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

© Optimización matemática con R. Volumen I
Introducción al modelado y resolución de problemas
© Enrique Gabriel Baquela
© Andrés Redchuk
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

¿Cómo leer este libro? **vii**
Referencias vii
Estructura del libro ix

Índice de contenido **xi**

Introducción **13**
1.1. ¿Qué es un problema de optimización? 13
1.2. Modelado de sistemas Vs Resolución de modelos 14
1.3. Clasificación de problemas de optimización 14
1.4. Clasificación de métodos de resolución 15
Resolución mediante cálculo 15
Resolución mediante técnicas de búsquedas 16
Resolución mediante técnicas de convergencia de soluciones 16
1.5. Optimización Convexa 17

Introducción al uso de R **19**
2.1 Actividades previas. 19
2.2. Utilizando R 21
2.3. Acerca del trabajo con matrices en R 40
2.4. Arrays 48
2.5. Listas 50

Un poco más de R **53**
3.1. Utilizando funciones en R 53
3.2. Guardando nuestras funciones 58
3.3. Armando scripts 59

Gráficos con R **61**
4.1. Trabajando con el paquete graphics 61
4.2. Trabajando con layouts 75

Implementando nuestras primeras búsquedas **83**
5.1. Acerca de las búsquedas por fuerza bruta 83
5.2. Búsqueda aleatoria 89
5.3. Búsqueda dirigida por gradientes 91

Algoritmos para programación lineal **93**
6.1. Introducción a la Programación Lineal 93

6.2. Cómo resolver problemas de PL 93
6.3. Otros paquetes de PL: lpSolve 96
6.4. Otros paquetes de PL: lpSolveAPI 105
6.5. Optimizando en base a muestras 109

Algoritmos para programación no lineal **115**
7.1. La función OPTIM 115
7.2. Optimizando con restricciones 123
7.3. Armando nuestra propia versión de OPTIM 124

Algoritmos genéticos **129**
8.1. Algoritmos genéticos 129
8.2. Armando nuestras propias funciones 137

Anexo I: Interacción con RStudio **145**
¿Por qué utilizar RStudio? 145
Configurando el layout de visualización 145
Trabajando con paquetes 146
Consultando la ayuda 147
Visualizando gráficos 147
Monitoreando la memoria de trabajo 148
Revisando el historial 149
Escribiendo nuestros script 149
Ejecutando View() 150

Anexo II: Más paquetes para optimización **151**
Como buscar más paquetes de optimización 151

Bibliografía **153**

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.

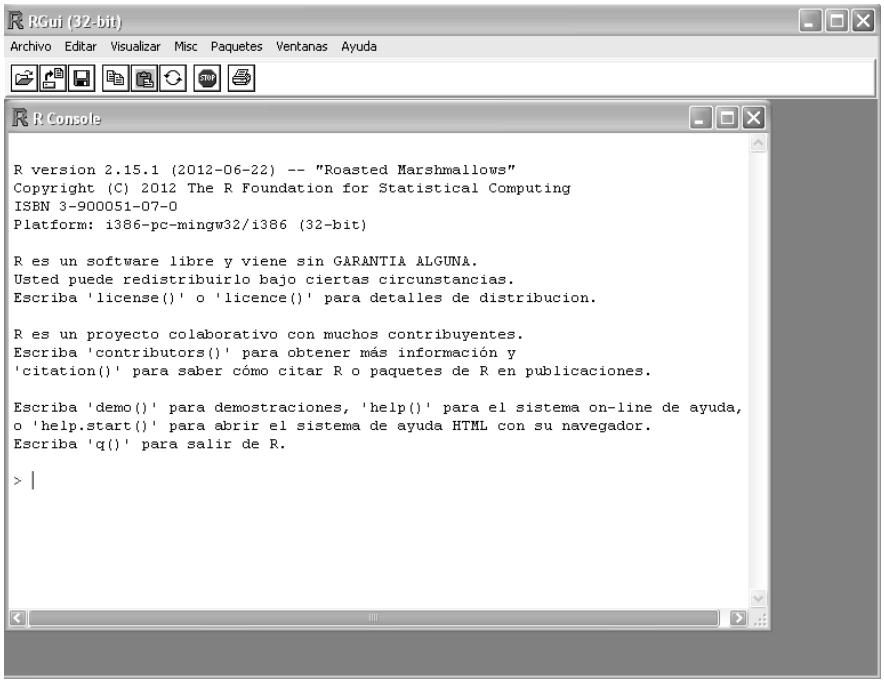
Dentro de los casos especiales, tenemos los siguientes tipos de problemas:

- 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

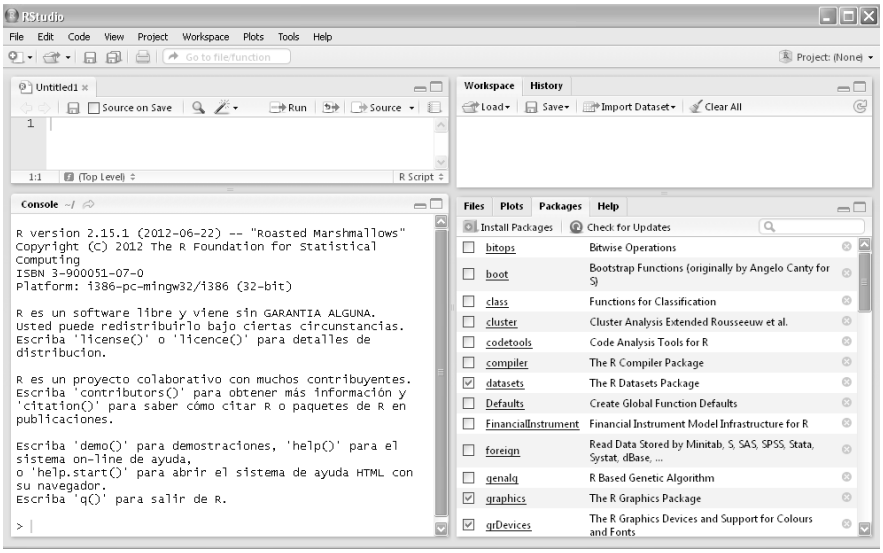
2.1 Actividades previas.

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
```

```
                [,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=r estriccionesA1, b1=termIndepb1          , A3=restriccionesA3,
b3=termIndepb3, maxi=TRUE )
simplexProblemaMezcla
```

```
Linear Progra    mming 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.
```

```
simple  xProblemaMezcla$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
```

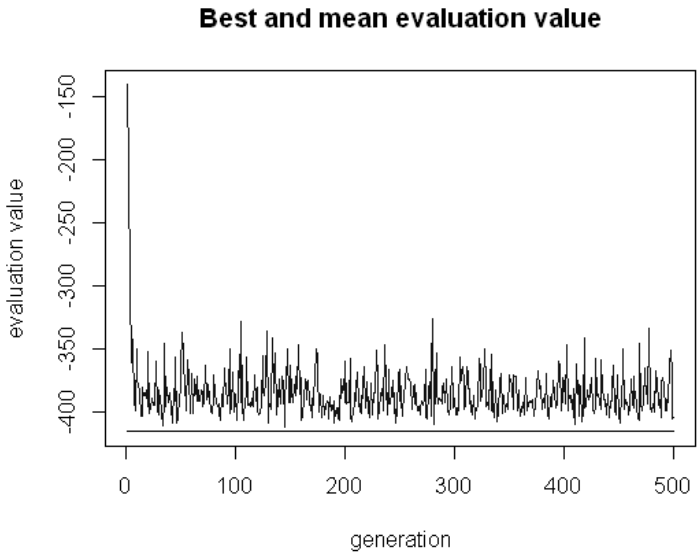
La función *rbga.bin()* es muy interesante para analizar cómo evoluciona la población en cada iteración. Analizando el objeto que devuelve, tenemos los atributos “mean” y “best”, que nos muestran para cada

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)
```

