] 2
```

```
mylp$int.vec # Devuelve los índices de las variables enteras
```

```
[1] 2 3
```

```
mylp$solution[mylp$int.vec] # Devuelvo los valores de las
variables enteras
```

```
[1] 7 0
```

Pero bueno, quizás quiero que la primera variable sea binaria, entonces uso el parámetro “binary.vec”:

```
mylp <- lp ("max", problem.coef.obj, problem.coef.restr,
problem.dir.restr, problem.term.restr, compute.sens=TRUE,
int.vec=c(2:3), binary.vec=c(1))
```

```
mylp
```

```
mylp$solution
```

```
mylp$bin.count
```

```
mylp$binary.vec
```

```
mylp$solution[mylp$binary.vec]
```

Si tengo todas mis variables enteras, puedo reemplazar int.vec por "all.int=TRUE":

```
mylp <- lp ("max", problem.coef.obj, problem.coef.restr,
problem.dir.restr, problem.term.restr, compute.sens=TRUE,
all.int=TRUE)
```

```
mylp
```

```
mylp$solution
```

```
mylp$int.count
```

```
mylp$int.vec
```

```
mylp$solution[mylp$int.vec]
```

Y si son todas binarias, uso "all.bin=TRUE":

```
mylp <- lp ("max", problem.coef.obj, problem.coef.restr,
problem.dir.restr, problem.term.restr, compute.sens=TRUE,
all.bin=TRUE)

mylp

mylp$solution

mylp$bin.count

mylp$binary.vec

mylp$solution[mylp$binary.vec]
```

Noten que en ningún lado pusimos ninguna restricción de no negatividad. Y es que la envoltura para R de lpSolve asume que todas las variables son no negativas. Si queremos tener una variable libre, es necesario hacer la siguiente transformación: $X_n = X_{nA} - X_{nB}$, donde X_{nA} y X_{nB} son no negativas. Supongamos ahora que nuestra segunda variable puede tomar valores negativos con las variables pertenecientes al dominio real. Primero, aplico la transformación $X_n = X_{nA} - X_{nB}$ a la segunda variable en la función objetivo, duplicando el segundo coeficiente, y cambiándole el signo a la segunda duplicación:

```
problem.coef.obj <- c(1, 9, -9, 3)
```

Donde al coeficiente 9 lo separé en 9 y -9 ya que:
$9X2 = 9(X2_A - X2_B) = 9X2_A - 9X2_B$.
Aplico el mismo principio en los coeficientes de las restricciones:

```
problem.coef.restr <-  matrix (c(1, 2, -2, 3, 3, 2, -2, 2),
nrow=2, byrow=TRUE)

problem.coef.restr
```

```
     [,1] [,2] [,3] [,4]
[1,]    1    2   -2    3
[2,]    3    2   -2    2
```

Y a los vectores de sentidos de las restricciones y términos independientes no los modifico, ya que siguen igual. Ejecuto la función *lp()*:

```
mylp <- lp ("max", problem.coef.obj, problem.coef.restr,
problem.dir.restr, problem.term.restr, compute.sens=TRUE)

mylp

mylp$solution
```

```
[1] 0.0 7.5 0.0 0.0
```

El paquete lpSolve tiene algunas funcionalidades adicionales. Viene con dos funciones que implementan los algoritmos de asignación y transporte. Veamos el caso de transporte. Definamos la matriz de costo:

```
matrizCosto <- matrix (10000, 8, 5)
matrizCosto
```

```
     [,1]  [,2]  [,3]  [,4]  [,5]
[1,] 10000 10000 10000 10000 10000
[2,] 10000 10000 10000 10000 10000
[3,] 10000 10000 10000 10000 10000
[4,] 10000 10000 10000 10000 10000
[5,] 10000 10000 10000 10000 10000
[6,] 10000 10000 10000 10000 10000
[7,] 10000 10000 10000 10000 10000
[8,] 10000 10000 10000 10000 10000
```

En esta matriz cada elemento [i,j] nos dice cual es costo unitario de transporte desde i a j. La modificaremos un poco para hacerla más interesante:[41]

```
matrizCosto[4,1]<- matrizCosto[-4,5]<- 0
matrizCosto
matrizCosto[1,2]<- matrizCosto[2,3]<- matrizCosto[3,4] <- 7
matrizCosto[1,3]<- matrizCosto[2,4]<- 7.7
matrizCosto
matrizCosto[5,1]<- matrizCosto[7,3]<- 8
matrizCosto
```

```
     [,1]  [,2]    [,3]    [,4]  [,5]
[1,] 10000     7     7.7 10000.0     0
[2,] 10000 10000     7.0     7.7     0
[3,] 10000 10000 10000.0     7.0     0
[4,]     0 10000 10000.0 10000.0 10000
[5,]     8 10000 10000.0 10000.0     0
[6,] 10000 10000 10000.0 10000.0     0
[7,] 10000 10000     8.0 10000.0     0
[8,] 10000 10000 10000.0 10000.0     0
```

Sigamos modificando:

[41] Observen el detalle que realizamos múltiples asignaciones en una línea. Si todas las asignaciones son del mismo valor, podemos hacer esto.

```
matrizCosto[1,4] <- 8.4

matrizCosto[6,2] <- 9

matrizCosto[8,4] <- 10

matrizCosto[4,2:4] <- c(.7, 1.4, 2.1)

matrizCosto
```

```
      [,1]  [,2]    [,3]    [,4]  [,5]
[1,] 10000 7e+00     7.7     8.4     0
[2,] 10000 1e+04     7.0     7.7     0
[3,] 10000 1e+04 10000.0     7.0     0
[4,]     0 7e-01     1.4     2.1 10000
[5,]     8 1e+04 10000.0 10000.0     0
[6,] 10000 9e+00 10000.0 10000.0     0
[7,] 10000 1e+04     8.0 10000.0     0
[8,] 10000 1e+04 10000.0    10.0     0
```

Armemos ahora los vectores de sentido de las restricciones y el de los términos independientes, tanto para columnas como filas:

```
row.dir <- rep ("<", 8)

row.term <- c(200, 300, 350, 200, 100, 50, 100, 150)

col.dir <- rep (">", 5)

col.term <- c(250, 100, 400, 500, 200)
```

Corramos ahora *lp.transport()*:

```
myTransport <- lp.transport(matrizCosto, "min", row.dir,
row.term, col.dir, col.term)

myTransport
```

```
Success: the objective function is 7790
```

```
myTransport$solution
```

```
     [,1] [,2] [,3] [,4] [,5]
[1,]    0  100  100    0    0
[2,]    0    0  200  100    0
[3,]    0    0    0  350    0
[4,]  200    0    0    0    0
[5,]   50    0    0    0   50
[6,]    0    0    0    0   50
[7,]    0    0  100    0    0
[8,]    0    0    0   50  100
```

Muy “inteligentemente”, R nos devuelve una matriz de asignación de orígenes a destinos, en vez de una lista de valores de variables de decisión.

Definir un problema de asignación es más fácil, ya que al ser todas las restricciones "≤1" con variables binarias, lpSolve me automatiza la carga, por lo cual solo me ocupo de la matriz de costo:

```
matrizCostoAsignacion <- matrix (c(2, 7, 7, 2, 7, 7, 3, 2,
7, 2, 8, 10, 1, 9, 8, 2), 4, 4)
```

```
matrizCostoAsignacion
```

```
     [,1] [,2] [,3] [,4]
[1,]    2    7    7    1
[2,]    7    7    2    9
[3,]    7    3    8    8
[4,]    2    2   10    2
```

```
myAssign <- lp.assign(matrizCostoAsignacion, "min")
```

```
myAssign
```

```
Success: the objective function is 8
```

```
myAssign$solution
```

```
     [,1] [,2] [,3] [,4]
[1,]    0    0    0    1
[2,]    0    0    1    0
[3,]    0    1    0    0
[4,]    1    0    0    0
```

6.4. Otros paquetes de PL: lpSolveAPI

lpSolve no es la única implementación de “lpSolve” para R, tenemos también lpSolveAPI:

```
install.packages("lpSolveAPI")
```

```
library(lpSolveAPI)
```

Los modelos con lpSolveAPI se construyen de otra forma. Para empezar, no son objetos R hechos y derechos, así que es muy parecido su uso a como se usaría desde C.

Con lpSolveAPI comienzo así:

```
lprec <- make.lp(0, 4)
```

```
lprec
```

```
Model name:
```

```
            C1    C2    C3    C4
Minimize     0     0     0     0
Kind       Std   Std   Std   Std
Type      Real  Real  Real  Real
Upper      Inf   Inf   Inf   Inf
Lower        0     0     0     0
```

Con esto creo un objeto "modelo lineal" con 4 variables de decisión. Definamos ahora los coeficientes:

```
set.objfn(lprec, c(1, 3, 6.24, 0.1))
lprec
```

```
Model name:
            C1    C2    C3    C4
Minimize     1     3  6.24   0.1
Kind       Std   Std   Std   Std
Type      Real  Real  Real  Real
Upper      Inf   Inf   Inf   Inf
Lower        0     0     0     0
```

Agrego las restricciones:

```
add.constraint(lprec, c(0, 78.26, 0, 2.9), ">=", 92.3)
add.constraint(lprec, c(0.24, 0, 11.31, 0), "<=", 14.8)
add.constraint(lprec, c(12.68, 0, 0.08, 0.9), ">=", 4)
lprec
```

```
Model name:
             C1     C2     C3     C4
Minimize      1      3   6.24    0.1
R1            0  78.26      0    2.9  >=  92.3
R2         0.24      0  11.31      0  <=  14.8
R3        12.68      0   0.08    0.9  >=     4
Kind        Std    Std    Std    Std
Type       Real   Real   Real   Real
Upper       Inf    Inf    Inf    Inf
Lower         0      0      0      0
```

En caso de restricciones cuya única función sea acotar una única variable, las agrego separadamente como "Bounds":

```
set.bounds(lprec, lower = c(28.6, 18), columns = c(1, 4))
set.bounds(lprec, upper = 48.98, columns = 4)
lprec
```

```
Model name:
             C1     C2     C3     C4
Minimize      1      3   6.24    0.1
R1            0  78.26      0    2.9  >=  92.3
R2         0.24      0  11.31      0  <=  14.8
R3        12.68      0   0.08    0.9  >=     4
```

```
Kind        Std    Std    Std    Std
Type       Real   Real   Real   Real
Upper       Inf    Inf    Inf  48.98
Lower      28.6      0      0     18
```

Puedo ponerle nombre:

```
rowNames <- c("Rest. 01", "Rest. 02", "Rest. 03")
colNames <- c("X1", "X2", "X3", "X4")
dimnames(lprec) <- list(rowNames, colNames)
lprec
```

```
Model name:
             X1     X2     X3     X4
Minimize      1      3   6.24    0.1
Rest. 01      0  78.26      0    2.9  >=  92.3
Rest. 02   0.24      0  11.31      0  <=  14.8
Rest. 03  12.68      0   0.08    0.9  >=     4
Kind        Std    Std    Std    Std
Type       Real   Real   Real   Real
Upper       Inf    Inf    Inf  48.98
Lower      28.6      0      0     18
```

Listo, ahora puedo resolverlo:

