Índice

1. Introducción a R y RStudio

1.1. Primeros pasos con R

Después de iniciar RStudio, es mejor comenzar creando un nuevo script (clic en Archivo>Nuevo>Script R o haciendo clic en el botón más a la izquierda) o abrir un script existente (haciendo clic en Archivo>Abrir...). Se pueden escribir comandos de R directamente en la consola que se ejecuta en la parte inferior izquierda de RStudio. Sin embargo, normalmente es mejor escribir los comandos o instrucciones en el editor en la parte superior izquierda. A continuación, los comandos se pueden enviar a R resaltando los comandos y haciendo clic en el botón Ejecutar. Al presionar Ctrl-Enter también se envía la selección actual o, si no se selecciona texto, la línea actual de código. La parte superior derecha tiene una lista de los objetos actuales en el espacio de trabajo, así como una segunda pestaña con un historial de comandos utilizados en el pasado. La parte inferior derecha contiene cuatro pestañas: una que muestra los archivos en el directorio de trabajo actual, otra que muestra los gráficos dibujados hasta ahora, una que muestra todos los paquetes de extensión disponibles y finalmente (y lo más importante) una que muestra la ayuda de R.

1.1.1. Usando R como calculadora

Operador	Significado	Expresión	Resultado
+	Suma	$3+2$	5
-	Resta	$3-2$	1
*	Multiplicación escalar	$3*2$	6
/	División	$5/2$	2.5
%/%	División entera	$5\,\%/\%2$	2
%%	Resto de la división entera	$5\,\%\%2$	1
^o **	Potencia	5^2	25

Hay una gran variedad de funciones matemáticas disponibles en R, como abs, sqrt, exp, log, sin, cos, tan, etc. La variable pi contiene el valor de π. Y la constante e se puede generar como exp(1) . Si una expresión contiene más de un operador, R usa las siguientes reglas (precedencia de operadores):

1. Primero evalúa ^o **, luego %% o %/%, luego * o /, y finalmente + o -.
2. En caso de empates (operadores de igual precedencia) las expresiones se evalúan de izquierda a derecha.

1.1.2. Variables y asignaciones

El operador de asignación predeterminado en R es <-. Para almacenar el resultado de $2/3*2$ en una variable llamada a se puede usar:

```
a <- 2/3 * 2
```

También se puede usar el operador `=` en lugar de `<-` en la mayoría de las circunstancias (pero no en todas).

Los nombres de las variables distinguen entre mayúsculas y minúsculas, es decir, se puede definir `a` y `A`, y `a` y `A` pueden contener valores diferentes. Históricamente, la mayoría de los usuarios de `R` solo usaban letras minúsculas y separaban las palabras con puntos, aunque los guiones bajos se han vuelto cada vez más populares en los nombres de variables y funciones.

Se pueden definir nuevas variables utilizando los valores almacenados en otras variables:

```
b <- a/5
```

Hay que tener en cuenta que cambiar el valor de `a` no cambiará automáticamente `b` , por lo que en el ejemplo anterior:

```
a <- 10
b <- a/5
a <- a + 30
b
```

```
## [1] 2
```

`b` seguirá siendo 2, aunque `a` sea 40, entonces `b` no es $a/5$ (en una hoja de cálculo como Excel se habría actualizado automáticamente).

El uso de variables es fundamental en la programación. Podemos listar todas las variables en el espacio de trabajo actual usando:

```
ls()
```

```
## [1] "a" "b"
```

1.1.3. Funciones `R`

`R` contiene una amplia variedad de funciones predefinidas. Entendemos por función un procedimiento que implementa un algoritmo que recibe unos datos de entrada (argumentos), realiza unas transformaciones y devuelve un dato de salida. Los datos de entrada y salida pueden ser opcionales.

Las funciones son tratadas como cualquier otro objeto. Para crearlas utilizamos el comando `function()` el cual crea objetos de tipo `function`, de la siguiente forma:

```r
f <- function (argumentos)
{
  # Código de la función (function body)
  # return (resultados)
}
```

Un ejemplo de función con argumentos sería la siguiente:

```r
area_rectangulo <- function(lado1, lado2)
{
  area <- lado1 * lado2
  print(paste("el area es ",area))
  return(area)
}
```

Podremos calcular el área de rectángulos de distintas dimensiones de la siguiente forma:

```r
area_rectangulo(2,3)
```

```
## [1] "el area es  6"
```

```
## [1] 6
```

```r
area <- area_rectangulo(2,3)
```

```
## [1] "el area es  6"
```

Cuando se hace una llamada a una función, el orden en que se pasan los valores de los argumentos corresponde con la ubicación de estos en la definición de la función. Por ejemplo, en el caso anterior, el valor 2 se pasa a la variable lado1, mientras que el valor 3 se pasa a la variable lado2. Si queremos indicar explícitamente que valor asignar a cada argumento debemos indicar el nombre de este al llamar a la función.

```r
area_rectangulo(lado1 = 2, lado2 = 3)
```

```
## [1] "el area es  6"
```

```
## [1] 6
```

Este último método es el más recomendable cuando las funciones tienen un gran número de argumentos.

Podemos asignar valores por defecto (default value) a las variables en la declaración de las funciones. Por ejemplo:

```r
f <- function (x = NULL, y = NULL)
{
  if (!is.null(x) & !is.null(y))
  {
    print (x+y)
  } else
  {
    print("Faltan valores")
  }
}
```

1.1.4. Instrucciones de control

En `R` existen diferentes instrucciones de control:

- `if()`. Esta instrucción sirve para realizar un conjunto de operaciones si se cumple cierta condición.

```r
if (condicion)
{
  operacion1
  operacion2
  ...
  operación_final
}
```

- `if() {} else {}`. Esta instrucción sirve para realizar un conjunto de operaciones cuando NO se cumple cierta condición evaluada por un `if`.

```r
if (condicion)
{
  operacion1
  operacion2
  ...
  operación_final
}
else
{
  operacion1
  operacion2
  ...
  operación_final
}
```

Cuando hay una sola instrucción para el caso `if` y para el caso `else` , es mejor utilizar

```
ifelse(condicion,operacion_si_cumple, operacion_no_cumple)
```

- `for()`, para repetir un procedimiento cierta cantidad de veces.

```
for (i in secuencia)
{
  operacion1
  operacion2
  ...
  operación_final
}
```

- `while()`, muy útil para repetir un procedimiento siempre que se cumple una condición.

```
while (condicion)
{
  operacion1
  operacion2
  ...
  operación_final
}
```

La instrucción `for()` se usa cuando sabemos el número de veces que se debe repetir el procedimiento, mientras que la instrucción `while` se usa cuando debemos repetir un procedimiento cuando se cumpla una condición.

1.1.5. Vectores en R

La forma más sencilla de crear vectores en `R` es usar la función `c()`:

```
a <- c(1,4,2)
a
```

```
## [1] 1 4 2
```

También podemos usar la función `c()` para concatenar dos vectores:

```
a <- c(1,4,2)
b <- c(5,9,13)
c <- c(a,b)
c
```

```
## [1]  1  4  2  5  9 13
```

Los vectores se pueden utilizar en expresiones aritméticas utilizando operaciones aritméticas y funciones matemáticas y estadísticas. En este caso, los cálculos se realizan por elementos. Por ejemplo:

```
a <- c(1,2,3,4)
b <- c(2,0,1,3)
2*a*b
```

```
## [1]  4  0  6 24
```

Si se utilizan vectores de diferente longitud en una expresión aritmética, los vectores más cortos se repiten ("reciclan") hasta que coinciden con la longitud del vector más largo. Aquí el vector `b` tuvo que ser "reciclado" una vez:

```
a <- c(1,2,3,4)
b <- c(2,0)
a*b
```

```
## [1] 2 0 6 0
```

Si la longitud de los vectores más largos no es múltiplo de la longitud de los vectores más cortos, `R` producirá una advertencia:

```
a <- c(1,2,3,4)
b <- c(2,0,1)
a*b
```

```
## Warning in a * b: longitud de objeto mayor no es múltiplo de la longitud de uno
## menor
```

```
## [1] 2 0 3 8
```

Se utilizan corchetes para acceder a un solo elemento de un vector, `x[i]` devuelve el i-ésimo elemento del vector `x`. Usando el vector `b` anterior:

```
b[2]
```

```
## [1] 0
```

Y también se puede utilizar para cambiar los elementos del vector

```
b[3]<- 7
b
```

```
## [1] 2 0 7
```

Los corchetes también se pueden utilizar para crear subconjuntos de un vector. Hay tres formas de especificar subconjuntos de vectores:

1. Utilizar un vector que especifique los índices que se devolverán:

   ```
   a <- c(1,4,9,16)
   a[c(1,2,3)]
   ```

   ```
   ## [1] 1 4 9
   ```

2. Utilizar un vector que especifique los índices que se eliminarán (como números negativos):

   ```
   a[-4]
   ```

   ```
   ## [1] 1 4 9
   ```

3. Utilizar un vector lógico que especifique los elementos que se devolverán:

   ```
   a[c(TRUE,TRUE,TRUE,FALSE)]
   ```

   ```
   ## [1] 1 4 9
   ```

La última forma se puede usar cuando queremos un subconjunto de un vector en función de sus valores. Supongamos que queremos los elementos de `a` que son divisibles por 2:

```
a[a%%2==0]
```

```
## [1]  4 16
```

Funciones útiles con vectores

- `min(x)`: para obtener el mínimo de un vector
- `max(x)`: para obtener el máximo de un vector
- `length(x)`: devuelve la longitud (número de elementos) del vector x

- `range(x)`: para obtener el rango de valores de un vector, devuelve el mínimo y el máximo
- `sum(x)`: devuelve la suma de todos los elementos del vector
- `prod(x)`: devuelve el producto de todos los elementos del vector
- `which.min(x)`: nos devuelve la posición en donde está el valor mínimo del vector.
- `which.max(x)`: nos da la posición del valor máximo del vector.
- `numeric(n)`: crea un vector numérico de longitud n (que contiene 0s).
- `unique(x)`: devuelve los elementos distintos de x.
- `rev(x)`: invierte el vector x, es decir, devuelve $(x_n, \ldots, x_1)$.
- `sort(x)`: ordena el vector x.
- `order(x)` devuelve la permutación requerida para ordenar el vector x.

Es importante entender la diferencia entre `sort(x)` y `order(x)`. Consideremos el vector:

```
x <- c(11, 7, 3, 9, 4)
```

Se puede ordenar x usando

```
sort(x)
```

```
## [1]  3  4  7  9 11
```

Entonces, ¿qué hace la función `order(x)`?

```
p <- order (x)
p
```

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

La primera entrada (o elemento) del resultado p es 3. La tercera entrada de x es la más pequeña: si queremos ordenar x tenemos que poner el tercer elemento el primero o, en otras palabras, la primera entrada del vector ordenado sería la tercera entrada de x. La segunda entrada de p es 5, porque la segunda entrada más pequeña de x es su quinto elemento. Podemos obtener el vector ordenado aplicando la permutación obtenida:

```
x[p]
```

```
## [1]  3  4  7  9 11
```

Si queremos ordenar el vector en orden descendente, debemos agregar el argumento `decreasing = TRUE`.

```
sort (x, decreasing = TRUE)
```

```
## [1] 11  9  7  4  3
```

Secuencias y funciones relacionadas

`R` tiene funciones integradas para crear secuencias simples y vectores con patrones.

- El operador `:` se puede utilizar para crear secuencias básicas.

  ```
  2:5
  ```

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

  Si el primer argumento es mayor que el segundo, la secuencia será decreciente.