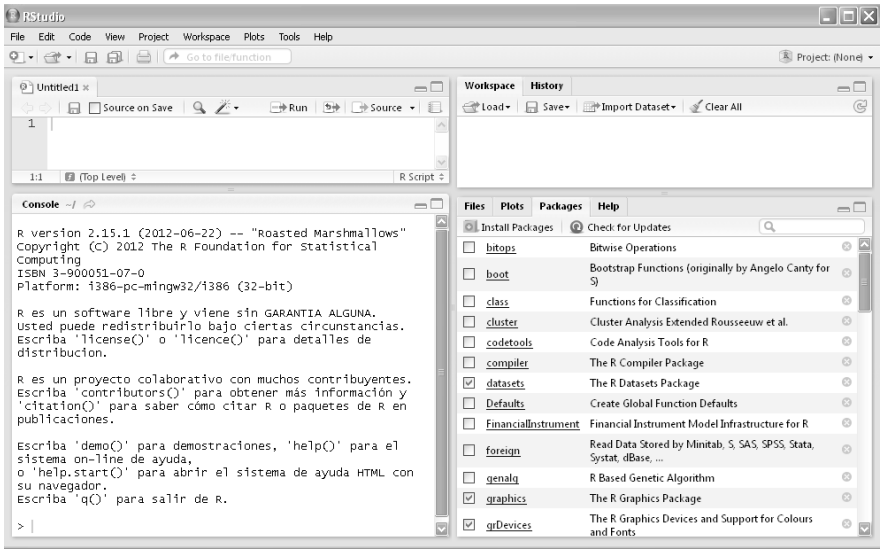
```
      [,1] [,2] [,3] [,4] [,5] [,6] [,7] [,8] [,9]
 [1,]    0    0    0    0    0    0    0    0    0
 [2,]    0    0    0    0    0    0    0    0    0
 [3,]    0    0    0    0    0    0    0    0    0
 [4,]    0    0    0    0    0    0    0    0    0
 [5,]    0    0    0    0    0    0    0    0    0
…
```

[52] Nuevamente, no se muestran los resultados completos, por cuestiones de espacio.

Anexo I: Interacción con RStudio

¿Por qué utilizar RStudio?

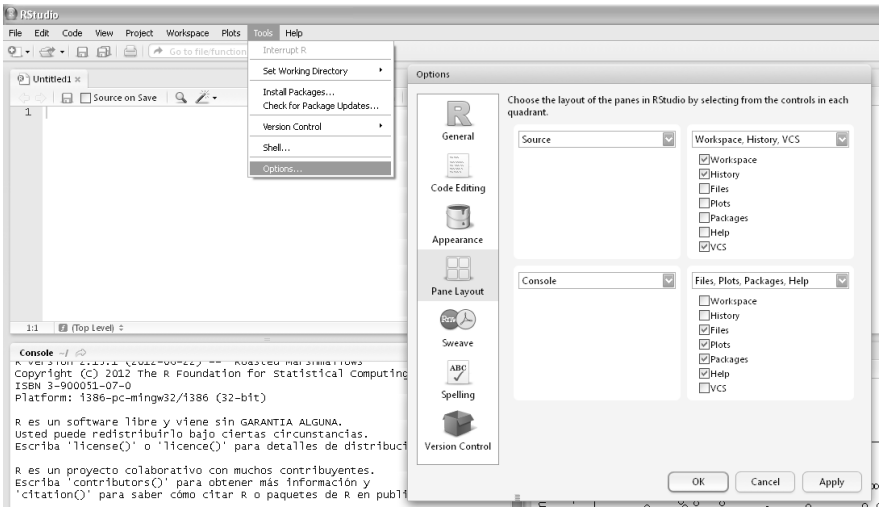
Si bien la distribución estándar de R se puede utilizar sin problemas, las IDE basadas en ella nos ofrecen algunas facilidades extras para el trabajo diario. Una de las distribuciones más populares es RStudio.
RStudio nos presenta una interfaz clara que, además de la consola, nos permite monitorear y 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