```
solve(lprec)
```

```
[1] 0
```

Si quiero ver el valor de Z:

```
get.objective(lprec)
```

```
[1] 31.78276
```

Las soluciones:

```
get.variables(lprec)
```

```
[1] 28.60000  0.00000  0.00000 31.82759
```

Los valores de las restricciones:

```
get.constraints(lprec)
```

```
[1]  92.3000   6.8640 391.2928
```

Si quiero que mi cuarta variable tome valores enteros:

```
set.type(lprec, 4, "integer")
lprec
```

```
Model name:
            X1     X2     X3     X4
Minimize     1      3   6.24    0.1
Rest. 01     0  78.26      0    2.9  >=  92.3
Rest. 02  0.24      0  11.31      0  <=  14.8
Rest. 03 12.68      0   0.08    0.9  >=     4
Kind        Std    Std    Std    Std
Type       Real   Real   Real    Int
Upper       Inf    Inf    Inf  48.98
Lower      28.6      0      0     18
```

```
solve(lprec)
get.objective(lprec)
```

```
[1] 31.792
```

```
get.variables(lprec)
```

```
[1] 28.60000000  0.03066701  0.00000000 31.00000000
```

```
get.constraints(lprec)
```

```
[1]  92.300   6.864 390.548
```

Si la segunda variable debiera ser binaria[42]:

```
set.type(lprec, 2, "binary")
lprec
```

```
Model name:
            X1     X2     X3     X4
Minimize     1      3   6.24    0.1
Rest. 01     0  78.26      0    2.9  >=  92.3
Rest. 02  0.24      0  11.31      0  <=  14.8
Rest. 03 12.68      0   0.08    0.9  >=     4
Kind        Std    Std    Std    Std
Type       Real    Int   Real    Int
Upper       Inf      1    Inf  48.98
Lower      28.6      0      0     18
```

```
solve(lprec)
get.objective(lprec)
```

[42] Observen como la declara entera y acotada por arriba en 1 y por abajo en 0.

```
[1] 31.8
```

```
get.variables(lprec)
```

```
[1] 28.6  0.0  0.0 32.0
```

Lo bueno del paquete lpSolveAPI es que permite leer y grabar archivos estándares de programación lineal en formato "mps":

```
write.lp(lprec, "..\\data\\lpModel01.mps", type = "mps")
```

Vaciemos la memora de trabajo:

```
rm(list=ls())
```

Y carguemos nuestro modelo:

```
lprec <- read.lp("..\\data\\lpModel01.mps", type = "mps")
lprec
```

6.5. Optimizando en base a muestras

Pero pese a todo, la potencia de R no radica en poder ejecutar un simplex, sino en cómo lo puede complementar. En un problema real, nuestro trabajo no comienza con una definición del problema en forma de ecuaciones, sino con un conjunto de datos de relevamientos que hay que procesar en forma estadística. Resolvamos por ejemplo, un problema de mezcla apelando a la función *simplex()*.
Supongamos que tenemos todos nuestros relevamientos en un archivo en formato delimitado por comas denominado "problemaMezclaMuestras.csv"[43]:

Con *read.csv()* lo leo y lo guardo en un data frame:

```
setwd("D://Proyectos//IOR//scripts")
```

[43] Lo puedo descargar de http://www.modelizandosistemas.com.ar/p/optimizacion-con-r.html, y debo grabarlo en el directorio "data" dentro de mi directorio de proyectos.

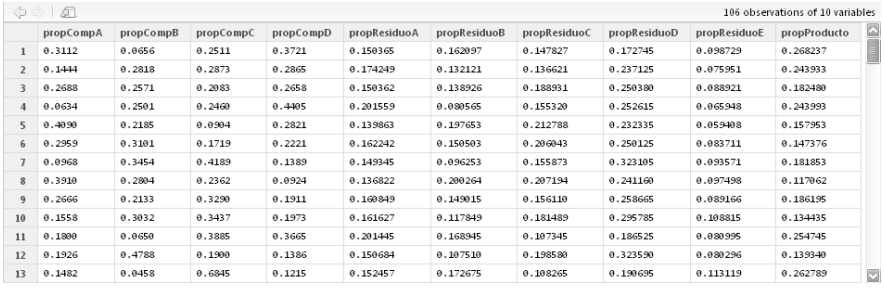
```
misMuestras <- read.csv("..//data//problemaMezcla-
Muestras.csv")
```

Con *View()* y *str()* puedo analizarlo:

```
str(misMuestras)
```

```
'data.frame':      106 obs. of  10 variables:
 $ propCompA    : num  0.3112 0.1444 0.2688 0.0634 0.409 ...
 $ propCompB    : num  0.0656 0.2818 0.2571 0.2501 0.2185 ...
 $ propCompC    : num  0.2511 0.2873 0.2083 0.246 0.0904 ...
 $ propCompD    : num  0.372 0.286 0.266 0.44 0.282 ...
 $ propResiduoA: num  0.15 0.174 0.15 0.202 0.14 ...
 $ propResiduoB: num  0.1621 0.1321 0.1389 0.0806 0.1977 ...
 $ propResiduoC: num  0.148 0.137 0.189 0.155 0.213 ...
 $ propResiduoD: num  0.173 0.237 0.25 0.253 0.232 ...
 $ propResiduoE: num  0.0987 0.076 0.0889 0.0659 0.0594 ...
 $ propProducto: num  0.268 0.244 0.182 0.244 0.158 ...
```

```
View(misMuestras)
```



106 observations of 10 variables

	propCompA	propCompB	propCompC	propCompD	propResiduoA	propResiduoB	propResiduoC	propResiduoD	propResiduoE	propProducto
1	0.3112	0.0656	0.2511	0.3721	0.150365	0.162097	0.147827	0.172745	0.098729	0.268237
2	0.1444	0.2818	0.2873	0.2865	0.174249	0.132121	0.136621	0.237125	0.075951	0.243933
3	0.2688	0.2571	0.2083	0.2658	0.150362	0.138926	0.188931	0.250380	0.088921	0.182480
4	0.0634	0.2501	0.2460	0.4405	0.201559	0.080565	0.155320	0.252615	0.065948	0.243993
5	0.4090	0.2185	0.0904	0.2821	0.139863	0.197653	0.212788	0.232335	0.059408	0.157953
6	0.2959	0.3101	0.1719	0.2221	0.162242	0.150503	0.206043	0.250125	0.083711	0.147376
7	0.0968	0.3454	0.4189	0.1389	0.149345	0.096253	0.155873	0.323105	0.093571	0.181853
8	0.3910	0.2804	0.2362	0.0924	0.136822	0.200264	0.207194	0.241160	0.097498	0.117062
9	0.2666	0.2133	0.3290	0.1911	0.160849	0.149015	0.156110	0.258665	0.089166	0.186195
10	0.1558	0.3032	0.3437	0.1973	0.161627	0.117849	0.181489	0.295785	0.108815	0.134435
11	0.1800	0.0650	0.3885	0.3665	0.201445	0.168945	0.107345	0.186525	0.080995	0.254745
12	0.1926	0.4788	0.1900	0.1386	0.150684	0.107510	0.198580	0.323590	0.080296	0.139340
13	0.1482	0.0458	0.6845	0.1215	0.152457	0.172675	0.108265	0.190695	0.113119	0.262789

Y tomemos ahora el archivo “problemaMezcla-Restricciones.csv" [44]en el cual tengo tabuladas mis restricciones.

```
misRestricciones <- read.csv("..//data//problemaMezcla-
Restricciones.csv")
```

```
str(misRestricciones)
```

```
'data.frame':      1 obs. of  5 variables:
 $ cantMaxResA: num 0.25
 $ cantMaxResB: num 0.3
 $ cantMaxResC: num 0.3
 $ cantMaxResD: num 0.1
 $ cantMaxResE: num 0.15
```

```
View(misRestricciones)
```

[44] Lo puedo descargar de http://www.modelizandosistemas.com.ar/p/optimizacion-con-r.html, y debo grabarlo en el directorio “data” dentro de mi directorio de proyectos.

1 observations of 5 variables

	cantMaxResA	cantMaxResB	cantMaxResC	cantMaxResD	cantMaxResE
1	0.25	0.3	0.3	0.1	0.15

Supongamos que tengo las muestras anteriores, las que indican que proporción de cada componente se agregó en la mezcla, y que proporción de productos y residuos se obtuvo:

```
componentes <- c(seq(1:4))
residuos <- c(seq(1:5) + 4)
producto <- 10
View(misMuestras[componentes])
View(misMuestras[residuos])
View(misMuestras[producto])
```

Chequeo que las proporciones estén normalizadas (o sea, sumen 1):

```
summary(misMuestras[1]+misMuestras[2]+misMuestras[3]+misMues
tras[4])
```

```
  propCompA
Min.   :1
1st Qu.:1
Median :1
Mean   :1
3rd Qu.:1
Max.   :1
```

```
summary(misMuestras[5]+misMuestras[6]+misMuestras[7]+misMues
tras[8]+misMuestras[9]+misMuestras[10])
```

```
 propResiduoA
Min.   :1
1st Qu.:1
Median :1
Mean   :1
3rd Qu.:1
Max.   :1
```

¿Cómo meto esto en un simplex? Bueno, en principio, no puedo. Pero lo que puedo hacer es estimar la relación entre cada componente con cada residuo y con el producto:

```
lm.propResiduoA <- lm (propResiduoA ~ 0 + propCompA +
propCompB + propCompC + propCompD, data = misMuestras)
lm.propResiduoB <- lm (propResiduoB ~ 0 + propCompA +
propCompB + propCompC + propCompD, data = misMuestras)
```

```
lm.propResiduoC <- lm (propResiduoC ~ 0 + propCompA +
propCompB + propCompC + propCompD, data = misMuestras)

lm.propResiduoD <- lm (propResiduoD ~ 0 + propCompA +
propCompB + propCompC + propCompD, data = misMuestras)

lm.propResiduoE <- lm (propResiduoE ~ 0 + propCompA +
propCompB + propCompC + propCompD, data = misMuestras)

lm.propProducto <- lm (propProducto ~ 0 + propCompA +
propCompB + propCompC + propCompD, data = misMuestras)
```

Con los ceros al inicio de la lista de variables independientes me aseguro de no incluir el término independiente. Veamos los ajustes que realizó lm:

```
summary(lm.propResiduoA)

summary(lm.propResiduoB)

summary(lm.propResiduoC)

summary(lm.propResiduoD)

summary(lm.propResiduoE)

summary(lm.propProducto)
```

Armemos nuestro vector de coeficientes de la función objetivo:

```
coefFuncionObjetivo <- lm.propProducto$coefficients[1:4]

coefFuncionObjetivo
```

```
  propCompA   propCompB   propCompC   propCompD
 0.14700694 -0.02702092  0.26196634  0.39491503
```

Las restricciones son todas del tipo ≤, así que tenemos que armar A1.