- `seq(from, to, by = )` crea una secuencia desde from hasta to usando by como incremento.

  ```
  seq (1, 2, by=0.2)
  ```

  ```
  ## [1] 1.0 1.2 1.4 1.6 1.8 2.0
  ```

- `seq(from, to, length.out=n)` crea una secuencia de longitud n desde from hasta to.

  ```
  seq (3, 5, length.out =5)
  ```

  ```
  ## [1] 3.0 3.5 4.0 4.5 5.0
  ```

- `rep(x,times=n)` repite el vector x n veces.

  ```
  rep (1:3, times =3)
  ```

  ```
  ## [1] 1 2 3 1 2 3 1 2 3
  ```

- `rep(x,each=n)` repite cada elemento del vector x n veces.

```
rep (1:3, each =3)
```

```
## [1] 1 1 1 2 2 2 3 3 3
```

1.1.6. Matrices en R

Las matrices son la generalización bidimensional de vectores. La principal diferencia entre un vector y una matriz es que un vector tiene un solo índice, mientras que una matriz tiene dos índices: fila y columna. Internamente, R almacena matrices por columnas, es decir, la matriz

$$A = \begin{pmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{pmatrix}$$

se almacena como `1 2 3 4 5 6 7 8 9`. `R` apila internamente las columnas una encima de la otra, lo que se conoce como "column-major mode". Si tuviéramos almacenada la matriz A en `C` o `Java`, se almacenaría en lo que se llama "row-major mode", es decir, las filas se apilarían una encima de la otra.

Creando matrices

Hay tres formas de crear una matriz en R.

1. El primero consiste en utilizar la representación interna de matrices como vectores. Si queremos crear una matriz

$$B = \begin{pmatrix} 0 & 2 & 9 \\ 7 & 4 & 6 \end{pmatrix}$$

```
B <- matrix (c(0, 7, 2, 4, 9, 6), nrow=2)
```

Alternativamente, se puede especificar el número de columnas usando `ncol = 3`. La función matrix también se puede utilizar para crear matrices "vacías": `matrix(a, nrow, ncol)` crea una matriz $nrow \times ncol$ en la que cada entrada toma valor a. Si el vector se crea por filas, se puede añadir el argumento byrow=TRUE.

2. Otra opción para construir la matriz por filas usando la función `rbind`

```
B <- rbind (c(0, 2, 9),
            c(7, 4, 6))
```

También podemos usar rbind para agregar una fila a una matriz existente.

```
rbind (B, c(1, 2, 9))
```

```
##      [,1] [,2] [,3]
## [1,]    0    2    9
## [2,]    7    4    6
## [3,]    1    2    9
```

3. La tercera opción consiste en usar `cbind` que añade una columna a una matriz y se puede usar para construir una matriz por columnas.

```
cbind (c(0, 7), c(2, 4), c(9, 6))
```

```
##      [,1] [,2] [,3]
## [1,]    0    2    9
## [2,]    7    4    6
```

Sistemas de ecuaciones lineales e inversas de matrices

La función `solve` calcula la matriz inversa. Por ejemplo:

```
C <- B %*%t(B)
C
```

```
##      [,1] [,2]
## [1,]   85   62
## [2,]   62  101
```

```
C.inv <- solve (C)
round(C.inv %*%C)
```

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

```
round(C %*%C.inv)
```

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

La función solve puede usarse también para resolver un sistema de ecuaciones lineales: `solve(A,b)` resuelve el sistema de ecuaciones `Ax = b` para x. Para resover el sistema de ecuaciones:

$$5x_2 + x_3 = 7$$
$$7x_2 - x_3 = 5$$
$$11x_1 + x_2 + x_3 = 14$$

se puede usar:

```
A <- rbind (c( 0, 5, 1),c( 0, 7, -1),c(11, 1, 1))
b <- c(7, 5, 14)
x <- solve (A, b)
```

Podemos verificar la respuesta calculando `Ax`, que debería ser `b`.

```
A%*%x
```

```
##      [,1]
## [1,]    7
## [2,]    5
## [3,]   14
```

También se podría calcular primero la inversa de A y luego calcular $A^{-1}b$, es decir, usar

```
x <- solve (A) %*%b
```

1.1.7. Operadores Lógicos

Una variable lógica (o booleana) solo puede contener los dos valores `TRUE` y `FALSE`.

Operadores binarios:

R tiene los siguientes tres operadores binarios `!` (negación), `&` (Y lógico) y `|` (O lógico) .

		Negación	Logical AND	Logical OR
expr1	expr2	!expr1	expr1 & expr2	expr1 | expr2
TRUE	TRUE	FALSE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE
FALSE	TRUE	TRUE	FALSE	TRUE
FALSE	FALSE	TRUE	FALSE	FALSE

Por ejemplo:

```
a <- TRUE
b <- FALSE
c <- a & !b
c
```

```
## [1] TRUE
```

Operadores de comparación:

Los operadores de comparación en `R` son `==` (prueba de igualdad exacta),`!=` ("No es igual") y los operadores `<`, `<=`, `>`, `>=`. Los operadores de comparación devuelven un valor lógico (es decir, `TRUE` o `FALSE`), por lo que se pueden utilizar los operadores `!`, `&` y `|` para combinarlos.

En el siguiente ejemplo, `outside.unit.interval` es TRUE si x es menor que 0 o mayor que 1:

```
x <- 1.3
outside.unit.interval <- (x<0) | (x>1)
outside.unit.interval
```

```
## [1] TRUE
```

1.1.8. Otros tipos de datos en `R`

Cadenas de caracteres

Las cadenas de caracteres en R se definen mediante comillas simples o dobles.

```
string <- "Una cadena entre comillas dobles"
another.string <- 'Esta vez definido usando comillas simples'
```

Se puede insertar una nueva línea en una cadena usando `\n`.

```
string.with.newline <- 'Texto \n Texto en una nueva linea'
cat(string.with.newline)
```

```
## Texto
##  Texto en una nueva linea
```

Las comillas y la barra invertida (backslash) deben llevar delante `"\"` cuando se usan dentro de cadenas.

```
string.with.escapes <- "Esto \" es una comilla y esto \\ es una barra invertida"
cat(string.with.escapes)
```

```
## Esto " es una comilla y esto \ es una barra invertida
```

Factores

Los factores están diseñados para usarse con variables categóricas. La principal diferencia entre un factor y uns cadena de caracteres es que los factores solo pueden tomar un conjunto predefinido de valores (niveles). Cualquier vector se puede convertir en un factor utilizando la función `factor()`, que tiene como argumentos adicionales (opcionales) los niveles (`levels`) y etiquetas (`labels`) que se pueden usar para configurar las etiquetas impresas cuando se muestra el vector.

```
x <- c(1, 4, 2, 4, 1, 3, 1, 2, 4)
X <- factor (x, levels=1:5, labels =c("one", "two", "three", "four", "five"))
X
```

```
## [1] one   four  two   four  one   three one   two   four
## Levels: one two three four five
```

En este ejemplo, el vector X solo puede tomar los valores "one", "two", "three", "four" y "five" (el nivel "five", no se está utilizando). Así podemos poner, por ejemplo:

```
X[1] <- "five"
X
```

```
## [1] five  four  two   four  one   three one   two   four
## Levels: one two three four five
```

Pero no podemos establecer la primera entrada en "six", ya que no está en el conjunto de etiquetas permitidas.

```
X[1] <- "six"
```

```
## Warning in `[<-.factor`(`*tmp*`, 1, value = "six"): invalid factor level, NA
## generated
```

```
X
```

```
## [1] <NA>  four  two   four  one   three one   two   four
## Levels: one two three four five
```

Para poder establecer la primera entrada en "six", primero debemos incluirla en el conjunto de niveles.

```
levels (X) <- c(levels (X), "six")
X[1] <- "six"
X
```

```
## [1] six   four  two   four  one   three one   two   four
## Levels: one two three four five six
```

1.2. Manejo de Datos Básico

1.2.1. Listas

Las listas son los objetos de R que pueden contener elementos de diferentes tipos como: números, cadenas, vectores u otra lista dentro. Una lista también puede contener una matriz o una función como elementos. La lista se crea usando la función `list()`

```
ejemplo1 <- list(1:3,c(TRUE,FALSE),7)
ejemplo1
```

```
## [[1]]
## [1] 1 2 3
##
## [[2]]
## [1]  TRUE FALSE
##
## [[3]]
## [1] 7
```

Las listas pueden anidar unas dentro de otras:

```
ejemplo2 <- list(list(pi,1:2),list("texto",1:3))
ejemplo2
```

```
## [[1]]
## [[1]][[1]]
## [1] 3.141593
##
## [[1]][[2]]
## [1] 1 2
##
##
## [[2]]
## [[2]][[1]]
```

```
## [1] "texto"
##
## [[2]][[2]]
## [1] 1 2 3
```

Al igual que los vectores, las listas se pueden concatenar usando la función `c()` . Una buena práctica es nombrar los elementos de una lista (si es posible). Esto se puede hacer usando la función `names()`:

```
names(ejemplo1) <- c("a","b","c")
ejemplo1
```

```
## $a
## [1] 1 2 3
##
## $b
## [1]  TRUE FALSE
##
## $c
## [1] 7
```

Accediendo a elementos

Se puede acceder a los elementos de una lista utilizando corchetes dobles o, si la lista tiene nombres, `$` .

```
ejemplo1[[1]]
```

```
## [1] 1 2 3
```

```
ejemplo1[["a"]]
```

```
## [1] 1 2 3
```

```
ejemplo1$a
```

```
## [1] 1 2 3
```

1.2.2. Data Frame

Los `data frame` se pueden entender como una versión más flexible de una matriz. En una matriz todas las celdas deben contener datos del mismo tipo, mientras que las filas de un `data frame` admiten datos de distintos tipos, pero sus columnas conservan la restricción de contener datos de un sólo tipo.

En términos generales, los renglones en un `data frame` representan casos, individuos u observaciones, mientras que las columnas representan atributos, rasgos o variables.

Consideremos una matriz que contiene la edad, el peso y la altura de dos niños:

```
datos <- rbind(c(4,15,101),c(11,28,132))
colnames(datos) <- c("Edad","Peso","Altura")
rownames(datos) <- c("A","B")
datos
```

```
##   Edad Peso Altura
## A    4   15    101
## B   11   28    132
```

Intentemos ahora agregar una columna con el género:

```
datos2 <- cbind(datos, Genero=c("f","m"))
datos2
```

```
##   Edad Peso Altura Genero
## A "4"  "15" "101"  "f"
## B "11" "28" "132"  "m"
```

Al añadir la variable género, que es un vector de caracteres, como todos los datos de la matriz deben ser del mismo tipo, `R` ha convertido todas las columnas, incluidad la edad, en cadenas de caracteres.