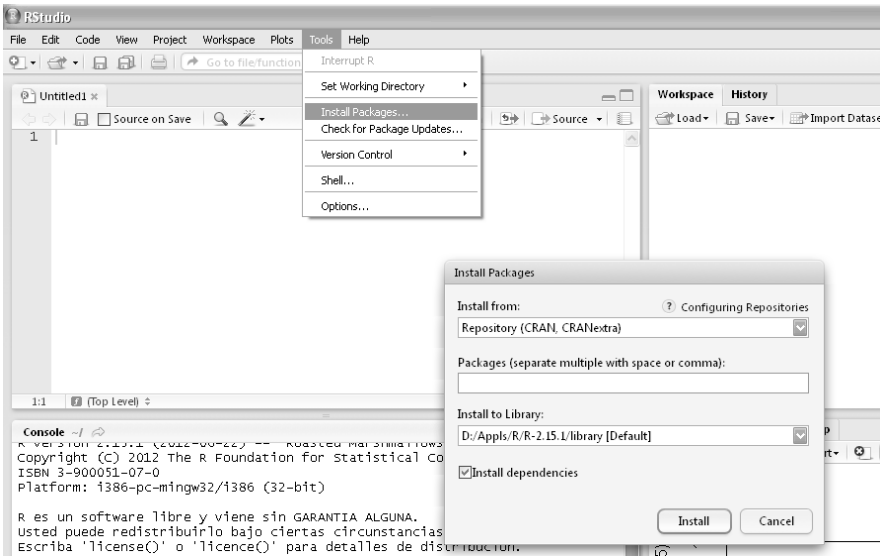
La interfaz gráfica de RStudio es altamente personalizable. Desde el menú "**Tools**", mediante el submenú "**Options**" podemos acceder a la ventana de configuración, dentro de la cual encontramos la solapa "**Panel Layout**":



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

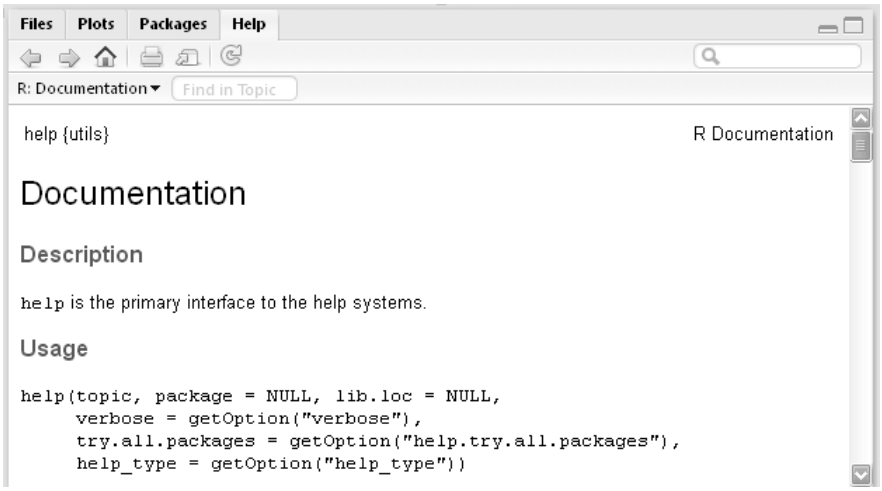
La gestión de paquetes se puede hacer directamente desde la interfaz gráfica. Desde "**Tool/Install Packages…**" podemos sustituir al comando *install.packages()*:



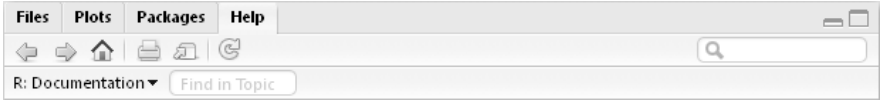
Desde el cuadro de diálogo emergente seleccionamos el repositorio que queremos utilizar, tipeamos los paquetes a instalar y definimos en cual biblioteca queremos instalarlo.

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:

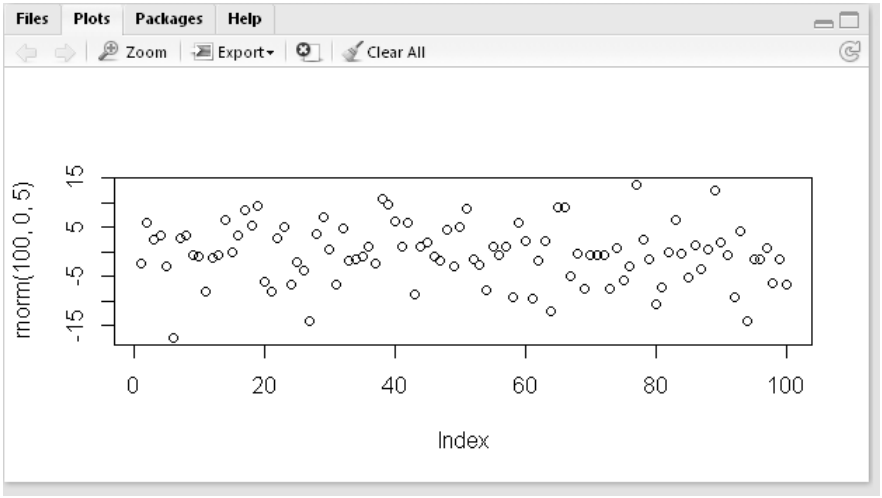


Se puede acceder en forma directa desde la pestaña "**Help**". Además. El panel ofrece una barra de navegación, con comandos para ir atrás, adelante, a la portada de la ayuda, imprimirla, mostrarla en una nueva ventana (en el navegador de internet activo), refrescar el tema y buscar algún tópico (a la derecha de la barra, equivale a usar *help.search()*):



Visualizando gráficos

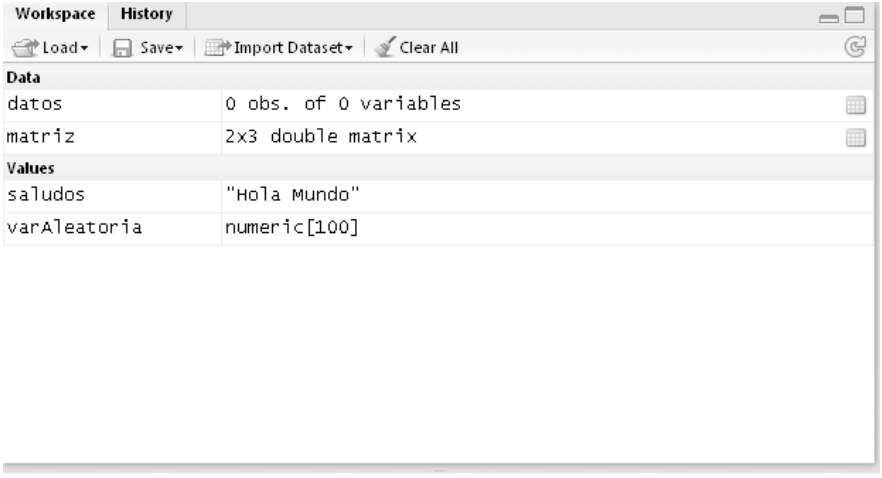
Dentro de los paneles tenemos la opción de visualización de gráficos:



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

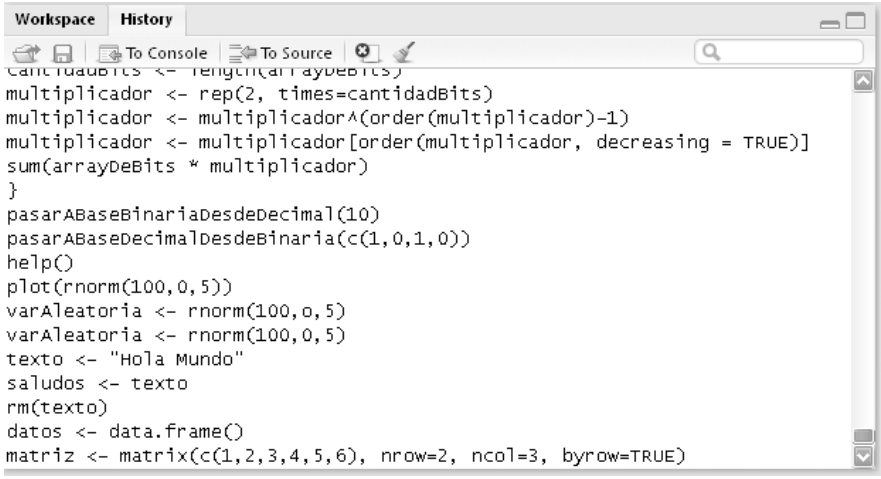
Una forma más rápida de revisar el contenido de la memoria de trabajo, sin necesidad de llamar al comando *ls()*, es visualizar el panel "**Workspace**":



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