```
aux_coeficientesA1 <-
c(lm.propResiduoA$coefficients[1:4],lm.propResiduoB$coeffici
ents[1:4] )

aux_coeficientesA1 <-
c(aux_coeficientesA1,lm.propResiduoC$coefficients[1:4],lm.pr
opResiduoD$coefficients[1:4] )

aux_coeficientesA1 <-
c(aux_coeficientesA1,lm.propResiduoE$coefficients[1:4] )

restriccionesA1 <- matrix(aux_coeficientesA1, 5,4,
byrow=TRUE)

restriccionesA1
```

```
           [,1]       [,2]       [,3]       [,4]
[1,] 0.06012122 0.14674360 0.15009861 0.27783015
[2,] 0.33830113 0.02659127 0.14828142 0.07405588
[3,] 0.27897496 0.28152555 0.06114977 0.07323397
[4,] 0.11694078 0.53158161 0.23098814 0.09057579
[5,] 0.05865496 0.04057890 0.14751572 0.08938917
```

Armemos los términos independientes (b1).

```
termIndepb1 <- as.numeric(c(misRestricciones[1,]))

termIndepb1
```

```
[1] 0.25 0.30 0.30 0.10 0.15
```

Ojo, tengo una restricción del tipo =, para asegurarme que las proporciones de los componentes sumen 1. Esto lo cargo en la matríz A3:

```
restriccionesA3 <- matrix(c(1,1,1,1),1,4,byrow=TRUE)

termIndepb3 <- 1
```

Y ahora ejecuto *simplex()*:

```
simplexProblemaMezcla <- simplex(coefFuncionObjetivo,
A1=restriccionesA1, b1=termIndepb1, A3=restriccionesA3,
b3=termIndepb3, maxi=TRUE )

simplexProblemaMezcla
```

```
Linear Programming Results

Call : simplex(a = coefFuncionObjetivo, A1 = restriccionesA1, b1 =
termIndepb1, A3 = restriccionesA3, b3 = termIndepb3, maxi = TRUE)

Maximization Problem with Objective Function Coefficients
         x1          x2          x3          x4
 0.14700694 -0.02702092  0.26196634  0.39491503


Optimal solution has the following values
        x1         x2         x3         x4
0.09940407 0.00000000 0.04845317 0.85214276
The optimal value of the objective  function is 0.363830172772084.

simplexProblemaMezcla$slack


[1] 0.00000000 0.19608060 0.20690005 0.00000000 0.06084952
```

Algoritmos para programación no lineal

7.1. La función OPTIM

Existe una función llamada *optim()*, dentro del paquete stats (el cual se carga por defecto y no es necesario instalarlo), que sirve para optimizar funciones lineales y no lineales, a la cual hay que pasarle como parámetro la función a optimizar de esta forma:

```
myCuadratica <- function(x){
  x^2
}
optim(par=100, fn=myCuadratica)
```

```
$par
[1] 0

$value
[1] 0

$counts
function gradient
      32       NA

$convergence
[1] 0

$message
NULL

Mensajes de aviso perdidos
In optim(par = 100, fn = myCuadratica) :
  one-diml optimization by Nelder-Mead is unreliable:
use "Brent" or optimize() directly
```

optim() es una función que permite minimizar funciones no lineales, por varios métodos. Los dos parámetros principales son "fn", el cual es la función a optimizar, y "par", que representa al valor inicial desde el cual queremos comenzar la búsqueda. Son los únicos dos parámetros obligatorios, y si los pongo en orden puedo prescindir de "nombre=":

```
optim(100, myCuadratica)
```

```
$par
[1] 0

$value
```

```
[1] 0

$counts
function gradient
      32       NA

$convergence
[1] 0

$message
NULL

Mensajes de aviso perdidos
In optim(100, myCuadratica) :
  one-diml optimization by Nelder-Mead is unreliable:
use "Brent" or optimize() directly
```

Esta función, al buscar mínimos en funciones objetivos de cualquier tipo, no puede inferir a priori la estructura de la función a optimizar, por lo cual acepta como parámetros funciones cuya única variable sea un vector. O sea, permite camuflar cualquier número de variables independientes en un vector de variables independiente. En castellano, si queremos minimizar una función de dos variables, no podemos declararla como function(x,y), sino que debemos declararla como function(x) donde "x" es un vector de dos componentes.

optim() aproxima el mínimo de la función que le pasemos como parámetro, pero funciona mediante métodos de búsqueda y, como tal, puede quedar trabada en un óptimo local. Nos pide que le pasemos "par" [45]para tener una idea desde donde arrancar a buscar. En este caso, "par" podría ser cualquier valor:

```
optim(25, myCuadratica)


$par
[1] 0

$value
[1] 0

$counts
function gradient
      32       NA

$convergence
[1] 0

$message
NULL

Mensajes de aviso perdidos
In optim(25, myCuadratica) :
  one-diml optimization by Nelder-Mead is unreliable:
```

[45] Cuidado, no confundir con la **función** *par()* utilizada para configurar gráficos.

```
use "Brent" or optimize() directly
```

```
optim(500000, myCuadratica)
```

```
$par
[1] 0

$value
[1] 0

$counts
function gradient
      32       NA

$convergence
[1] 0

$message
NULL

Mensajes de aviso perdidos
In optim(5e+05, myCuadratica) :
  one-diml optimization by Nelder-Mead is unreliable:
use "Brent" or optimize() directly
```

```
optim(-500000, myCuadratica)
```

```
$par
[1] 50000

$value
[1] 2.5e+09

$counts
function gradient
      10       NA

$convergence
[1] 0

$message
NULL

Mensajes de aviso perdidos
In optim(-5e+05, myCuadratica) :
  one-diml optimization by Nelder-Mead is unreliable:
use "Brent" or optimize() directly
```

¿Qué pasó?, funcionó bien para 25 y 50.000, pero no para -50.000. Lo que pasó es que el método por defecto de búsqueda de *optim()* no está preparado para búsquedas unidimensionales (fíjense que lo último que devuelve optim es un aviso acerca de la no confiabilidad del algoritmo "Nelder-Mead" para búsquedas unidimensionales, y nos sugiere usar el método "Brent"). Hagámosle caso:

```
optim(-500000, myCuadratica, method="Brent")
```

```
Error en optim(-5e+05, myCuadratica, method = "Brent") :
  'lower' and 'upper' must be finite values
```

¿Y ahora qué?. Bueno, "Brent" exige que le acotemos el espacio de búsqueda:

```
optim(-500000, myCuadratica, method="Brent", lower=-510000,
upper=510000)

$par
[1] 0

$value
[1] 0

$counts
function gradient
      NA       NA

$convergence
[1] 0

$message
NULL
```

Ahora sí, probemos con diferentes valores iniciales y cotas:

```
optim(-23.8, myCuadratica, method="Brent", lower=-100,
upper=100)$value

optim(-230000000.8, myCuadratica, method="Brent", lower=-
100000, upper=100000)$value

optim(-23.8, myCuadratica, method="Brent", lower=-1000000,
upper=100)$value

optim(-11111111, myCuadratica, method="Brent", lower=-
1000000, upper=10000000)$value
```

Vemos que aproxima bien la el mínimo de esta función cuadrática. Tiene un error de precisión mínimo cuando las cotas del espacio de búsqueda son muy grandes en valor absoluto, a efectos prácticos es bastante insignificante. También vemos que, como todo objeto, podemos acceder a sus miembros. Veamos un poco su estructura:

```
optimizacion <- optim(-23.8, myCuadratica, method="Brent",
                      lower=-100000,upper=100000)

str(optimizacion)

optimizacion$par # Valores de las variables de decisión
```

```
optimizacion$value # Valor de la función objetivo
optimizacion$convergence # Vale 0 si la solución converge
optimizacion$message # Nos devuelve los mensajes de alerta
```

Si lo pensamos un poco, seguimos con el mismo problema que vimos cuando explicamos el uso de funciones, no podemos pasarle parámetros adicionales a nuestra función. Definamos una cuadrática en su forma general:

```
myCuadraticaGeneral <- function(x, coefa, coefb, coefc){
  coefa*(x^2) + coefb*x + coefc
}
```

Supongamos que quiero minimizar la cuadrática con coeficientes 0.5, -25, -10, respectivamente. Podría crear una nueva función proxy, que solo tome como parámetro la variable de decisión, y encapsule la asignación de los coeficientes de la función cuadrática:

```
myCuadraticaParticular <- function(x){
  myCuadraticaGeneral(x, 0.5, -25, -10)
}
optim(0, myCuadraticaParticular)
```

Pero si tengo que optimizar muchas cosas voy a tener un montón de funciones cargadas en la memoria. ¿Por qué no usar funciones anónimas?:

```
optim(0, (function(x){myCuadraticaGeneral(x,0.5,-25,-10)}))
```

¡¡¡Chan!!!, fíjense lo que hicimos: creamos una versión de la función *myCuadraticaParticular()* al vuelo, declarándola dentro de *optim()*, con toda esta declaración entre paréntesis. Esta función existe solo dentro de la ejecución de *optim()*, después desaparece de memoria. Si se fijan, es la misma declaración que *myCuadráticaParticular()* pero sin asignación a ninguna variable (en realidad la estamos asignando al parámetro "fn" de *optim()*). Lo bueno es que pasamos los parámetros dentro de la misma *optim(),* por lo que no tengo que andar creando nuevas funciones proxys si quiero optimizar cuadráticas con distintos coeficientes..

Bueno, pero optimizar funciones de una sola variable es aburrido. Hagamos algo más complicado:

```
myFuncionNoLineal <- function(x,y){
  x^2 + y^2
}
```

Tenemos ahora una función que toma como entrada un vector de dos componentes, y devuelve como resultado una combinación no lineal de las mismas.
Visualicémosla con *persp()*:

```
?persp
```

persp() necesita como input un vector de puntos para el eje X, uno para el eje Y, y una matriz en el cual el elemento [x,y] sea el valor de la función para (x,y):

```
x <- seq(0:20)-10
y<-x
valoresFuncion <- outer(x,y,myFuncionNoLineal)
persp(x,y,valoresFuncion, col="royalblue", phi=30, theta=30)
```

optim() necesita que la función acepte un solo vector de variables independientes, así que nuestra definición de *myFuncionNoLineal()* con dos parámetros no está en el formato que necesitamos.
Para optimizarlo, utilicemos una función anónima que se encargue de la conversión:

```
optim(c(100,100), (function(x){myFuncionNoLineal(x[1],
x[2])}))
```

El método de resolución para problemas por defecto funciona bien con funciones de dos variables.
Este método es el "Nelder – Mead"[46], el cual optimiza basado solo en los valores de las función objetivo. El método funciona bastante bien pero es lento. Igualmente, al no necesitar gradientes para operar, trabaja muy bien frente a funciones no diferenciables.

[46] Publicado en 1965

Un método basado en gradientes es el "BFGS", el cual es un método del tipo “cuasi-Newton”:

```
optim(c(100,100), (function(x){myFuncionNoLineal(x[1],
x[2])}),method="BFGS")
```

Comparemos el atributo "counts" de optim usando "Nelder - Mead" versus "BFGS":