Una matriz se puede convertir en un `data.frame` usando `as.data.frame()` y se pueden convertir de nuevo a una matriz usando `as.matrix()`.

```
datos <- as.data.frame(datos2)
datos
```

```
##   Edad Peso Altura Genero
## A    4   15    101      f
## B   11   28    132      m
```

Los `data frame` se pueden crear usando la función `data.frame()`, por lo que podríamos haber creado el conjunto de datos usando:

```
datos <- data.frame (Edad =c(4,11), Peso =c(15,28), Altura =c(101,132),
Genero =c("f", "m"))
rownames (datos) <- c("A", "B")
datos <- rbind (A = data.frame ( Edad =4, Peso =15, Altura =101, Genero ="f"),
B= data.frame ( Edad =11, Peso =28, Altura =132, Genero ="m"))
```

Los `data frame` no solo se comportan como matrices, también se comportan como listas donde las columnas son las entradas. Hay muchas formas para acceder a las columnas:

```
datos$Edad
```

```
## [1]  4 11
```

```
datos[,"Edad"]
```

```
## [1]  4 11
```

```
datos[,1]
```

```
## [1]  4 11
```

```
datos[["Edad"]]
```

```
## [1]  4 11
```

```
datos[[1]]
```

```
## [1]  4 11
```

Ahora veremos varias funciones R para manipular `data frame`. Vamos a trabajar con un conjunto de datos llamado `CoronavirusAbril`.

```
load("Coronavirus.Rdata")
CoronaVirus <- as.data.frame(CoronaVirusAbril)
```

El conjunto de datos contiene datos sobre la incidencia del coronavirus en el mes de abril de 2021 en las distintas provincias de la Comunidad Autónoma de Castilla y León. Supongamos que queremos añadir una nueva columna que sea el porcentaje de contagiados.

```
CoronaVirus <-
  cbind(CoronaVirus,porcentaje =
CoronaVirus[,"casos_confirmados"]/CoronaVirus[,"Poblacion"])*100
```

```
CoronaVirus[,"porcentaje"]<-
CoronaVirus[,"casos_confirmados"]/CoronaVirus[,"Poblacion"]*100
```

```
CoronaVirus$porcentaje <-
CoronaVirus[,"casos_confirmados"]/CoronaVirus[,"Poblacion"]*100
```

y se obtiene

```
##            casos_confirmados nuevos_positivos altas fallecimientos Poblacion
## Avila                   1831               26   602            131    158966
## Burgos                  2611               23   866            204    355420
## Leon                    3420               91  1528            392    460415
## Palencia                1130                3   315             80    160280
## Salamanca               3946              154  1119            346    331382
## Segovia                 3271               78   833            198    154084
## Soria                   2208               59   380            118     89612
## Valladolid              4208               63  1480            346    520197
## Zamora                   812               14   306             82    172522
##            porcentaje
## Avila       1.1518186
## Burgos      0.7346238
## Leon        0.7428081
## Palencia    0.7050162
## Salamanca   1.1907708
## Segovia     2.1228680
## Soria       2.4639557
## Valladolid  0.8089243
## Zamora      0.4706646
```

Generalmente, es mejor usar la funcion `transform()`. Su principal ventaja es que no necesitamos poner CoronaVirusAbril$ en todas partes.

```
CoronaVirus <-
transform(CoronaVirus,porcentaje=casos_confirmados/Poblacion*100)
```

También podemos usar las mismas técnicas que en las matrices para eliminar columnas. Por ejemplo:

```
CoronaVirus <- CoronaVirus[,-6]
```

Alternativamente se puede usar:

```
CoronaVirus [[6]] <- NULL
CoronaVirus$porcentaje <- NULL
```

```
CoronaVirus
```

```
##            casos_confirmados nuevos_positivos altas fallecimientos Poblacion
## Avila                   1831               26   602            131    158966
## Burgos                  2611               23   866            204    355420
## Leon                    3420               91  1528            392    460415
## Palencia                1130                3   315             80    160280
## Salamanca               3946              154  1119            346    331382
## Segovia                 3271               78   833            198    154084
## Soria                   2208               59   380            118     89612
## Valladolid              4208               63  1480            346    520197
## Zamora                   812               14   306             82    172522
```

También se pueden obtener subconjuntos como con matrices. Por ejemplo, para eliminar las provincias con menos de 100 fallecidos podemos usar:

```
Altaincidencia <- CoronaVirus[CoronaVirus$fallecimientos > 100,]
```

Es importante señalar que por cada `NA` en la condición, `R` colocará una fila de `NA` en la parte superior del conjunto de datos. Luego si esto ocurriera tendriamos que eliminar la(s) fila(s) con valores perdidos, lo cual se hace mejor usando la función `na.omit()`.

También podemos crear el subconjunto de datos de una sola vez usando la función `subset()`.

```
Altaincidencia <- subset(CoronaVirus,fallecimientos > 100)
Altaincidencia
```

```
##            casos_confirmados nuevos_positivos altas fallecimientos Poblacion
## Avila                   1831               26   602            131    158966
## Burgos                  2611               23   866            204    355420
## Leon                    3420               91  1528            392    460415
## Salamanca               3946              154  1119            346    331382
## Segovia                 3271               78   833            198    154084
## Soria                   2208               59   380            118     89612
## Valladolid              4208               63  1480            346    520197
```

1.3. Gráficos

`R` dispone de distintos sistemas gráficos siendo los más comunes:

- Sistema gráfico base de R. Se encuentra en los paquetes `graphics` (contiene las funciones de gráficos base: plot, hist, etc) y `grDevices` (implementa los distintos dispositivos gráficos: pdf, ps, png,...). Podemos tener una idea del funcionamiento del paquete `graphics` mediante:

```
demo(graphics)
```

- `ggplot2` . Un manual de uso del mismo puede encontrarse en: https://link.springer.com/book/10.1007/978-3-319-24277-4.

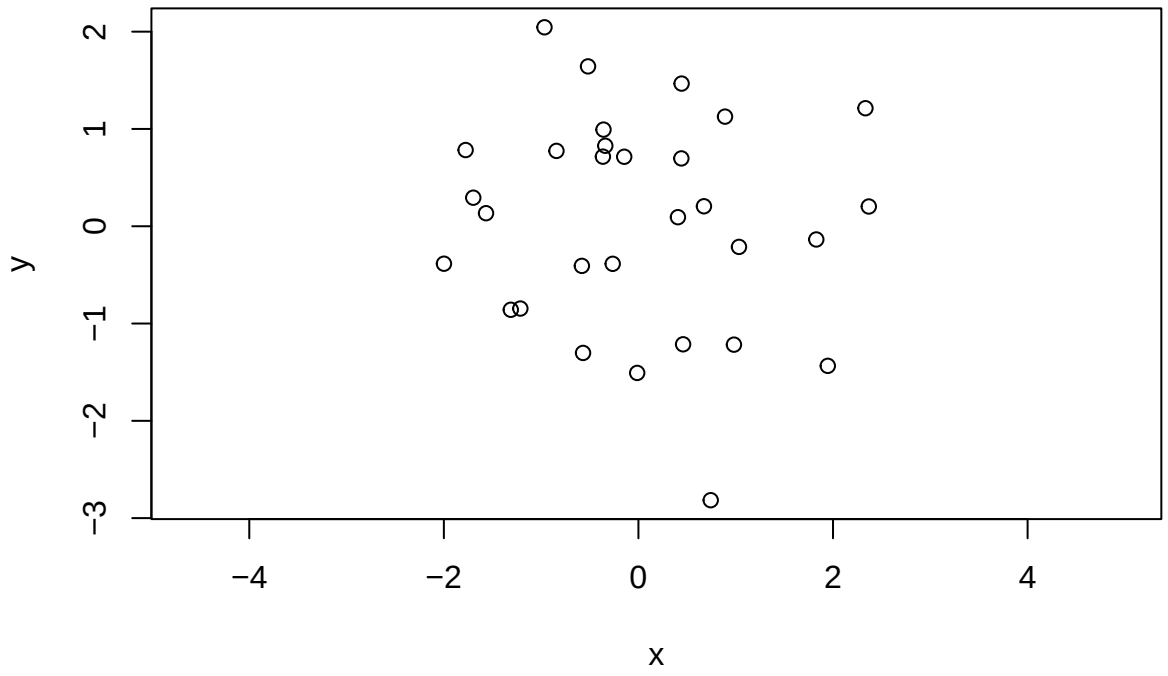
1.3.1. La función `plot()`

La función `plot()` se puede utilizar para crear gráficos de dispersión. Los siguientes cuatro comandos crean el mismo gráfico de dispersión de y frente a x:

- `plot(x,y,...)` donde x e y son vectores que contienen las coordenadas.
- `plot(y~x,...)` donde x e y son vectores que contienen las coordenadas ó `plot(y~x,data=data,...)` donde x e y son columnas de `data`.
- `plot(A,...)`donde A es una matriz de dos columnas (coordenadas x e y). Si M tiene más de dos columnas, estas serán ignoradas.
- `plot(lst,...)` donde lst es una lista con un elemento x y un elemento y.

Veamos un ejemplo:

```
n <- 30
x <- rnorm(n)
y <- rnorm(n)
plot(x,y,asp=1)
```



Podríamos obtener el mismo resultado con:

```
plot(y~x)
```

o con:

```
data <- cbind(x=x, y=y)
plot(data)
```

o con:

```
plot(y~x, data = data)
```

Además de `plot` existen otras funciones tales como `hist, barplot, boxplot` , ... que permiten realizar otros tipos de gráficos.

1.3.2. La librería ggplot2

Cuando usamos `ggplot2` para hacer gráficos, en realidad lo que hacemos es crear un contenedor, al que le tendremos que incorporar la información de:

- los datos que vamos a usar
- los ejes que se van a representar
- y una capa por cada gráfico que se quiera dibujar en esos ejes.

Así, para cada diagrama que se dibuje con `ggplot2` debemos definir al menos:

- `data=` es obligatoriamente un conjunto de datos o "data.frame"
- `aes()` es la configuración de los ejes
- y la capa de gráficos propiamente dicha

En primer lugar tenemos que instalar y cargar el paquete:

```
# Instala la librería
install.packages("ggplot2")
```

```
# Carga la librería
library(ggplot2)
```

Vamos a utilizar `ggplot2` con uno de los conjuntos de datos integrados en `R` que es `iris`.

```
data(iris)
head(iris)
```

```
##   Sepal.Length Sepal.Width Petal.Length Petal.Width Species
## 1          5.1         3.5          1.4         0.2  setosa
## 2          4.9         3.0          1.4         0.2  setosa
## 3          4.7         3.2          1.3         0.2  setosa
## 4          4.6         3.1          1.5         0.2  setosa
## 5          5.0         3.6          1.4         0.2  setosa
## 6          5.4         3.9          1.7         0.4  setosa
```

```
p <- ggplot(iris)
```

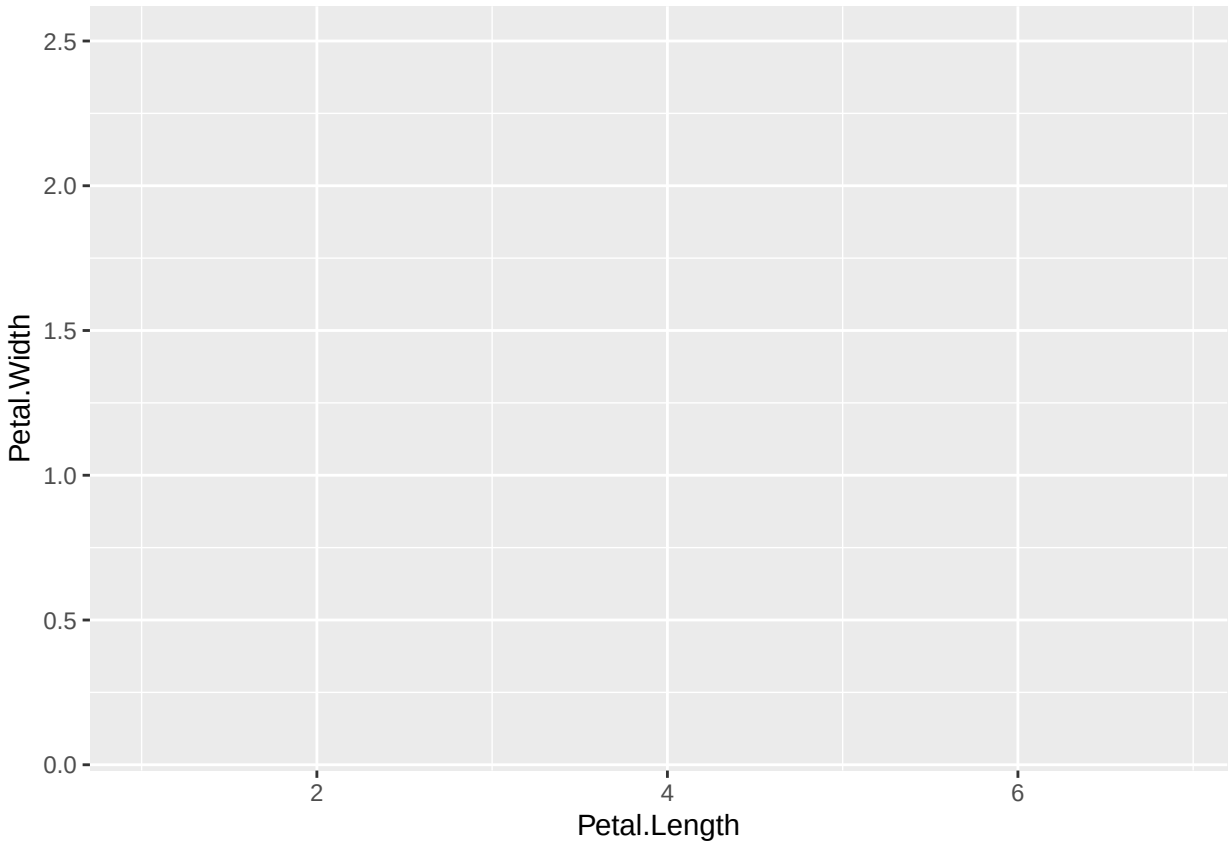
El código anterior crea un objeto, p que es un objeto que solo contiene los datos que vamos a utilizar y al que hay que añadir qué queremos hacer con ellos.