```
optim(c(100,100), (function(x){myFuncionNoLineal(x[1],
x[2])})
      )$counts
optim(c(100,100), (function(x){myFuncionNoLineal(x[1],
x[2])}),
      method="BFGS")$counts
```

Este atributo tiene dos componentes. El primero (function) nos indica la cantidad de iteraciones del procedimiento, el segundo la cantidad de veces que se debió recalcular el gradiente (en este caso, solo aplica a "BFGS"). A simple vista, "BFGS" es más "rápido", o sea, necesitó menos iteraciones para llegar al objetivo (y tuvo mayor grado de precisión). El método "BFGS" necesita de una función para computar el gradiente. Si no la pasamos, calcula el gradiente por diferencias finitas, pero podemos pasarle la función generadora de gradientes:

```
gradienteMyFuncionNoLineal <- function(x){
  c(2*x[1], 2*x[2])
}
```

Como es una función generadora de gradientes, me tiene que devolver un vector de la misma cantidad de componentes que el vector de variables independientes de la función a optimizar. Entonces:

```
optim(c(100,100), (function(x){myFuncionNoLineal(x[1],
x[2])}),
      gr= gradienteMyFuncionNoLineal, method="BFGS")
```

Y vemos que, al no tener que aproximar por diferencias finitas, optim trabaja muchísimo más rápido (4 iteraciones contra 19 en mi computadora)

optim() acepta muchos más parámetros adicionales, en general pasados como lista en el parámetro "control":

```
optim(c(100,100), (function(x){myFuncionNoLineal(x[1],
x[2])}),
      gr= gradienteMyFuncionNoLineal, method="BFGS",
      control=list(trace=1))
```

"control=list(trace=1)" nos devuelve los valores iniciales y finales de la función objetivo, por ejemplo.

Hasta ahora minimizamos solamente, ¿cómo hacemos para maximizar?. Definamos una función más complicada:

```
funcionMasComplicada <- function(x,y){
  r <- sqrt(x^2+y^2)
  10 * sin(r)/r
}
```

Hagamos su gráfico:

```
x <- seq(-15, 15, length= 40)
y <- x
matrizGraficacion <- outer(x, y, funcionMasComplicada)
matrizGraficacion[is.na(matrizGraficacion)] <- 1
persp(x, y, matrizGraficacion, theta = 30, phi = 30,
      expand = 0.5, col = "royalblue", ltheta = 120,
      shade = 0.75, ticktype = "detailed")
```

Y maximicémoslas con el argumento de control "fnscale":

```
optim(c(1,10), (function(x){funcionMasComplicada(x[1],
x[2])}),
      control=list(trace=6, fnscale=-1))
```

Con un método basado en búsqueda, esta función es candidata a quedar atrapada en un óptimo local. Con "trace=6" vimos la secuencia de pasos que hizo para encontrar el valor óptimo. Probemos con "BFGS":

```
optim(c(1,10), (function(x){funcionMasComplicada(x[1],
x[2])}),
      method="BFGS", control=list(trace=6, fnscale=-1))
```

¿Y si quisiéramos optimizar una función más genérica, es decir, que admita que le modifiquemos sus parámetros?. Armemos un paraboloide:

```
myParaboloideEliptico <- function(x,y,par_a,par_b,par_corrx,
                                  par_corry, par_termIndep){
  ((x+par_corrx)/par_a)^2 + ((y+par_corry)/par_b)^2 +
par_termIndep
}
```

Para optimizarla podemos usar una función implícita. Supongamos que grabamos los valores de los parámetros en variables:

```
a <- 2
b <- 3
corrx <- 1
corry <- 0
termIndep <- -10
optim(c(1,10), (function(x){myParaboloideEliptico(x[1],
x[2],

a,b,corrx,corry,

termIndep)}),
      method="BFGS",control=list(trace=6))
```

7.2. Optimizando con restricciones

Hasta ahora, entonces, estuvimos trabajando con *optim()* para optimización no lineal. Es una función muy buena, pero trabaja minimizando funciones no restringidas. En la práctica de la IO, en general nos tocan problemas que se caracterizan por un conjunto de restricciones que limitan nuestro rango de soluciones. En el caso que queramos minimizar funciones no lineales sujetas a restricciones lineales, disponemos de la función "*constrOptim()*". En líneas generales, es muy

parecida a "*optim()*" pero se le agregan parámetros relativos a las restricciones. Minimicemos un paraboloide elíptico, sujeto a restricciones del tipo ≥. Para ello, usemos la función que creamos anteriormente:

```
myParaboloideEliptico <- function(x,y,par_a,par_b,par_corrx,
                                  par_corry, par_termIndep){
  ((x+par_corrx)/par_a)^2 + ((y+par_corry)/par_b)^2 +
par_termIndep
}
```

Ahora defino las restricciones:

```
matrizA <- matrix(c(1,0,0,1), nrow=2, byrow=TRUE)
vectorb <- c(5,10)
```

Y optimizo:

```
constrOptim(c(100,100),
function(x){myParaboloideEliptico(x[1],x[2],1,1,0,0,0)},
            grad=NULL, ui=matrizA,ci=vectorb)
```

Al igual de lo que pasaba con *optim()*, debo proporcionarle un valor inicial de búsqueda, que en este caso, además, debe estar dentro de la región factible.

El objeto que devuelve es similar al que devuelve es similar al que devuelve *optim()*, pero con los parámetros adicionales "outer.iteration" y "barrier.value", donde el primero son las llamadas que se hacen a *optim()* y el segundo el valor de la barrera.

7.3. Armando nuestra propia versión de OPTIM

Ahora creemos nuestra propia versión de *optim()* personalizada. ¿Qué personalización le queremos dar?, bueno, a priori, hagamos que si mi variable de decisión es unidimensional, ejecute el método Brent por defecto:

Veamos primero la definición de *optim()*:

```
optim
```

Hagamos ahora nuestro prototipo:

```
myOptim <- function (valIniciales, funcion, gradiente =
NULL, ..., cotaInf = -Inf,
          cotaSup = Inf, listaControl = list(), hessiano =
FALSE) {
}
```

El valor inicial se pasa en el parámetro valInicial, así que miremos cuantas componentes tiene y, mediante la función *if()*, determinemos si usamos Brent o no:

```
myOptim <- function (valIniciales, funcion, gradiente =
NULL, ..., cotaInf = -Inf, cotaSup = Inf, listaControl =
list(), hessiano = FALSE) {


  # Chequeo que método usar
  if (length(valIniciales) == 1){
    metodo <- "Brent"
  }
  else {
    metodo <- "Nelder-Mead"
  }


#Llamo a optim():
  print(paste("El método utilizado fue: ", metodo))
  optim(valIniciales, funcion, gradiente, ..., method =
metodo , lower=cotaInf,
        upper=cotaSup, control=listaControl,
hessian=hessiano)
}
```

Probémoslo:

```
myCuadratica <- function(x){
  x^2 + 3
}
```

```
myOptim(c(10), myCuadratica)
```

¿Qué pasó? Bueno, recordemos que el método Brent requiere que las cotas máximas y mínimas sean finitas. Por lo tanto, modifiquemos nuestra definición:

```
myOptim <- function (valIniciales, funcion, gradiente =
NULL, ..., cotaInf = -Inf, cotaSup = Inf, listaControl =
list(), hessiano = FALSE) {


# Chequeo que método usar
if (length(valIniciales) == 1){
    metodo <- "Brent"
    if (cotaInf == -Inf){
      cotaInf <- -100000000000000
    }
    if (cotaSup == Inf){
      cotaSup <-  100000000000000
    }
  }
  else {
    metodo <- "Nelder-Mead"
  }


# Llamo a optim():
  print(paste("El método utilizado fue: ", metodo))
  optim(valIniciales, funcion, gradiente, ..., method=metodo
, lower = cotaInf,
        upper = cotaSup, control = listaControl, hessian =
hessiano)
}
```

Llamemos a nuestra versión mejorada:

```
myOptim(c(10), myCuadratica)
```

Ojo, esta función sirve en el caso unidimensional cuando los valores posibles están dentro de las cotas por defecto (-100000000000000 y 100000000000000), sino debemos explicitarlas.

Puedo hacer ahora que, en caso de optimización multidimensional, si le paso el gradiente resuelva por BFGS:

```
myOptim <- function (valIniciales, funcion, gradiente =
NULL, ..., cotaInf = -Inf, cotaSup = Inf, listaControl =
list(), hessiano = FALSE) {


  # Chequeo que método usar
  if (length(valIniciales) == 1){
    metodo <- "Brent"
    if (cotaInf == -Inf){
      cotaInf <- -100000000000000
    }
    if (cotaSup == Inf){
      cotaSup <-  100000000000000
    }
  }
  else {
    if (is.null(gradiente)){
      metodo <- "Nelder-Mead"
    }
    else {
      metodo <- "BFGS"
    }
  }


  # Llamo a optim():
  print(paste("El método utilizado fue: ", metodo))
  optim(valIniciales, funcion, gradiente, ..., method=metodo
, lower = cotaInf,
        upper = cotaSup, control = listaControl, hessian =
hessiano)
}
```

Llamemos ahora a *myOptim()* con y sin gradiente:

```
myFuncionNoLineal <- function(x,y){
  x^2 + y^2
}


gradienteMyFuncionNoLineal <- function(x){
  c(2*x[1], 2*x[2])
}


myOptim(c(100,100), (function(x){myFuncionNoLineal(x[1],
x[2])}))


myOptim(c(100,100), (function(x){myFuncionNoLineal(x[1],
x[2])}),  gradienteMyFuncionNoLineal)
```

8.1. Algoritmos genéticos

Hasta ahora, trabajamos con problemas lineales resolviéndolos mediante el método simplex (usando tres funciones diferentes), y con problemas no lineales utilizando tres métodos diferentes con la función *optim()* (Brent, Nelder-Mead y BFGS), con búsqueda por gradientes o búsquedas aleatorias. Vamos a estudiar ahora como implementar en R un conjunto de métodos más robustos englobados dentro de la familia de los algoritmos evolutivos, los algoritmos genéticos.

Como siempre, para empezar a trabajar, definamos nuestro directorio de trabajo y borremos el contenido del workspace:

```
setwd("D://Proyectos//IOR//scripts")
rm(list=ls())
```

El paquete que vamos a utilizar es el “genalg”

```
install.packages("genalg")
library("genalg")
```

Parte del poder de los algoritmos genéticos es que su operación abstrae cualquier detalle de la estructura del problema, reduciéndose todas las variables de decisión a un "cromosoma". [47]La forma estándar de trabajo es mediante cromosomas con "genes" binarios. Así que empecemos con este tipo de problemas.

Supongamos el siguiente problema de minimización:

```
# Max Z = XA * 24 + XB * 20
#
# Sujeto a:
#
#   10 * XA + 7 * XB <= 200
#   1 * XA + 0.5 * XB <= 12
#   1 * XA + 0 * XB <= 15
#   0 * XA + 1 * XB <= 24
#   1 * XA + 1 * XB <= 20
```

[47] Raro pasar de hablar de conceptos como solución factible a conceptos como cromosomas, ¿no?

Con todas las variables no negativas y donde el problema consiste en maximizar la cantidad de vapor generada por tonelada de carbón (X) en una caldera, siendo la primer restricción el presupuesto que se dispone para gastar en carbón, la segunda la cantidad máxima de kg de humo por tonelada de carbón quemado que se puede generar, la segunda y la tercera las capacidades de proceso para los carbones A y B, y la última la capacidad de transporte de carbón.

A fin de utilizar el paquete "genalg", es necesario convertir el problema anterior al formalismo de los algoritmos genéticos. Lo primero es decidir cómo codificamos las variables. A priori, dado que tenemos dos variables, nuestros cromosomas deberían tener dos genes solamente:

CROMOSOMA = GEN_A | GEN_B

El siguiente paso es definir con cuantos bits codificamos cada gen. En general, si bien se pueden aplicar algoritmos genéticos a problemas no acotados, siempre es preferible acotar, aunque sea artificialmente, todas las variables de decisión. En este caso, tenemos dos restricciones que podemos usar como cotas naturales:

$1 * X_A + 0 * X_B <= 15$ - Cota XA
$0 * X_A + 1 * X_B <= 24$ - Cota XB

X_A debe pertenecer al intervalo [0; 16]. Como vamos a codificarla por bits, que solo admiten dos valores posibles (0 y 1), debemos buscar la potencia de dos igual o inmediatamente mayor a 16. 16 = 2 elevado a la 4^{ta} potencia, por lo cual con 4 bits podemos codificar X_A:

0000 = 0
...
1111 = 15

Haciendo el mismo análisis con XB, podemos ver que la potencia de 2 inmediatamente superior a 24 es 32 = 2^5, por lo cual:

00000 = 0
...
11111 = 31

Si lo pensamos un poco, trabajar en binario puede ser un poco tedioso, así que hagamos dos funciones para hacer las conversiones:[48]

```
pasarABaseBinariaDesdeDecimal <- function(numero){
  numero.redondeado <- floor(numero)
  binario <- c()
  while (numero.redondeado>=2){
    binario <- c(binario, numero.redondeado %% 2)
    numero.redondeado <- numero.redondeado %/% 2
  }
  binario <- c(binario, numero.redondeado)
  binario[length(binario):1]
}

pasarABaseDecimalDesdeBinaria <- function(arrayDeBits){
  cantidadBits <- length(arrayDeBits)
  multiplicador <- rep(2, times=cantidadBits)
  multiplicador <- multiplicador^(order(multiplicador)-1)
  multiplicador <- multiplicador[order(multiplicador,
decreasing = TRUE)]
  sum(arrayDeBits * multiplicador)
}

pasarABaseBinariaDesdeDecimal(10)
[1] 1 0 1 0


pasarABaseDecimalDesdeBinaria(c(1,0,1,0))
[1] 10
```

Ya codificada nuestra solución, tenemos que analizar cómo armar nuestra función objetivo. A lo que van a decir, "la función objetivo es X_A * 24 + X_B * 20", lo cual es cierto. Pero los algoritmos genéticos no trabajan con restricciones más allá de las cotas de las variables, por lo cual es necesario incluir las restricciones a la función objetivo en transformándola en una otra función. Una forma bastante sencilla puede ser que la función objetivo sea X_A * 24 + X_B * 20 si no se viola ninguna restricción, y si se violan, ya que es un problema de maximización, reducir el valor objetivo en “M” (valor absoluto del desvío), siendo “M” un número muy grande. Entonces, podemos escribir nuestra función objetivo como:

```
funobj <- function(XA, XB){
  # Defino el valor de M
  M <- 50
```

[48] Como ya sabemos, mucho mejor si las guardamos en un script para poder llamarlas cuando necesitemos.

```
  # Chequeo las restricciones:
  desvio <- abs(min(0,200-(10 * XA + 7 * XB)))
  desvio <- desvio + abs(min(0,12-(1 * XA + 0.5 * XB)))
  desvio <- desvio + abs(min(0,16-(1 * XA + 0 * XB)))  # No
necesaria, ya está acotada
  desvio <- desvio + abs(min(0,24-(0 * XA + 1 * XB)))  # Es
necesaria, la cota calculada es 32
  desvio <- desvio + abs(min(0,20-(1 * XA + 1 * XB)))

  XA * 24 + XB * 20 - M * desvio
}
```

Pero recordemos que tenemos que trabajar con cadenas de bits, así que hagamos una función objetivo que acepte una solución potencial al estilo de los algoritmos genéticos, y que internamente llame a nuestra función de evaluación:

```
funobjbin <- function(cromosoma){
  cromosoma.XA <- cromosoma[1:4]
  cromosoma.XB <- cromosoma[5:9]

  XA <- pasarABaseDecimalDesdeBinaria(cromosoma.XA)
  XB <- pasarABaseDecimalDesdeBinaria(cromosoma.XB)

  funobj(XA, XB)
}
```

Codificadas nuestras variables de decisión y armada la función objetivo, ya podemos usar el paquete "genalg". La funcion a utilizar es *rbga.bin()*, la cual minimiza, por lo tanto tenemos que invertir el signo de nuestra función objetivo. Es este caso, mediante una función anónima:[49]

```
myProb <- rbga.bin(size=9, evalFunc = function(x){-
1*funobjbin(x)}, showSettings = TRUE)

GA Settings
  Type                  =
  Population size       = 200
  Number of Generations = 100
  Elitism               = 40
  Mutation Chance       = 0.1

Search Domain
  Var 1 = [,]
  Var 0 = [,]
```

Noten que demora cierto tiempo en terminar el análisis. Veamos los resultados:

[49] Por razones de espacio, los vectores “population”, “evaluations” y “mean” no se muestran en su totalidad en este libro.

```
myProb

$type
[1] "binary chromosome"

$size
[1] 9

$popSize
[1] 200

$iters
[1] 100

$suggestions
NULL

$population
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    0    1    0    0    1    0    0    0    0
 [2,]    0    1    0    0    1    0    0    0    0
 [3,]    0    1    0    0    1    0    0    0    0
 [4,]    0    1    0    0    1    0    0    0    0
 [5,]    0    1    0    0    1    0    0    0    0
 [6,]    0    1    0    0    1    0    0    0    0
 [7,]    0    1    0    0    1    0    0    0    0
 [8,]    0    1    0    0    1    0    0    0    0
…
$elitism
[1] 40

$mutationChance
[1] 0.1

$evaluations
  [1] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [15] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [29] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [43] -416 -416  -96 -196 -416 -416 -361 -416 -416  -96 -416 -416 -416 -
416 -416
…

$best
  [1] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [15] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [29] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [43] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [57] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [71] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [85] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
 [99] -416 -416

$mean
  [1] -142.230 -207.135 -261.915 -275.175 -329.960 -344.005 -353.000
  [8] -353.885 -352.705 -367.890 -378.365 -365.255 -373.680 -354.060
 [15] -337.930 -360.325 -390.730 -311.695 -369.665 -322.950 -366.875
 [22] -377.155 -362.000 -378.525 -367.885 -370.135 -371.840 -363.525
 [29] -366.775 -356.055 -381.345 -384.490 -373.285 -347.235 -365.435
```

…

```
attr(,"class")
[1] "rbga"
```

Donde "size" es el tamaño de cada cromosoma. Usamos el "showSetting = TRUE" para mostrar la configuración por defecto que utilizamos. Analicemos un poco el objeto que nos devuelve la función:

```
myProb$popSize
[1] 200
```

El algoritmo funcionó con un tamaño de población igual a 200, o sea, en cada iteración se evaluaron 200 potenciales soluciones del problema (en términos de Algoritmos Genéticos, cromosomas). Si vemos las soluciones:[50]

```
myProb$population
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    0    1    0    0    1    0    0    0    0
 [2,]    0    1    0    0    1    0    0    0    0
 [3,]    0    1    0    0    1    0    0    0    0
 [4,]    0    1    0    0    1    0    0    0    0
 [5,]    0    1    0    0    1    0    0    0    0
 [6,]    0    1    0    0    1    0    0    0    0
 [7,]    0    1    0    0    1    0    0    0    0
 [8,]    0    1    0    0    1    0    0    0    0
```

Tenemos una matriz de 200 filas (una por cada cromosoma en la población con la mejor solución alcanzada) y 10 columnas (el tamaño de cada cromosoma que la función supone que estamos utilizando). Para obtener nuestra mejor solución:

```
myProb$evaluations
```

```
[1] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416
-416
[15] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
[29] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -416 -
416 -416
[43] -416 -416  -96 -196 -416 -416 -361 -416 -416  -96 -416 -416 -
416 -416
[57] -416 -416 -416 -416 -320 -196 -416 -416 -320 -416 -320 -416 -
416 -416
[71] -320 -416 -416 -416 -416  -96 -416 -306 -320 -416 -416 -416 -
416 -416
[85] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416  -96 -416 -
416 -416
[99] -416 -416  -96 -320 -416 -416 -416 -416 -416 -196 -416 -416 -
416 -416
```

[50] Nuevamente, no mostramos las 200 soluciones, solo las 8 primeras.

```
[113] -416  -96 -416 -416 -416 -416 -416 -416  -96 -320 -416 -416 -
416 -320
[127] -416 -416 -416 -416 -416 -416 -416 -416 -416 -416    0 -416 -
416 -416
[141] -416 -416 -416 -416 -320 -416 -416 -416 -416 -416 -416 -306 -
416  -96
[155] -416 -416 -416 -416 -416 -416 -136 -320 -416  -96 -416 -416 -
416 -416
[169] -320 -416 -416 -416 -416 -320 -416 -416 -320  -96  -96 -416 -
416 -320
[183] -416 -361 -416 -416 -340 -416 -416 -416  -96 -416  -96  -96  -
96 -416
[197] -416 -196 -416 -416
```

Y tenemos los valores de función objetivo para cada uno de los 200 cromosomas en la población final. Nosotros nos tenemos que quedar con la solución con el mejor valor de función objetivo, entonces si con orden podemos obtener los índices ordenados:

```
order(myProb$evaluations)
```

```
[1]   1   2   3   4   5   6   7   8   9  10  11  12  13  14  15  16
17
[18]  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33
34
[35]  35  36  37  38  39  40  41  42  43  44  47  48  50  51  53  54
55
[52]  56  57  58  59  60  63  64  66  68  69  70  72  73  74  75  77
80
[69]  81  82  83  84  85  86  87  88  89  90  91  92  93  94  96  97
98
[86]  99 100 103 104 105 106 107 109 110 111 112 113 115 116 117 118
119
[103] 120 123 124 125 127 128 129 130 131 132 133 134 135 136 138
139 140
[120] 141 142 143 144 146 147 148 149 150 151 153 155 156 157 158
159 160
[137] 163 165 166 167 168 170 171 172 173 175 176 180 181 183 185
186 188
[154] 189 190 192 196 197 199 200  49 184 187  61  65  67  71  79
102 122
[171] 126 145 162 169 174 177 182  78 152  46  62 108 198 161  45
52  76
[188]  95 101 114 121 154 164 178 179 191 193 194 195 137
```

Podemos hacer:

```
myProb.solucion <-
myProb$population[order(myProb$evaluations)[1],]
myProb.solucion
[1] 0 1 0 0 1 0 0 0 0
```

Si quiero saber a qué valores de mis variables objetivos corresponden, las decodifico:

```
myProb.XA <-
pasarABaseDecimalDesdeBinaria(myProb.solucion[1:4])
myProb.XB <-
pasarABaseDecimalDesdeBinaria(myProb.solucion[5:9])
```

Y ya que estamos:

```
myProb.obj <- -1 * min(myProb$evaluations)
```

Entonces:

```
myProb.XA
[1] 4

myProb.XB
[1] 16

myProb.obj
[1] 416
```

rbga.bin() trabajó con casi todos los parámetros por defecto, pero podemos modificarlos antes de hacer otra corrida:

```
myProb <- rbga.bin(size=9, popSize = 100, iters = 500,
                   mutationChance = 0.05, elitism = 10,
                   evalFunc = function(x){-1*funobjbin(x)},
                   showSettings = TRUE)
GA Settings
  Type                  =
  Population size       = 100
  Number of Generations = 500
  Elitism               = 10
  Mutation Chance       = 0.05

Search Domain
  Var 1 = [,]
  Var 0 = [,]

myProb.solucion <-
myProb$population[order(myProb$evaluations)[1],]
myProb.XA <-
pasarABaseDecimalDesdeBinaria(myProb.solucion[1:4])
myProb.XB <-
pasarABaseDecimalDesdeBinaria(myProb.solucion[5:9])
myProb.obj <- -1 * min(myProb$evaluations)

myProb.XA
[1] 4

myProb.XB
[1] 16

myProb.obj
[1] 416
```

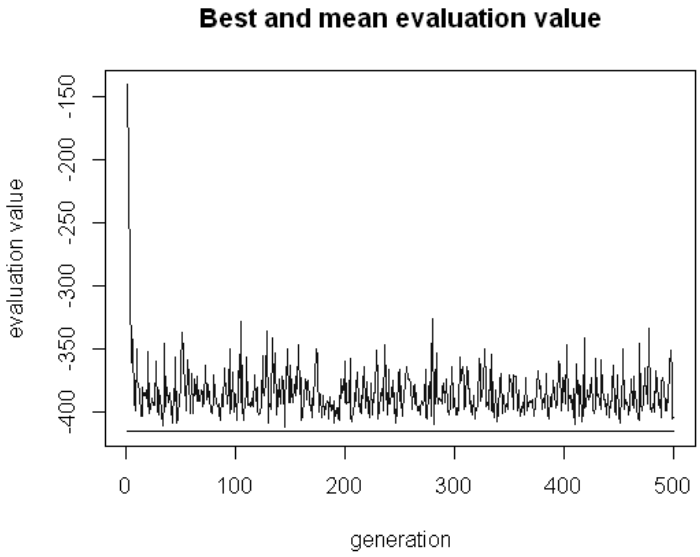
La función *rbga.bin()* es muy interesante para analizar cómo evoluciona la población en cada iteración. Analizando el objeto que devuelve, tenemos los atributos “mean” y “best”, que nos muestran para cada

iteración el promedio del valor objetivo y el mejor valor objetivo respecto de la población vigente en ese momento:

```
myProb$mean
myProb$best
```

Podemos ver la evolución de las soluciones en un gráfico:

```
plot(myProb)
```



8.2. Armando nuestras propias funciones

El paquete "genalg" es bastante útil para la aplicación práctica de los Algoritmos Genéticos, pero quizás nos interese armar nuestra propia biblioteca de funciones para Algoritmos Genéticos, de manera de probar otras configuraciones o nuevas técnicas. Así que armemos una función básica de optimización basada en este paradigma para tener de ejemplo [51]. Primero que todo, armemos una función que nos genere una población inicial aleatoria:

```
generarPoblacionInicial <- function(tamanoCromosoma,
tamanoPoblacion){
```

[51] Nuevamente, recordemos de guardar estas funciones en un script independiente.

```
  matrix(round(runif(tamanoCromosoma * tamanoPoblacion)),
                              nrow = tamanoPoblacion,
                              ncol = tamanoCromosoma,
                              byrow = TRUE)

}
```

Vamos a probarla :

```
generarPoblacionInicial(5,10)
```

```
      [,1] [,2] [,3] [,4] [,5]
 [1,]    1    0    0    0    0
 [2,]    0    1    0    1    1
 [3,]    0    0    0    1    0
 [4,]    0    1    0    0    1
 [5,]    1    0    1    1    1
 [6,]    0    0    1    0    1
 [7,]    1    1    1    0    1
 [8,]    0    0    0    0    1
 [9,]    1    1    1    1    0
[10,]    1    1    1    1    1
```

Hagamos una función para seleccionar las “n” mejores:

```
seleccionarMejores <- function (poblacion, funcionObjetivo,
cantMejores, minimizar = TRUE){
  valoresObj <- c()
  for (i in 1:nrow(poblacion)){
    valoresObj <- c(valoresObj,
funcionObjetivo(poblacion[i,1:ncol(poblacion)]))
  }
  order(valoresObj, decreasing = !minimizar)[1:cantMejores]
}
```

Observemos que hay que pasarle un parámetro denominado “funciónObjetivo”, que es la función de evaluación que utilizará para seleccionar las soluciones. En este caso, le pasamos una función en forma implícita que lo que hace es sumar las componentes del vector a evaluar. Probémosla:

```
datos <- generarPoblacionInicial(5,10)
datos
```

```
      [,1] [,2] [,3] [,4] [,5]
 [1,]    1    0    1    1    0
 [2,]    1    1    1    1    1
 [3,]    0    1    1    1    1
 [4,]    0    1    1    0    0
 [5,]    0    0    1    1    0
 [6,]    1    0    1    1    1
 [7,]    0    0    0    1    0
 [8,]    0    1    1    0    0
 [9,]    1    1    1    0    1
[10,]    1    0    0    0    0
```

```
seleccionarMejores(datos, function(x){sum(x)}, cantMejores =
3, minimizar = FALSE)
[1] 2 3 6
```

Ya sabemos cómo generar la población inicial y como seleccionar por las mejores. Veamos cómo mutar a poblacion:

```
mutarCromosomas <- function(poblacion, porcentajeMutacion){
  cantCromosomas <- nrow(poblacion)
  cromosomasMutados <- ceiling(poblacion *
porcentajeMutacion)
  cromosomasSeleccionados <-
sample(1:cantCromosomas,cromosomasMutados,replace=FALSE)
  bitsSeleccionados <-
sample(1:ncol(poblacion),cromosomasMutados,replace=TRUE)

  nuevaPoblacion <- poblacion
  nuevaPoblacion[cromosomasMutados,bitsSeleccionados] <-
    ((poblacion[cromosomasMutados,bitsSeleccionados]+1) %%
2)

  nuevaPoblacion
}
```

Probemos nuestra función de mutación:

```
datos
      [,1] [,2] [,3] [,4] [,5]
 [1,]    1    0    1    1    0
 [2,]    1    1    1    1    1
 [3,]    0    1    1    1    1
 [4,]    0    1    1    0    0
 [5,]    0    0    1    1    0
 [6,]    1    0    1    1    1
 [7,]    0    0    0    1    0
 [8,]    0    1    1    0    0
 [9,]    1    1    1    0    1
[10,]    1    0    0    0    0
```

```
mutarCromosomas(datos, 0.25)
      [,1] [,2] [,3] [,4] [,5]
 [1,]    1    1    1    1    0
 [2,]    1    1    1    1    1
 [3,]    0    1    1    1    1
 [4,]    0    1    1    0    0
 [5,]    0    0    1    1    0
 [6,]    1    0    1    1    1
 [7,]    0    0    0    1    0
 [8,]    0    1    1    0    0
 [9,]    1    1    1    0    1
[10,]    1    0    0    0    0
```

Y ahora armemos la función para generar hijos:

```
generarHijosDeMadreYPadre <- function(madre, padre,
cantHerenciaMadre){
  hijo <- padre
```

```
  herenciaMadre <- sample(1:length(madre),
cantHerenciaMadre, replace = FALSE)
  hijo[herenciaMadre] <- madre[herenciaMadre]
  hijo

}
```

El parámetro “cantHerenciaMadre” define cuantos genes provienen de la madre, el resto son los genes del padre. La probamos:

```
generarHijosDeMadreYPadre(c(1,1,1,1,1), c(0,0,0,0,0), 3)
[1] 1 0 1 0 1

generarHijosDeMadreYPadre(c(1,1,1,1,1), c(0,0,0,0,0), 3)
[1] 0 1 1 1 0

generarHijosDeMadreYPadre(c(1,1,1,1,1), c(0,0,0,0,0), 3)
[1] 0 0 1 1 1
```

Como vemos, definimos cuantos genes hereda de la madre, pero la selección de dichos genes se produce aleatoriamente, a fin de poder generar hijos distintos de una misma pareja.

Tenemos que armar ahora una gran función que tome como parámetros todos los parámetros necesarios para ejecutar todas las funciones definidas en la biblioteca recién cargada. Su prototipo sería:

```
myAlgoritmoGenetico <- function(tamanoCromosoma,
tamanoPoblacion,cantMejores,porcentajeMutacion,cantHerenciaM
adre, nroIteraciones = 100, funcionObjetivo, minimizar =
TRUE){
}
```

Como vemos, tomamos todos los parámetros de todas las funciones, excepto aquellos que se pueden obtener de resultados de otras. Empecemos a armarla:

```
myAlgoritmoGenetico <- function(tamanoCromosoma,
tamanoPoblacion,cantMejores,porcentajeMutacion,cantHerenciaM
adre, nroIteraciones = 100, funcionObjetivo, minimizar =
TRUE){

  poblacion <- generarPoblacionInicial(tamanoCromosoma,
tamanoPoblacion)
  mejoresCromosomas <- c()
}
```

Para empezar, lo básico. En base al tamaño del cromosoma y el tamaño de la población deseada, generamos nuestra población inicial. Ya que

estamos, también creamos una variable del tipo vector, vacía por ahora, que se llama "mejoresCromosomas", y que nos servirá para almacenar los mejores individuos de la solución. Sigamos:

```
myAlgoritmoGenetico <- function(tamanoCromosoma,
tamanoPoblacion,cantMejores,porcentajeMutacion,cantHerenciaM
adre, nroIteraciones = 100, funcionObjetivo, minimizar =
TRUE){

  poblacion <- generarPoblacionInicial(tamanoCromosoma,
tamanoPoblacion)
  mejoresCromosomas <- c()

  for (i in 1:nroIteraciones){
    mejoresCromosomas <-
seleccionarMejores(poblacion,funcionObjetivo,cantMejores,
minimizar)
    nuevaPoblacion <- poblacion[mejoresCromosomas,]
  }
}
```

Agregamos ahora un bucle que se repite tantas veces como iteraciones hayamos pasado como parámetro. En cada iteración, lo primero que hacemos es seleccionar los mejores individuos de la población y los copiamos a una nueva variable. Recuerden que "poblacion" es una matriz, "mejoresCromosomas" un vector y "nuevaPoblacion" es entonces una matriz que proviene de filtrar a la "poblacion" actual por las filas asociadas a los mejores individuos de dicha población. Si estoy en la primer iteración, "poblacion" almacena a mi población actual, en iteraciones posteriores almacenas a las nuevas poblaciones que vayamos creando.

Sigamos:

```
myAlgoritmoGenetico <- function(tamanoCromosoma,
tamanoPoblacion,cantMejores,porcentajeMutacion,cantHerenciaM
adre, nroIteraciones = 100, funcionObjetivo, minimizar =
TRUE){

  poblacion <- generarPoblacionInicial(tamanoCromosoma,
tamanoPoblacion)
  mejoresCromosomas <- c()

  for (i in 1:nroIteraciones){
    mejoresCromosomas <-
seleccionarMejores(poblacion,funcionObjetivo, cantMejores,
minimizar)
    nuevaPoblacion <- poblacion[mejoresCromosomas,]
    for (j in 1:(nrow(poblacion) -
length(mejoresCromosomas))){
```

```
      padres <- sample(mejoresCromosomas, 2, replace =
FALSE)
      nuevaPoblacion <- rbind(nuevaPoblacion,
generarHijosDeMadreYPadre(poblacion[padres[1],],poblacion[pa
dres[2],],cantHerenciaMadre))
    }
  }
}
```

Dentro del bucle principal (el asociado a las iteraciones) creo un nuevo bucle encargado de generar una nueva población en base a la selección de los mejores individuos. Para ello, repito el bucle la cantidad de veces que hagan falta para llenar la matriz "nuevaPoblacion" con el mismo tamaño que la poblacion original, pero generando cada elemento mediante el cruzamiento de las mejores soluciones. Por tanto, selecciono los padres aleatoriamente en cada iteración, los cruzo y los agrego a la nueva población. Noten que la aleatoriedad para generar hijos es alta, ya que los padres se eligen al azar, y los hijos se generan seleccionando cromosomas al azar de sus padres.

Sigamos:

```
myAlgoritmoGenetico <- function(tamanoCromosoma,
tamanoPoblacion,cantMejores,porcentajeMutacion,cantHerenciaM
adre, nroIteraciones = 100, funcionObjetivo, minimizar =
TRUE){

  poblacion <- generarPoblacionInicial(tamanoCromosoma,
tamanoPoblacion)
  mejoresCromosomas <- c()

  for (i in 1:nroIteraciones){
    mejoresCromosomas <-
seleccionarMejores(poblacion,funcionObjetivo, cantMejores,
minimizar)
    nuevaPoblacion <- poblacion[mejoresCromosomas,]
    for (j in 1:(nrow(poblacion) -
length(mejoresCromosomas))){
      padres <- sample(mejoresCromosomas, 2, replace =
FALSE)
      nuevaPoblacion <- rbind(nuevaPoblacion,
generarHijosDeMadreYPadre(poblacion[padres[1],],poblacion[pa
dres[2],],cantHerenciaMadre))
    }
    poblacion <- mutarCromosomas(nuevaPoblacion,
porcentajeMutacion)
  }
  poblacion
}
```

Por último, me aseguro de mutar un poco la población al final de cada iteración del bucle principal, y devuelvo la última población generada al finalizar dicho bucle.
Probémoslo a ver que devuelve. Si maximizamos, como la función objetivo es la suma de sus componentes, la población devuelta debería tener mayoría de vectores con todos sus componentes valiendo 1. Para el caso de minimización, la mayoría de los vectores debería tener sus componentes valiendo 0:[52]

```
myAlgoritmoGenetico(9,100,10, 0.1,5,100,function(x){sum(x)},
minimizar=FALSE)
```

```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    1    1    1    0    1    1    1    1    1
 [2,]    1    1    1    1    1    1    1    1    1
 [3,]    1    1    1    1    1    1    1    1    1
 [4,]    1    1    1    1    1    1    1    1    1
…
```

```
myAlgoritmoGenetico(9,100,10, 0.1,5,100,function(x){sum(x)},
minimizar=TRUE)
```

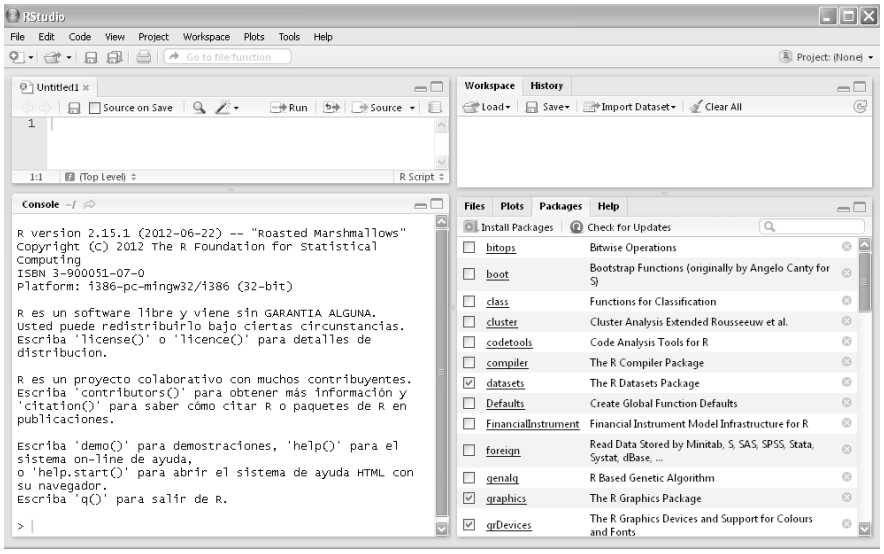
```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    0    0    0    0    0    0    0    0    0
 [2,]    0    0    0    0    0    0    0    0    0
 [3,]    0    0    0    0    0    0    0    0    0
 [4,]    0    0    0    0    0    0    0    0    0
 [5,]    0    0    0    0    0    0    0    0    0
…
```

[52] Nuevamente, no se muestran los resultados completos, por cuestiones de espacio.

Anexo I: Interacción con RStudio

¿Por qué utilizar RStudio?

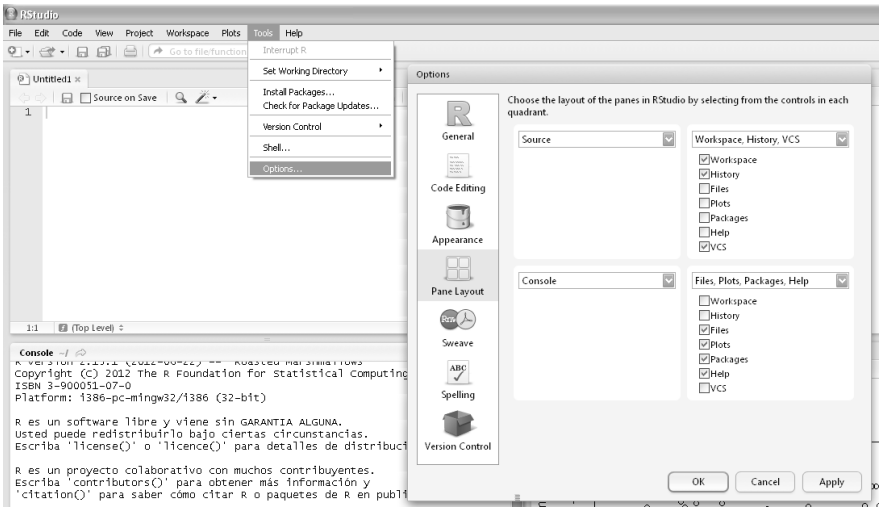
Si bien la distribución estándar de R se puede utilizar sin problemas, las IDE basadas en ella nos ofrecen algunas facilidades extras para el trabajo diario. Una de las distribuciones más populares es RStudio.
RStudio nos presenta una interfaz clara que, además de la consola, nos permite monitorear y acceder a varias estructuras del sistema.



En la configuración que nosotros utilizamos, podemos obtener, adicionalmente al acceso a la consola (panel inferior izquierdo), acceso al listado de paquetes instalados, con la posibilidad de cargarlos en memoria y consultar su ayuda con un clic del mouse (panel inferior derecho), un detalle de todos los objetos cargados en memoria (panel superior derecho) y un editor de script para no tener que salir de R cuando queremos guardar nuestras funciones a archivos (panel superior izquierdo). Pero mejor, vayamos paso a paso.

Configurando el layout de visualización

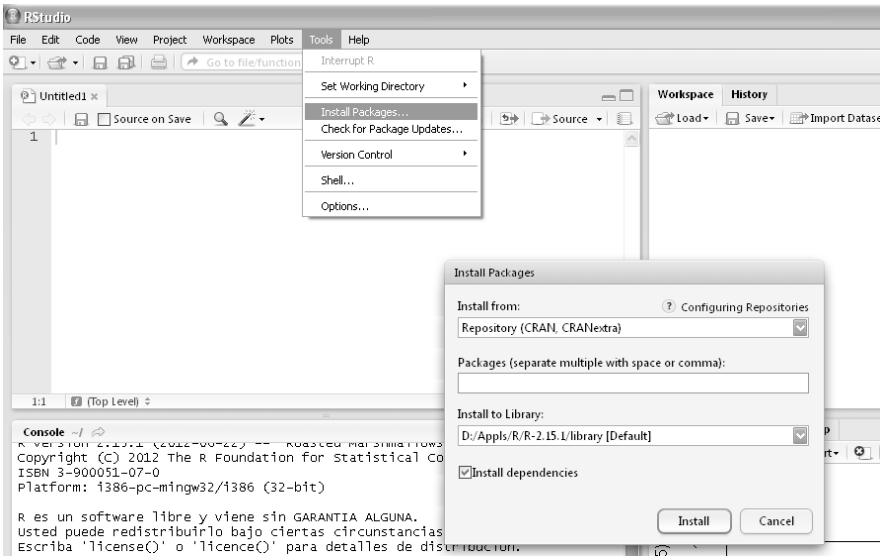
La interfaz gráfica de RStudio es altamente personalizable. Desde el menú "**Tools**", mediante el submenú "**Options**" podemos acceder a la ventana de configuración, dentro de la cual encontramos la solapa "**Panel Layout**":



En esta ventana podemos definir el contenido de cada una de las cuatro secciones del layout, seleccionando las opciones los desplegables de cada cuadro y tildando o destildando las pestañas que queremos ver.

Trabajando con paquetes

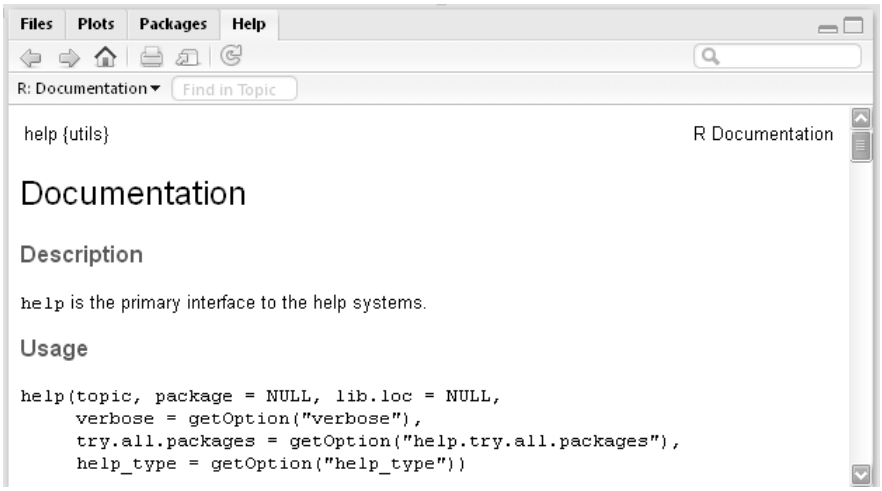
La gestión de paquetes se puede hacer directamente desde la interfaz gráfica. Desde "**Tool/Install Packages…**" podemos sustituir al comando *install.packages()*:



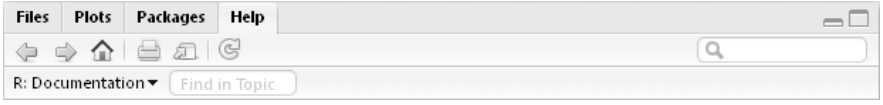
Desde el cuadro de diálogo emergente seleccionamos el repositorio que queremos utilizar, tipeamos los paquetes a instalar y definimos en cual biblioteca queremos instalarlo.

Consultando la ayuda

Desde la consola podemos acceder a los comandos habituales de ayuda, como *help()*. Sin embargo, a diferencia del entorno estándar de R la ayuda se abre en uno de los paneles del entorno, en vez de en una ventana adicional:

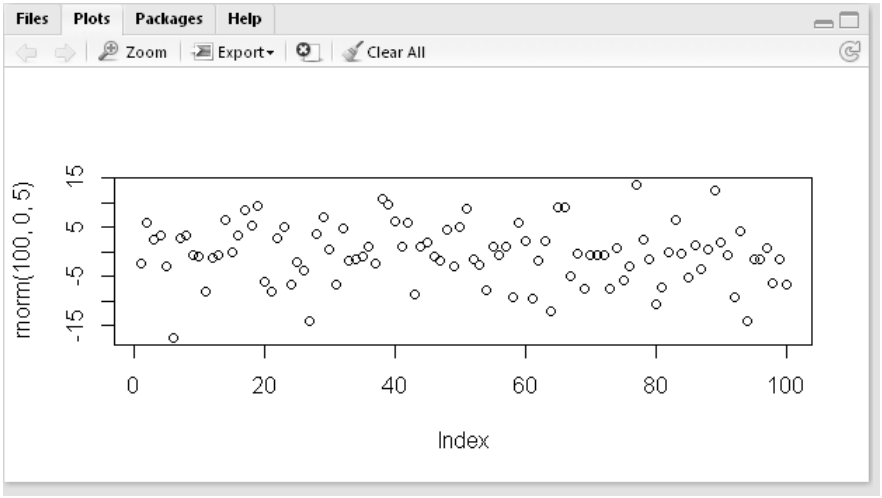


Se puede acceder en forma directa desde la pestaña "**Help**". Además. El panel ofrece una barra de navegación, con comandos para ir atrás, adelante, a la portada de la ayuda, imprimirla, mostrarla en una nueva ventana (en el navegador de internet activo), refrescar el tema y buscar algún tópico (a la derecha de la barra, equivale a usar *help.search()*):



Visualizando gráficos

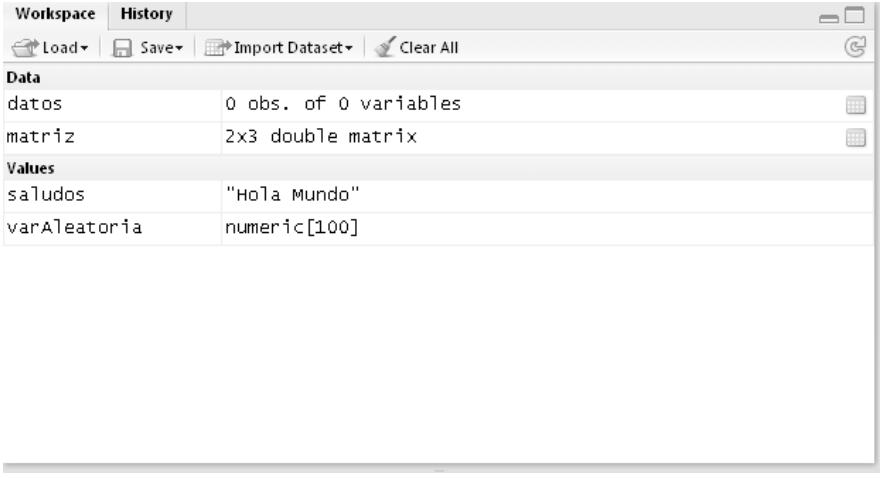
Dentro de los paneles tenemos la opción de visualización de gráficos:



RStudio lleva un historial de gráficos dibujados, por lo tanto, podemos acceder a los mismos sin necesidad de volver a tipear los comandos con los cuales lo generamos.
Dentro de este panel, podemos ver los gráficos anteriores y posteriores que estén grabados en el historial, hacerles zoom (en una ventana externa), grabarlos a un archivo, eliminar el gráfico del historial o borrar todo el historial.

Monitoreando la memoria de trabajo

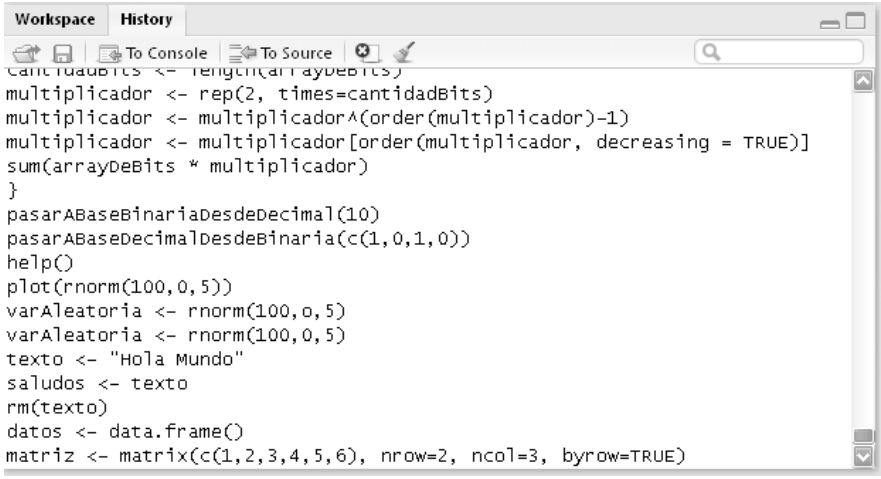
Una forma más rápida de revisar el contenido de la memoria de trabajo, sin necesidad de llamar al comando *ls()*, es visualizar el panel "**Workspace**":



Desde el panel podemos abrir un workspace existente, grabarlo en memoria, cargar un conjunto de datos desde archivo (siendo equivalente a, entre otros, hacer un *read.csv()*) y limpiar la memoria de trabajo (com0 si hiciéramos *rm(list=ls())*). Además, haciendo clic sobre alguno de los datos almacenados en la memoria, llamamos al comando *fix()*.

Revisando el historial

Otra muy buena funcionalidad es poder revisar el historial de comandos ingresados a la consola desde la pestaña "**History**":



Escribiendo nuestros script

Para escribir nuestros propios script no es necesario utilizar un editor externo. Tenemos el panel "**Source**" para ello:

Desde el mismo podemos generar guardar nuestros scripts, disponemos de algunas funcionalidades de edición básicas (como por ejemplo, buscar y reemplazar) y tenemos la posibilidad de mandar los sentencias escritas en el script a la consola, o bien hacer un *source()* y ejecutar todo el script. Todos los archivos que creemos desde "**File/New**" o que abramos desde "**File/Open**" se abrirán en este panel. Además, desde el mismo, tenemos la posibilidad de compilar lo escrito como HTML, a fin de generar un archivo de documentación, por ejemplo.

Ejecutando View()

Por último, en RStudio, cada vez que llamemos a la *View()* no se nos abre una nueva ventana, sino que se nos muestran los datos en una nueva pestaña del panel "**Source**":

Anexo II: Más paquetes para optimización

Como buscar más paquetes de optimización

Las posibilidades del uso de R para optimización no se limitan a lo visto aquí. Existen numerosos paquetes registrados en CRAN para optimización. Uno puede consultarlo en la "Task View" dedicada al tema "Optimization and Mathematical Programming"[53]. No podemos tratar todos, ni tampoco darles toda la profundidad que se merecen pero, igualmente, a continuación reseñamos algunos de estos paquetes.

ROI

Este paquete presenta un marco de trabajo para modelar problemas de programación lineal y no lineal independientemente del solver a utilizar.

optimx

Paquete que permite encapsular la llamada a diferentes métodos de optimización para problemas no lineales dentro de una única función (*optimx*). Basicamente es una extensión de la función *optim*, pero que puede llamar a muchos más algoritmos de resolución.

dfoptim

Paquete para optimizaciones no lineales mediante métodos de búsquedas sin necesidad de declarar gradientes. Implementa algoritmos de Nelder-Mead y Hooke-Jeeves.

hydroPSO

Poderosísimo paquete que implementa algoritmos de Optimización de Enjambre de partículas.

[53] Al día de la publicación de este libro, el enlace a dicha Task View es: http://cran.r-project.org/web/views/Optimization.html

rgenoud

Paquete que implementa la función *rgenoud*, la cual permite resolver problemas de optimización mediante la aplicación de una combinación de algoritmos evolutivos con métodos basados en derivadas.

DEoptim

Paquete para resolución de problemas mediante técnicas de optimización basadas en algoritmos de evolución diferencial.

Bibliografía

[1]. R Core Team (2012). R: A language and environment for statistical computing. R Foundation for Statistical Computing, Vienna, Austria. ISBN 3-900051-07-0, URL http://www.R-project.org/.

[2]. R Development Core Team. 2000. Introducción a R, V1.0.1.

[3]. A. J. Arriaza Gómez y otros. 2008. Estadística Básica con R y RCommander. Ediciones UCA

[4]. R CRAN. Mathemtatical Programming Task View.

[5]. Norman Matloff. From Algorithms to Z-Scores: Probabilistic and Statistical Modeling in Computer Science. http://heather.cs.ucdavis.edu/matlo/probstatbook.html

[6]. Norman Matloff. R for Programmers. http://heather.cs.ucdavis.edu/~matloff/132/NSPpart.pdf

[7]. Paula Elosua. Introducción al entorno R. ISBN: 978-84-9860-497-9

[8]. Andrew Robinson. Icebreaker. http://cran.r-project.org/doc/contrib/Robinson-icebreaker.pdf

[9]. Sean Luke. Essentials of Metaheuristics. http://cs.gmu.edu/~sean/book/metaheuristics/

[10]. Jason Brownlee. Clever Algorithms. http://www.cleveralgorithms.com/

[11]. R-Enthusiasts: http://gallery.r-enthusiasts.com

Los problemas del mundo real son, en general, no lineales enteros mixtos y de gran dimensión. Poder resolver esa clase de problemas es de vital importancia para todo profesional. Es conveniente disponer de algoritmos eficientes y robustos para su resolución.

El software R, ha demostrado ser muy eficiente para programar e implementar las técnicas estadísticas y los algoritmos de optimización más tradicionales.

ISBN 978-84-686-3748-8