Podemos añadirle una etiqueta a los ejes e indicar que utilice colores distintos para cada especie.

```
p <- p + aes(x = Petal.Length, y = Petal.Width, colour = Species)
```

Podríamos haber combinado ambos pasos en una única expresión:

```
p <- ggplot(iris, aes(x = Petal.Length, y = Petal.Width, colour = Species))
p
```



Como se puede observar, hemos creado el "contendor" del gráfico, con sus ejes, y etiqueta los ejes. Pero falta lo principal, ¿dónde están los puntos?¿o las líneas?¿o el área?. La respuesta es que aún no la hemos dibujado. Está todo preparado pero falta pedir que tipo de gráfico queremos hacer.

```
p <- p + geom_point()
p
```

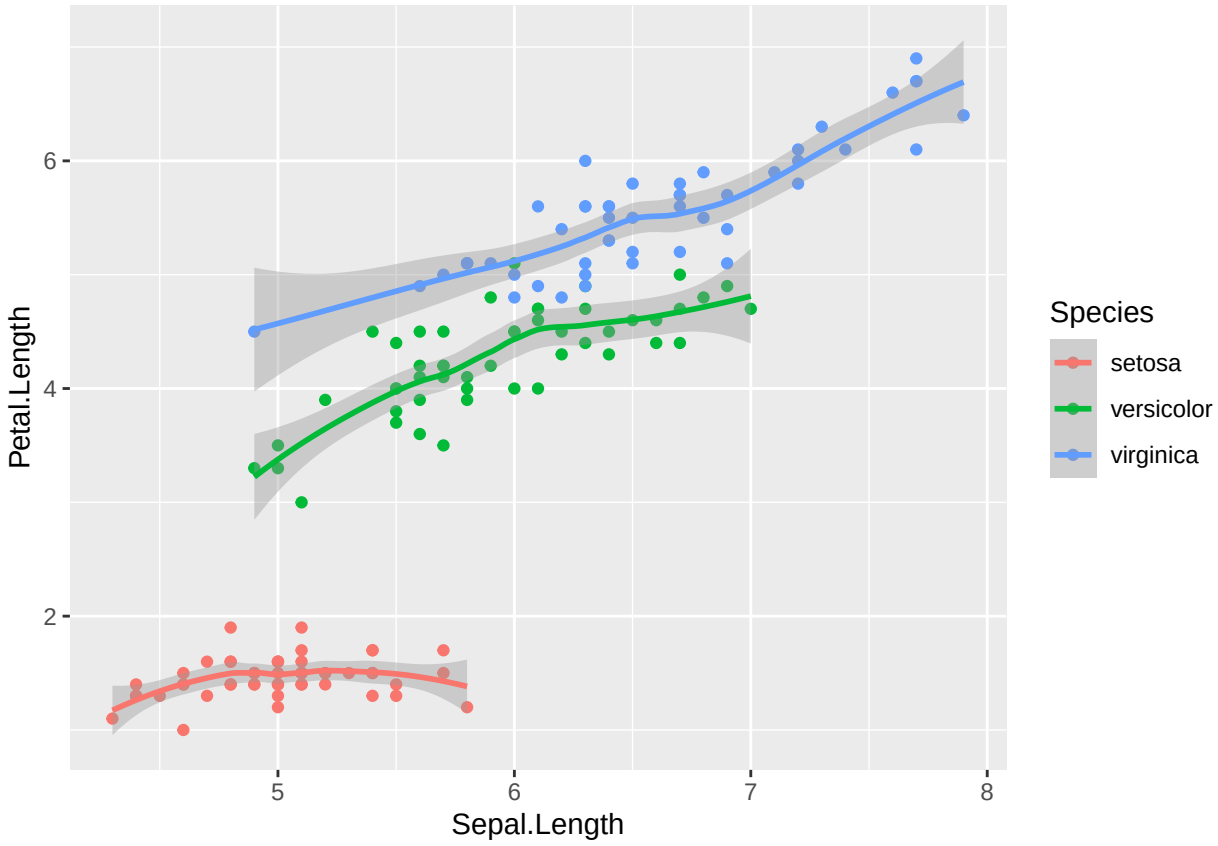
Podríamos obtener el mismo resultado en una única línea:

```
ggplot(data=iris, aes(x = Petal.Length, y = Petal.Width, colour = Species)) +
geom_point() # los puntos
```

Podrían seguir superponiéndose capas:

```
# Creamos el contenedor de ggplot
ggplot(data=iris, aes(Sepal.Length, Petal.Length,color=Species)) +
geom_point() + # los puntos
stat_smooth() # líneas y bandas de suavizado (smooth)
```

```
## `geom_smooth()` using method = 'loess' and formula = 'y ~ x'
```



Los tipos de capas más utilizadas con `ggplot` son:

Nombre	Descripción
`geom_point`	Puntos
`geom_line`	Líneas
`geom_abline`	Líneas de referencia
`geom_rect`	Rectángulo
`geom_bar`	Bar plot
`geom_histogram`	Histograma
`geom_ribbon`	Sombreado

2. Programación Lineal

La programación lineal es una clase de modelos de programación matemática donde todas las funciones que aparecen en el modelo son lineales. Se desarrolló a partir de la Segunda Guerra Mundial para resolver problemas de asignación de recursos. Posteriormente su aplicación ha sido mucha lo cual ha llevado a que los modelos de optimización lineal sean una de las herramientas básicas más utilizadas de la Investigación Operativa.

Formulación de modelos

El proceso de formulación del problema real a estudio es el primer paso, dependiendo del modelo que se construya se tendrá una u otra solución para el problema, por lo que una modelización errónea puede derivar en grandes pérdidas.

A la hora de formular un problema, lo primero es determinar las **variables de decisión,** x_j que son los factores sujetos a cambios cuyos valores pueden variar, o al menos hacerlo dentro de unos límites. El objetivo será encontrar una solución óptima $x = (x_1, \ldots, x_n)$ para estas variables cumpliendo las condiciones del problema. En segundo lugar se formulan las **restricciones** que se expresan como ecuaciones o inecuaciones lineales en las variables de decisión y que representan la limitación en la disponibilidad de algún recurso:

$$g_i \leq b_i$$

$$g_i \geq b_i$$

$$g_i = b_i$$

Existen también las restricciones de signo de las variables, $(x_j \geq 0)$, que en muchos casos se evitan en la formulación porque se suponen obvias pero que deben de tenerse en cuenta. Para terminar la modelización es necesario formular la **función objetivo** a optimizar, que será una función lineal en las variables y que generalmente representa el objetivo de maximizar un beneficio o minimizar un coste:

$$\text{máx}\, z = f(x)$$

$$\text{mín}\, z = f(x)$$

2.1. Solución gráfica de problemas bidimensionales

En esta sección veremos cómo resolver los problemas más elementales de programación lineal mediante su representación gráfica. Haremos uso de la librería de `R ggplot2` .

2.1.1. Ejemplo

Resolver gráficamente:

$$\text{máx}\, 3x_1 + x_2$$

$$2x_1 + x_2 \leq 6$$

$$x_1 + 3x_2 \leq 9$$

$$x_i \geq 0, i = 1, 2$$

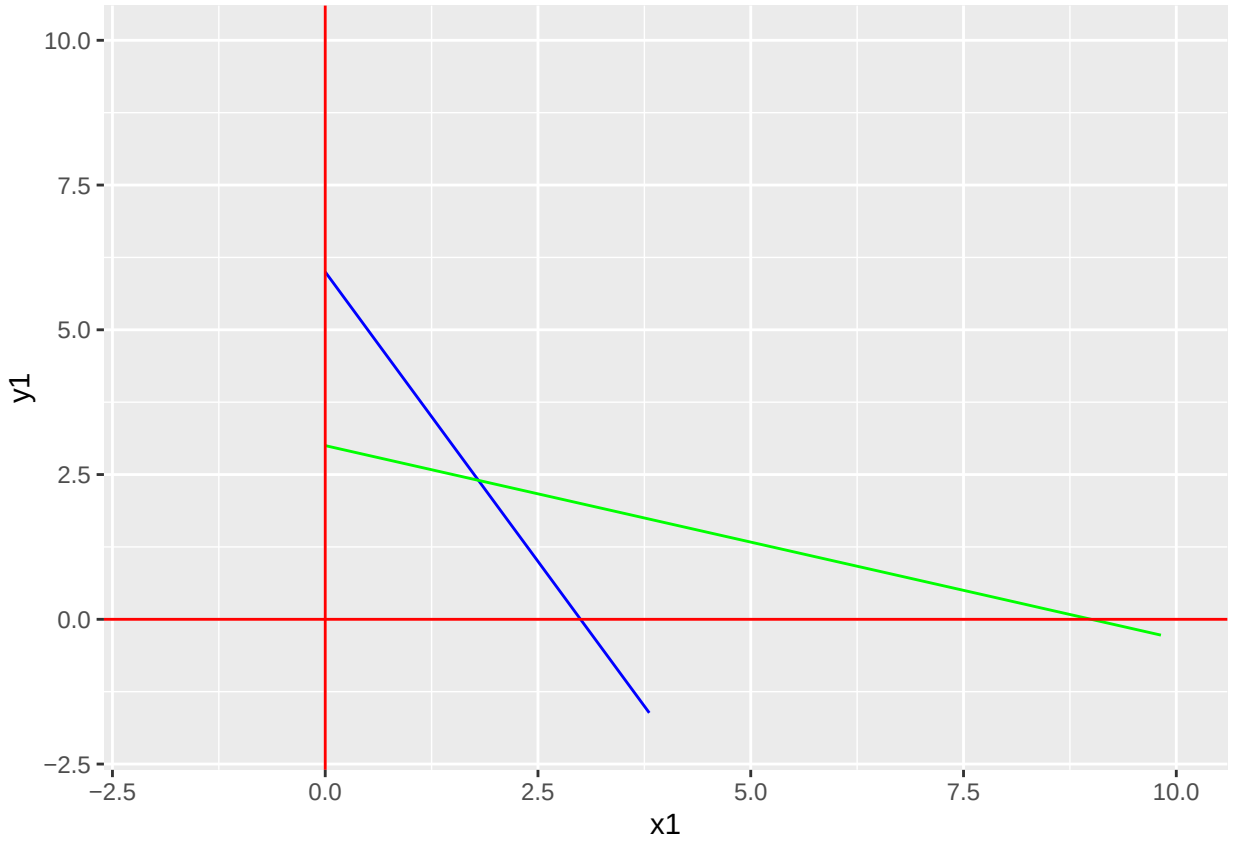
Para encontrar la solución de un PL con dos variables, se dibuja un sistema de coordenadas cartesianas en el que cada variable está representada por un eje. Sobre los ejes anteriores

se representan las restricciones del problema (incluyendo las de no negatividad), teniendo en cuenta que si una restricción es una inecuación define una región que será uno de los semiplanos limitados por la línea recta que se tiene al considerar la restricción como una igualdad. Así, dibujamos las rectas:

- $R1 : 2x_1 + x_2 = 6$
- $R2 : x_1 + 3x_2 = 9$

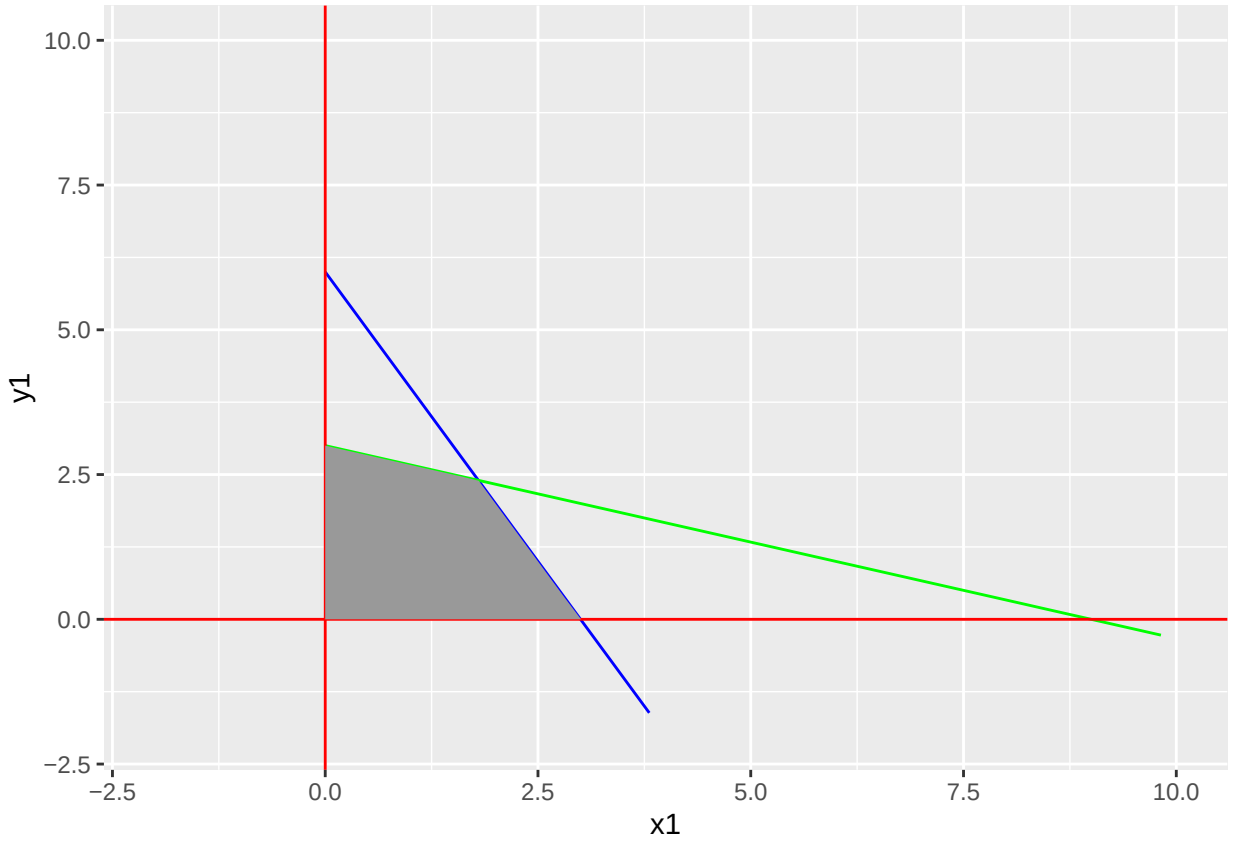
Para dibujar las restricciones despejamos una variable en función de la otra. En este caso, despejamos x_2.

```
library(ggplot2)
R1 <- function (x) 6 - 2*x
R2 <- function (x) 3 - x/3
x1<- seq(0,100,length.out=500)
datos <- data.frame (x1, y1 = R1(x1), y2 = R2(x1))
p <- ggplot(datos, aes (x = x1)) +
  geom_line(aes(y = y1), colour = "blue") +
  geom_line(aes(y = y2), colour = "green") +
  geom_hline(yintercept = 0, colour ="red") +
  geom_vline(xintercept = 0, colour = "red") +
  ylim(-2,10)+
  xlim(-2,10)
  p
```



Ahora pintamos la **región factible**, que viene dada por la intersección de las dos regiones

```
datos <-  transform(datos, z = pmax(0,pmin(y1,y2)))
p +  geom_ribbon(data=datos,aes(ymin=rep(0,length(z)),ymax = z), fill = 'gray60')
```



A continuación se tienen que determinar los puntos extremos de la región factible que son candidatos a solución óptima:

- $(0, 3)$ Intersección R2 (línea verde) con el eje de ordenadas.
- $(3, 0)$ Intersección R1 (línea azul) con el eje de abscisas.
- (a, b) Intersección de R1 y R2

La intersección de R1 y R2 se puede calcular resolviendo el sistema:

$$\begin{bmatrix} 2 & 1 \\ 1 & 3 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 6 \\ 9 \end{bmatrix}$$

```
A <- rbind(c(2,1), c(1,3))
b <- c(6,9)
pto <- solve(A,b)
pto
```

```
## [1] 1.8 2.4
```

Por último, se evalúa la función objetivo en estos puntos y aquel o aquellos que la maximicen determinarán la solución óptima del problema.

```
obj <- function(x) 3*x[1]+x[2]
ptR2 <- c(0,3)
ptR1 <- c(3,0)
obj(ptR2)
```

```
## [1] 3
```

```
obj(ptR1)
```

```
## [1] 9
```

```
obj(pto)
```

```
## [1] 7.8
```

En este caso, la solución óptima es:

$$x_1 = 3$$
$$x_2 = 0$$

y la solución óptima es $Z = 9$.

2.2. Función `simplex()` de la libreria `boot`

Esta función optimiza una función lineal c_x sujeta a restricciones $A_1x \leq b_1$, $A_2x \geq b_2$, $A_3x = b_3$ y/o $x \geq 0$. Sirve para problemas tanto de maximización como de minimización, pero por defecto minimiza.

El método empleado es adecuado solo para sistemas relativamente pequeños. Además, si es posible, el número de restricciones debe reducirse al mínimo para acelerar el tiempo de ejecución, que es aproximadamente proporcional al cubo del número de restricciones. En particular, si existen restricciones de la forma $x_i \geq b_{2i}$, se recomienda omitirla estableciendo $x_i = x_i - b_{2i}$, cambiando todas las restricciones y la función objetivo en consecuencia y luego volver a transformarse después de que se haya encontrado la solución.

Se debe instalar y cargar la libreria previamente.

```
install.packages("boot")
```

```
library(boot)
```

Uso de la función:

- Función: `simplex(a, A1, b1, A2, b2, A3, b3, maxi, n.iter, eps)`
- Argumentos:
  - `a`: coeficientes de la función objetivo
  - A_1: matriz que contiene los coeficientes de las restricciones del tipo $\leq$
  - b_1 : vector de términos independientes de las restricciones de tipo $\leq$ (correspondientes a la matriz A_1). Este argumento es necesario si está A_1, sino se ignora. Todos los valores en b_1 deben ser no negativos.
  - A_2: matriz que contiene los coeficientes de las restricciones del tipo $\geq$
  - b_2: vector de términos independientes correspondientes a la matriz A_2. Este argumento es necesario si está A_2, si no se ignora. Todos los valores en b_2 deben ser no negativos. **Importante: las restricciones $x \geq 0$ se incluyen automáticamente y no se deben introducir aquí.**
  - A_3: matriz que contienes los coeficientes de las restricciones del tipo $=$
  - b_3: vector de términos independientes correspondientes a la matriz A_3. Este argumento es necesario si está A_3, sino se ignora. Todos los varlores en b_3 deben ser no negativos.
  - `maxi`: Un carácter lógico que especifica la minimización si es $FALSE$ (por defecto) y la maximización en casa contrario. Si $maxi = TRUE$ entonces el problema de maximización se convierte en un problema de minimización cambiando los coeficientes de la función objetivo por sus negativos.

  - `n.iter`: número máximo de iteraciones en cada fase (usa el método de las dos fases)
  - `eps`: la precisión decimal a la hora de de comparar valores.

- Devuelve un objeto de la clase simplex, que es una lista con componentes (entre otras):

  - `$soln`: solución óptima, si se alcanza
  - `$solved`: código de terminación:
    - 1 indica solución óptima alcanzada
    - 0 indica que se ha alcanzado el número máximo de iteraciones sin acabar
    - -1 indica que no existe solución factible
  - `$value`: valor de la función objetivo en la solución $*soln*
  - `$val.aux`: valor de las variables artificiales en el óptimo
  - `etc`.

2.2.1. Ejemplo 1: Problema con solución única

Resuelve:

$$\begin{gathered}\text{máx } 2x_1 + 2x_2 + 3x_3 \\ -2x_1 + x_2 + x_3 \leq 1 \\ 4x_1 - x_2 + 3x_3 \leq 3 \\ x_i \geq 0, i = 1, 2, 3\end{gathered}$$

Para resolverlo haciendo uso de la función `simplex()`, las instrucciones serían:

```
coef <- c(2,2,3)
A1 <- rbind (c(-2,1,1),c(4,-1,3))
b1 <- c(1,3)
sol <- simplex(a=coef, A1=A1, b1=b1,maxi="TRUE")
```

Por lo tanto, la solución se encuentra en los valores:

```
sol$soln
```

```
## x1 x2 x3
##  2  5  0
```

```
sol$value
```

```
##  b
## 14
```

$x_1 = 2, x_2 = 5, x_3 = 0$

El valor de la función objetivo es 14

2.2.2. Ejemplo 2: Problema infactible

Resuelve:

$$\begin{gathered} \min x_1 + x_2 \\ x_1 + x_2 \leq 1 \\ 4x_1 + 2x_2 \geq 6 \\ x_i \geq 0, i = 1, 2 \end{gathered}$$

```
a <- c(1,1)
A1 <- c(1,1)
b1 <- 1
A2 <- c(4,2)
b2 <- 6
simplex(a=a, A1=A1, b1=b1, A2=A2, b2=b2)
```

```
##
## Linear Programming Results
##
## Call : simplex(a = a, A1 = A1, b1 = b1, A2 = A2, b2 = b2)
##
## Minimization Problem with Objective Function Coefficients
## x1 x2
##  1  1
##
## No feasible solution could be found
```

El problema anterior no tiene solución factible.

2.2.3. Inconvenientes

1. Prueba a resolver

$$\begin{gathered}\text{máx } 2x_1 + x_2 \\ -x_1 + x_2 \leq 2 \\ x_1 + 2x_2 \leq 6 \\ 2x_1 + x_2 \leq 6 \\ x_i \geq 0, i = 1, 2\end{gathered}$$

```
a<- c(2,1)
A1 <- rbind(c(-1,1),c(1,2),c(2,1))
b1 <- c(2,6,6)
simplex(a=a, A1=A1, b1=b1, maxi = "TRUE")
```

```
##
## Linear Programming Results
##
## Call : simplex(a = a, A1 = A1, b1 = b1, maxi = "TRUE")
##
## Maximization Problem with Objective Function Coefficients
## x1 x2
##  2  1
##
##
## Optimal solution has the following values
## x1 x2
##  3  0
## The optimal value of the objective  function is 6.
```

Aparece como solución $x_1 = 3, x_2 = 0$, con valor de la función objetivo igual a 6. Sin embargo, la función objetivo en el punto $(2, 2)$ también vale 6. Por tanto son soluciones $(2, 2)$ y $(3, 0)$ y, por tanto cualquier punto del segmento que los une es solución.

2. Prueba a resolver

$$\begin{gathered}\text{máx } x_1 + x_2 \\ -x_1 + x_2 \leq 2 \\ x_2 \leq 4 \\ x_i \geq 0, i = 1, 2\end{gathered}$$

R nos devuelve un error. Probemos a resolverlo gráficamente:

```
library(ggplot2)
```

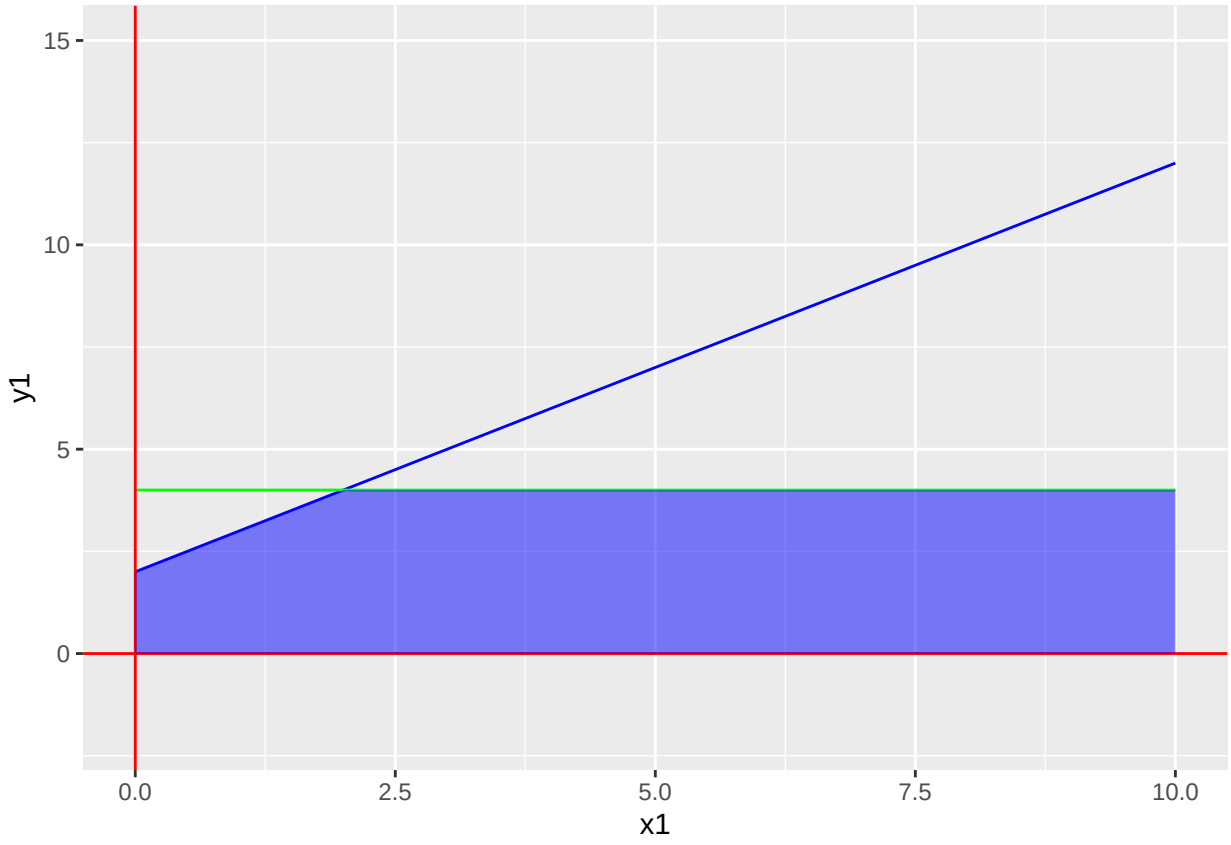
```
R1 <- function (x) 2 + x
R2 <- function (x) 4 + 0*x
```

```
x1<- seq(0,10,length.out=200)
datos <- data.frame (x1, y1 = R1(x1), y2 = R2(x1))
p <- ggplot(datos, aes (x = x1)) +
  geom_line(aes(y = y1), colour = "blue") +
  geom_line(aes(y = y2), colour = "green") +
  geom_hline(yintercept = 0, colour ="red") +
  geom_vline(xintercept = 0, colour = "red") +
  ylim(-2,15)
```

Y rellenamos **la región factible**, tras comprobar los planos definidos por las inecuaciones:

```
datos <-  transform(datos, z = pmin(y1,y2))
```

```
p +  geom_ribbon(data=datos,aes(ymin=0,ymax = z), fill = 'blue',alpha=0.5)
```



El problema tiene solución no acotada.

2.2.4. Conclusión:

La función `simplex()` se encarga de resolver problemas de programación lineal. Es de las funciones más básicas para trabajar con problemas de optimización. Sin embargo, **no trae incorporado análisis de sensibilidad ni permite declarar variables enteras**.

Además **presupone no negatividad**, por lo cual si queremos que nuestras variables puedan tomar valores negativos tenemos que utilizar un artilugio para separar nuestra variable en dos[1] .Sin embargo es muy intuitiva en su operación con matrices.

El inconveniente más grande que tiene es que **es lenta**, como ya se ha mencionado, tarda un tiempo promedio proporcional al cubo de la cantidad de restricciones para hallar la respuesta.

[1]Si x_i puede tomar valores negativos, tenemos que utilizar dos variables, x_{i1} y x_{i2}, relacionándolas mediente $x_i = x_{i1} - x_{i2}$

2.3. Función `lp()` de la libreria `lpSolve`

Se trata de otra función para resolver problemas de Programación Lineal, pero más completa pues admite variables enteras (programación entera) y variables binarias (programación 0-1).

Para usarla se deber instalar y cargar la librería previamente:

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

```
library(lpSolve)
```

Uso de la función:

- Función: `lp(direction="min", objective.in, const.mat, const.dir, const.rhs, transpose.constraints=TRUE, int.vec, binary.vec, all.int=FALSE, all.bin=FALSE)`
- Argumentos más utilizados:
  - `direction`: Variable cadena de caracteres que indica la dirección de optimización: `min` (por defecto) o `max`.
  - `objective.in`: coeficientes de la función objetivo.
  - `const.mat`: matriz con los coeficientes numéricos de las restricciones, una fila por restricción, una columna por variable (a menos que transpose.constraints = FALSE; ver más adelante).
  - `const.dir`: vector de cadenas de caracteres que indica la dirección de las restricciones. Los valores pueden ser '<', '<=', '=', '>' o '>='.
  - `const.rhs`: vector de términos independientes de las restricciones.
  - `transpose.constraints`: por defecto cada restricción ocupa una fila de `const.mat`, y tal matriz necesita trasponerse antes de pasar al código de optimización. Para matrices de restricciones grandes es aconsejable construir la matriz de restricciones por columnas. En tal caso, se debe poner `transpose.constraints=FALSE`.
  - `int.vec`: Vector numérico que da los índices de las variables que deben ser enteras. La longitud de este vector debe ser igual al número de variables enteras.
  - `binary.vec`: Vector numérico como int.vec que da los índices de las variables que deben ser binarias.
  - `all.int`: Carácter lógico que responde a la pregunta ¿son todas la variables enteras? Por defecto toma el valor `FALSE`.
  - `all.bin`: Carácter lógico que responde a la pregunta ¿son todas la variables binarias? Por defecto toma el valor `FALSE`.
- Devuelve: un objeto de la clase lp, que es una lista con componentes (entre otros):

- `$direction`: La dirección de optimización del problema.
- `$x.count`: Número de variables en la función objetivo.
- `$objective`: Vector con los coeficientes de la función objetivo.
- `$const.count`: Número de restricciones.
- `$int.count`: Número de variables enteras.
- `$int.vec`: Vector con los índices de las variables enteras.
- `$bin.count`: Número de variables binarias.
- `$binary.vec`: Vector con los índices de las variables binarias
- `$objval`: valor de la función objetivo en el óptimo.
- `$solution`: solución óptima.
- `$status`: Número que indica
  - 0 = éxito
  - 2 = solución no factible
  - 3 = solución no acotada

2.3.1. Ejemplo

Con 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 ($\leq$;$\geq$;=). Este paquete es un poco más sencillo. Resolvamos el problema:

$$\begin{gathered}\text{máx}\, x_1 + 9x_2 + 3x_3 \\ x_1 + 2x_2 + 3x_3 \geq 1 \\ 3x_1 + 2x_2 + 2x_3 \leq 15 \\ x_i \geq 0, i = 1, 2, 3\end{gathered}$$

Primero se definen los coeficientes de la función objetivo

```
coef <- c(1,9,3)
```

Las restricciones se guardan en tres objetos: una única matriz con los coeficientes de todas las restricciones, un vector con los signos de las restricciones (dirección) y otro vector con los términos independientes:

```
A <- rbind (c(1, 2, 3), c(3, 2, 2))
b <- c(1,15)
dir <- c(">=", "<=")
```

Por último, se maximiza la función objetivo

```
lp("max", coef , A, dir , b)
```

```
## 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.

Los resultados se pueden guardar en un objeto para obtener más información.

```
sol <- lp("max", coef , A, dir , b)
sol$solution
```

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

```
sol$objval
```

```
## [1] 67.5
```

```
sol$status
```

```
## [1] 0
```

Aparece como solución $x_1 = 0$, $x_2 = 7.5$ y $x_3 = 0$

2.3.2. Inconvenientes

- Al igual que la función `simplex()`, `lp()` no identifica cuando un problema tiene **óptimos alternativos:**

$$\begin{aligned} &\text{máx } 2x_1 + x_2 \\ &-x_1 + x_2 \leq 2 \\ &x_1 + 2x_2 \leq 6 \\ &2x_1 + x_2 \leq 6 \\ &x_i \geq 0, i = 1, 2 \end{aligned}$$

```
coef <- c(2,1)
A <- rbind(c(-1,2),c(1,2),c(2,1))
b <- c(2,6,6)
dir <- rep("<=",3)
sol <- lp("max",coef,A,dir,b)
sol$objval
```

```
## [1] 6
```

```
sol$solution
```

```
## [1] 3 0
```

```
sol$status
```

```
## [1] 0
```

Sol soluciones $(2, 2)$ y $(3, 0)$ y, por tanto, cualquier punto del segmento que los une, pero vuelve a dar únicamente la solución $(3, 0)$.

- En este caso los problemas **no acotados** devuelven un "error", indican `status 3`.

$$\begin{gathered} \text{máx } x_1 + x_2 \\ -x_1 + x_2 \leq 2 \\ x_2 \leq 4 \\ x_i \geq 0, i = 1, 2 \end{gathered}$$

```
coef <- c(1,1)
A <- rbind(c(-1,1),c(0,1))
b <- c(2,4)
dir <- rep("<=",2)
sol <- lp("max",coef,A,dir,b)
sol
```

```
## Error: status 3
```

2.4. Programación Entera

Veamos, a partir de los datos del Ejemplo 2.3.1, que de las tres variables, la segunda y la tercera deben ser enteras. Con el parámetro `int.vec` puedo pasarle a `lp()` un vector con los índices de las variables enteras.

```
coef <- c(1,9,3)
A <- matrix(c(1,2,3,3,2,2),byrow = TRUE,nrow = 2)
b <- c(1,15)
dir <- c(">=","<=")
sol <- lp("max", coef, A, dir, b, int.vec=c(2:3))
sol$solution
```

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

La solución ahora es: $x_1 = 0.333$, $x_2 = 7$ y $x_3 = 0$

Además:

```
sol$int.count # Devuelve la cantidad de variables enteras
```

```
## [1] 2
```

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

```
## [1] 2 3
```

```
sol$solution[sol$int.vec] # Devuelve los valores de las variables enteras
```

```
## [1] 7 0
```

Por último, imaginemos que la primera variable es **binaria** (solo puede tomar los valores 0 y 1). Entonces, podemos usar el parámetro `binary.vec` de la misma forma que `int.vec`, indicando los índices de las variables binarias:

```
sol<-lp("max",coef,A,dir,b,int.vec=c(2:3),binary.vec=c(1))
sol
```

```
## Success: the objective function is 63
```

```
sol$solution
```

```
## [1] 0 7 0
```

```
sol$bin.count
```

```
## [1] 1
```

```
sol$binary.vec
```

```
## [1] 1
```

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

```
## [1] 0
```

Cabe destacar que en ningún lado pusimos ninguna restricción de no negatividad sin embargo `lpSolve` también **asume que todas las variables son no negativas por defecto**. Para tener una variable libre, es necesario hacer la transformación ya mencionada: $x_i = x_{i1} - x_{i2}$, donde x_{i1} y x_{i2} son no negativas.

Supongamos ahora que la segunda variable ya no tiene que ser entera pero puede tomar valores negativos. Primero, aplica la transformación $x_2 = x_{21} - x_{22}$ en la función objetivo, $9x_2 = 9(x_{21} - x_{22}) = 9x_{21} - 9x_{22}$.

```
coef <- c(1,9,-9,3)
```

Y aplicamos el mismo principio en los coeficientes de las restricciones

```
A <- matrix(c(1,2,-2,3,3,2,-2,2),byrow=TRUE,nrow = 2)
```

Los vectores de dirección de las restricciones y términos independientes no necesitan modificación, puesto que la transformación no les afecta.

```
sol <- lp("max", coef,A,dir,b)
sol
```

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

En este caso, la solución óptima sigue siendo cuando la variable toma el valor positivo 7.5, no cambia. Veamos qué pasa si volvemos a añadir la condición de que sea entera. En este caso, habrá que añadir los índices para x_{21}, x_{22} y x_3.

```
sol <- lp("max",coef,A,dir,b,int.vec=c(2:4))
sol$solution
```

```
## [1]  0.3333333 56.0000000 49.0000000  0.0000000
```

```
sol$solution[2]-sol$solution[3]
```

```
## [1] 7
```

3. Problemas Clásicos de Programación Lineal

Características generales de un problema de transporte y asignación

- Surgen con frecuencia en diferentes contextos de la vida real.
- Requieren un número muy grande de variables y restricciones. Por lo que la aplicación directa del método simplex resulta complicada y, por tanto, supone un gran esfuerzo computacional.
- La mayor parte de los coeficientes a_{ij} son cero y los que son distintos de cero suelen seguir un patrón.
- Su estructura ha permitido desarrollar algoritmos específicos más eficientes que el algoritmo del Simplex. Se resuelven problemas de hasta 62 millones de variables.

3.1. El problema de transporte

Objetivo: Determinar las cantidades que hay que enviar desde cada origen a cada destino para satisfacer todas las demandas sin superar los límites que establece la oferta y de forma que se minimice el coste total de distribución.

El origen i con $i = 1, \ldots, m$ dispone de b_i unidades para distribuir a los distintos destinos y el destino j con $j = 1, \ldots, n$ tiene una demanda d_j unidades que recibe desde los orígenes.

$$\text{mín} \sum_{j=1}^{n} \sum_{i=1}^{m} c_{ij} x_{ij}$$

$$\sum_{j=1}^{n} x_{ij} \leq b_i, i = 1, \ldots, m$$

$$\sum_{i=1}^{m} x_{ij} \geq d_j, j = 1, \ldots, n$$

$$x_{ij} \in \mathbb{Z}^+$$

Cualquier problema que se ajuste a esta formulación se considera un problema de transporte independientemente del contexto fisico.

Propiedad de las soluciones posibles: un problema de transporte tiene solución si y solo si Oferta Total $\geq$ Demanda total, es decir $\sum_{i=1}^{m} b_i \geq \sum_{j=1}^{n} d_j$

El modelo se dice que está d*esequilibrado* cuando: Oferta total $\neq$ Demanda total. Si Oferta total $>$ Demanda total el problema es imposible.

Es usual que los problemas de transporte vengan dados a partir de lo que se conoce como *Tabla de costos y requerimientos.* Esta tabla consta de tantas filas como orígenes y tantas

columnas como destinos tenga el problema bajo estudio. En el interior de la tabla se tienen los costos c_{ij}.

Ejemplo:

Dada la siguiente tabla de costos y requerimientos de un problema de transporte, expresar el problema de programación lineal asociado.

	D_1	D_2	D_3	b_i
O_1	3	2	5	8
O_2	4	3	1	6
d_j	10	5	6	

La formulación del problema sería:

$$\text{mín } 3x_{11} + 2x_{12} + 5x_{13} + 4x_{21} + 3x_{22} + x_{23}$$

s.a.

$$x_{11} + x_{12} + x_{13} \leq 8$$
$$x_{21} + x_{22} + x_{23} \leq 6$$
$$x_{11} + x_{21} \geq 10$$
$$x_{12} + x_{22} \geq 5$$
$$x_{13} + x_{23} \geq 6$$
$$x_{ij} \in \mathbb{Z}^+$$

En este problema la oferta es menor que la demanda y por tanto el problema es imposible.

3.1.1. 3.1.1. Problemas de transporte con `lp.transport()`

El paquete `lpSolve` incluye también la función `lp.transport()` para resolver el problema de transporte.

Uso de la función:

- Función: `lp.transport (cost.mat, direction="min", row.signs, row.rhs, col.signs, col.rhs, presolve=0, compute.sens=0, integers = 1:(nc*nr) )`
- Argumentos más utilizados:
  - `cost.mat`: La matriz de costes donde el elemento ij-ésimo elemento es el coste de transportar un artículo desde el origen `i` hasta el destino `j`.
  - `direction`: Variable cadena de caracteres que indica la dirección de optimización: min (por defecto) o max.

- `row.signs`: Vector de cadenas de caracteres que dan la dirección de las restricciones de fila.
- `row.rhs`: Vector de términos independientes de las restricciones de fila.
- `col.signs`: vector de cadenas de caracteres que dan la dirección de las restricciones de columna.
- `col.rhs`: Vector de términos independientes de las restricciones de columna.
- `integers`: Vector numérico que da los índices de las variables que deben ser enteras. La longitud de este vector debe ser igual al número de variables enteras. *Por defecto todas las variables son enteras.* Se debe especificar = `NULL` para que las variables no sean enteras

- Devuelve: un objeto de la clase `lp`, que es una lista con componentes (ver Sección 2.3).

3.1.2. Ejemplo 1

Como ejemplo, vamos a resolver el problema de transporte definido por los siguientes datos:

	D_1	D_2	D_3	D_4	D_5	b_i
O_1	3	4	6	8	9	30
O_2	2	2	4	5	5	80
O_3	2	2	2	3	3	10
O_4	3	3	2	4	2	60
d_j	10	50	20	80	20	

- Las filas corresponden a los lugares de Origen y las columnas a los lugares de Destino.
- Los valores de la matriz son los costes de transporte desde cada origen a cada destino.
- El último elemento de cada fila es la oferta de producto en cada origen.
- El último elemento de cada columna es la demanda total en cada destino.

A continuación definimos todos los parámetros del problema y los pasamos como argumentos de `lp.transport()` en el orden adecuado:

```
costes <- rbind (c(3,4,6,8,9),
                 c(2,2,4,5,5),
                 c(2,2,2,3,3),
                 c(3,3,2,4,2))
dir.f <- rep('<=', 4)
b <- c(30, 80, 10, 60)
dir.c <- rep('>=', 5)
d <- c(10, 50, 20, 80, 20)
sol <- lp.transport ( costes , "min", dir.f, b, dir.c, d)
sol$objval
```

```
## [1] 610
```

```
sol$solution
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]   10   20    0    0    0
## [2,]    0   30    0   50    0
## [3,]    0    0    0   10    0
## [4,]    0    0   20   20   20
```

La solución óptima del problema es:

- Desde $O1$ enviar 10 unidades a D_1 y 20 a D_2
- Desde O_2 enviar 30 unidades a D_2 y 50 a D_4
- Desde O_3 enviar 10 unidades a D_4
- Desde O_4 enviar 20 unidades a D_3, 20 a D_4 y 20 a D_5.

3.1.3. Ejemplo 2

Una empresa produce un único artículo en tres plantas, A1, A2 y A3. La capacidad de producción mensual de la empresa está limitada a 2000 unidades mensuales en cada una de las plantas. La empresa tiene cuatro clientes mayoristas cuyas demandas mensuales son, exactamente, 1000, 1200, 1500 y 1300 unidades respectivamente. Los costes de envío (en euros) a los 4 clientes mayoristas que la empresa tiene vienen dados por la siguiente tabla:

	1	2	3	4
A1	30	10	25	20
A2	15	25	30	10
A3	20	30	15	20

Encontrar la solución óptima para minimizar los costes de envío.

```
costes <- rbind (c(30,10,25,20),
                 c(15,25,30,10),
                 c(20,30,15,20))
dir.f <- rep("<=", 3)
b <- rep (2000,3)
dir.c <- rep(">=",4)
d <- c(1000,1200,1500,1300)
sol <- lp.transport (costes , "min", dir.f, b, dir.c, d)
sol$objval
```

```
## [1] 64000
```

```
sol$solution
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0 1200    0    0
## [2,]  700    0    0 1300
## [3,]  300    0 1500    0
```

La solución óptima del problema es:

- En la planta A1 producir los 1200 artículos del mayorista 2.
- En la planta A2 producir los 1300 artículos para el mayorista 4 y 700 para el mayorista 1.
- En la planta A3 producir los 1500 artículos para el mayorista 3 y 300 para el mayorista 1.
- Esta solución óptima conlleva un coste de 64000€.

Supongamos ahora que el objetivo de la empresa es organizar la producción en cada uno de los meses para obtener el máximo beneficio. El beneficio unitario que le proporciona su producto, considerados los costes de producción y el precio de venta, es de 110 unidades en la planta A1, 100 en la planta A2 y 120 en la planta A3. En este caso los valores cij deben de ser los beneficios de producir en la planta Ai y enviarlo al cliente j para su venta y vendrán dados por el beneficio unitarios menos el coste de envío.

```
benf.ud <- matrix (rep(c(110,100,120), 4), ncol =4)
benef <- benf.ud - costes
sol <- lp.transport (benef , "max", dir.f, b, dir.c, d)
sol$objval
```

```
## [1] 585000
```

```
sol$solution
```

```
##      [,1] [,2] [,3] [,4]
## [1,]    0 2000    0    0
## [2,]  700    0    0 1300
## [3,]  300    0 1700    0
```

En este caso la solución óptima es:

- En la planta A1 producir 2000 artículos del mayorista 2.
- En la planta A2 producir 700 artículos para el mayorista 1 y 1300 para el mayorista 4.
- En la planta A3 producir 300 artículos para el mayorista 1 y 1700 para el mayorista 3.
- Esta solución óptima conlleva un beneficio de 585000€.

3.2. El problema de asignación

Situación: Asignar recursos a tareas cuando:

- Nº de recursos = Nº de Tareas = n
- Cada recurso se debe asignar a una única tarea exactamente
  - Cada tarea debe tener asignado exactamente un único recurso
  - Para cada pareja (recurso, tarea) se conoce el coste que supone realizar la tarea utilizando dicho recurso.

Objetivo: Determinar cómo deben hacerse las n asignaciones para que el coste total de la asignación sea mínima.

El Problema de Asignación es un caso particular del Modelo del Transporte.

Ejemplo:

Los tres hijos de Jacinto García, Juan, Pepe y Lucía, quieren ganar algún dinero para cubrir sus gastos de móvil del mes actual. El Sr. García ha elegido tres tareas para sus hijos: podar el césped, pintar el garaje y lavar los tres coches de la familia. Para evitar las peleas entre hermanos les pidió que entregaran una nota secreta indicando el pago (en euros) que ellos considerarían justo por cada una de las tareas. Los hijos se pusieron de acuerdo en aceptar la asignación de tareas que finalmente hiciese su padre. A la vista de las notas entregadas (tabla siguiente) ¿qué asignación debería hacer el Sr. García para tener que pagar lo mínimo posible?

	Podar	Pintar	Lavar
Juan	15	10	9
Pepe	9	15	10
Lucía	10	12	8

Si $x_{ij} = 1$ si el hijo i realiza la tarea j.

$i = 1$ (Juan), $i = 2$ (Pepe), $i = 3$ (Lucía)

$j = 1$ (Podar), $j = 2$ (Pintar), $j = 3$ (Lavar)

$$\text{mín}\, 15x_{11} + 10x_{12} + 9x_{13} + 9x_{21} + 15x_{22} + 10x_{23} + 10x31 + 12x_{32} + 8x_{33}$$

s.a.

$$x_{11} + x_{12} + x_{13} = 1$$

$$x_{21} + x_{22} + x_{23} = 1$$

$$x_{31} + x_{32} + x_{33} = 1$$

$$x_{11} + x_{21} + x_{31} = 1$$
$$x_{12} + x_{22} + x_{32} = 1$$
$$x_{13} + x_{23} + x_{33} = 1$$
$$x_{ij} \in \{0, 1\}$$

3.2.1. Problemas de asignación con `lp.assign()`

El paquete `lpSolve` incluye también la función `lp.assign()` para resolver este tipo de problemas.

Uso de la función:

- Función: `lp.assign(cost.mat, direction = c("min", "max"), compute.sens = 0)`
- Argumentos más utilizados:
  - `cost.mat`: La matriz de costes donde el elemento ij-ésimo elemento es el coste de asignar al recurso `i` la tarea `j`.
  - `direction`: Variable cadena de caracteres que indica la dirección de optimización: `min` (por defecto) o `max`.
- Devuelve: un objeto de la clase `lp`, que es una lista con componentes (ver Sección 2.3).

3.2.2. Ejemplo 1

Estos son los problemas más sencillos de resolver. En el ejemplo anterior del Sr. García:

```
costes <- rbind (c(15,10,9),
                 c(9,15,10),
                 c(10,12,8))
sol <- lp.assign ( costes )
sol$solution
```

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

```
sol$objval
```

```
## [1] 27
```

La solución óptima es asignar la tarea de podar a Pepe, la tarea de pintar a Juan y la tarea de lavar a Lucía. El coste de esta solución es 27.

3.3. El problema del transbordo

Situación: Enviar un bien desde unos puntos de origen a unos puntos de destino pero pudiendo pasar por puntos intermedios.

A veces en la vida real resulta más económico enviar mercancías a través de puntos intermedios en lugar de hacerlo directamente desde el origen hasta el destino. En este tipo de problemas habrá:

- Orígenes puros: sólo se puede enviar desde ellos.
- Destinos puros: sólo se puede recibir en ellos.
- Transbordos: pueden enviar y/o recibir mercancías.

Para construir una tabla de transporte:

- Incluir una fila por cada punto de oferta y de transbordo
- Incluir una columna por cada punto de demanda y de transbordo
- Cada punto `i` de oferta debe tener una oferta igual a su oferta original `bi` y cada punto de demanda `j` debe tener una demanda igual a su demanda original `dj`
- Se define el "buffer": $B = \sum b_i (= \sum d_j$ si el problema está equilibrado)
- Cada punto de transbordo debe tener una oferta igual a la oferta original + B y una demanda igual a su demanda original + B. Como de antemano no se conoce la cantidad que transitaría por cada punto de transbordo, la idea es asegurar que no se exceda su capacidad. Se agrega B a la oferta y a la demanda del punto de transbordo para no desbalancear la tabla.
- Si es necesario, se debe agregar un punto de demanda ficticio (con oferta 0 y demanda igual al excedente) para balancear el problema. Los costos de envío al punto ficticio deben ser cero.

3.3.1. Ejemplo 1

Una fábrica posee dos plantas, una en Memphis y otra en Denver. La planta de Memphis puede producir hasta 150 unidades al día y la de Denver hasta 200 unidades al día. Los productos son enviados por avión a Los Angeles y Boston. En ambas ciudades, se requieren 130 unidades diarias. Existe una posibilidad de reducir costos enviando algunos productos en primer lugar a New York o a Chicago y luego a sus destinos finales. Los costos unitarios de cada tramo factible se ilustran en la siguiente tabla:

	Memphis	Denver	N.Y.	Chicago	L.A.	Boston
Memphis	0	-	8	13	25	28
Denver	-	0	15	12	26	25
N.Y.	-	-	0	6	16	17
Chicago	-	-	-	-	14	16
L.A.	-	-	-	-	0	-
Boston	-	-	-	-	-	0

La fábrica desea satisfacer la demanda, minimizando el costo total de envío. En este problemas, Memphis y Denver son puntos de oferta de 150 y 200 unidades respectivamente. New York y Chicago son puntos de transbordo. Los Angeles y Boston son puntos de demanda de 130 unidades cada uno.

En el ejemplo, $B = 150 + 200 = 350$. La demanda total es $D = 130 + 130 = 260$. Luego, el punto ficticio debe tener una demanda de $350 - 260 = 90$. Como en el ejemplo los puntos de transbordo no tienen ni demanda ni oferta por sí mismos, la oferta y demanda en la tabla deber ser igual a b.

	N.Y.	Chicago	L.A.	Boston	Ficticio	Oferta
Memphis	8	13	25	28	0	150
Denver	15	12	26	25	0	200
N.Y.	0	6	16	17	0	350
Chicago	6	0	14	16	0	350
Demanda	**350**	**350**	**130**	**130**	**90**	

Una vez planteada la tabla, se puede emplear la función vista anteriormente para obtener la solución óptima.

```
costes <- rbind (c(8,13,25,28,0),
                 c(15,12,26,25,0),
                 c(0,6,16,17,0),
                 c(6,0,14,16,0))
dir.f <- rep("=", 4)
b <- c(150,200,350,350)
dir.c <- rep("=",5)
d <- c(350,350,130,130,90)
sol <- lp.transport ( costes , "min", dir.f, b, dir.c, d)
sol$solution
```

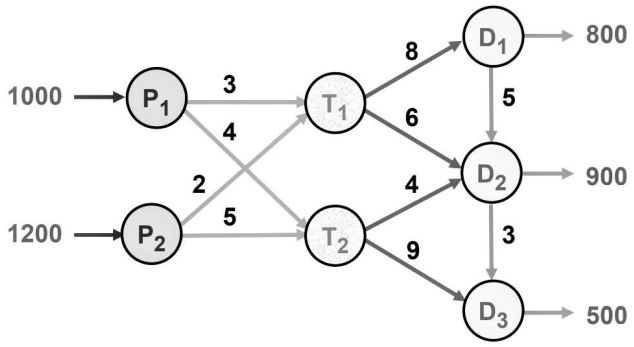
```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]  150    0    0    0    0
## [2,]    0    0    0  110   90
## [3,]  200    0  130   20    0
## [4,]    0  350    0    0    0
```

```
sol$ objval
```

```
## [1] 6370
```

3.3.2. Ejemplo 2

Dos fábricas de automóviles, P1 y P2, están conectadas a tres distribuidores, D1, D2 y D3, por medio de dos centros de tránsito, T1 y T2, de acuerdo con la red que se muestra debajo. Las cantidades de la oferta en las fábricas P1 y P2, son de 1000 y 1200 automóviles, y las cantidades de la demanda en las distribuidoras D1, D2 y D3, son de 800, 900 y 500 automóviles. El coste de envío por automóvil (en decenas de euros) entre los pares de nodos, se muestra en los arcos de conexión de la red. Encontrar el plan de envío óptimo para minimizar los costes.



En este caso, no se pueden realizar envíos desde las fábricas a los destinos directamente. Una solución simple es especificar un alto coste para aquellos envíos que no se pueden realizar, de manera que no sean elegidos en la solución óptima que minimiza los costes. En los nodos D1 y D2, que son nodos destino y trasbordo a la vez, habrá que ajustar la demanda: demanda original + B.

	T1	T2	D1	D2	D3	Oferta
P1	3	4	100	100	100	1000
P2	2	5	100	100	100	1200
T1	0	100	8	6	100	2200
T2	100	0	100	4	9	2200
D1	100	100	0	5	100	2200
D2	100	100	100	0	3	2200
Demanda	2200	2200	3000	3100	500	

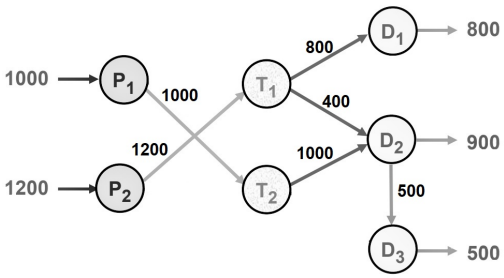
```
costes <- rbind (c(3,4,100,100,100),
                 c(2,5,100,100,100),
                 c(0,100,8,6,100),
                 c(100,0,100,4,9),
```

```
                  c(100,100,0,5,100),
                  c(100,100,100,0,3))
dir.f <- rep("<=",6)
b <- c (1000,1200,2200,2200,2200,2200)
dir.c <- rep(">=",5)
d <- c (2200,2200,3000,3100,500)
sol <- lp.transport ( costes , "min", dir.f, b, dir.c, d)
sol$solution
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    0 1000    0    0    0
## [2,] 1200    0    0    0    0
## [3,] 1000    0  800  400    0
## [4,]    0 1200    0 1000    0
## [5,]    0    0 2200    0    0
## [6,]    0    0    0 1700  500
```

```
sol$objval
```

```
## [1] 20700
```



4. Análisis de Sensibilidad

El análisis de sensibilidad (o de Post-optimalidad) permite, una vez resuelto un problema, analizar cómo afectaría a la solución obtenida la variación dentro de un intervalo de valores de uno de los parámetros, manteniendo fijos los restantes. En caso de pretender estudiar los efectos de la variación de más de un parámetro, se tendría que re-programar el problema.

Para obtener esta información debemos añadir en la función lp() el argumento `compute.sens = TRUE`

Modificaciones en los coeficientes de la función objetivo (Análisis de sensibilidad de c)

- El intervalo `sol$sens.coef.from - sol$sens.coef.to`, indica entre qué valores pueden moverse los coeficientes de las variables.
- Si el coeficiente que se cambia corresponde a una variable básica (con coste reducido = 0), la base óptima no cambia si los nuevos coeficientes están dentro del rango de variación.
- Si el coeficiente que se cambia corresponde a una variable no básica, dentro del intervalo la variable seguirá sin entrar en el problema. El coste reducido es la cantidad en la que hay que modificar, al menos, el valor del coeficiente de la función objetivo para que la variable deje de ser variable no básica y entre en la base. Por lo tanto es la cantidad máxima en la que se puede modificar el coeficiente de la función objetivo sin que se modifique la solución óptima del problema.
- Los costes reducidos vienen dados por los últimos n elementos de `sol$duals`, donde `n` es el número de variables. Los m primeros elementos corresponden a las `m` restricciones.

Términos independientes de las restricciones (Análisis de sensibilidad de b)

- Se pueden calcular las holguras como `A%*%sol$solution-b`.
- Los precios sombra vienen dados por los m primeros elementos de `sol$duals`.
- Los intervalos para los términos independientes vienen dados por los `m` primeros elementos de `sol$duals.from` y `sol$duals.from`.
- En aquellas variables con holgura, el intervalo vendrá de $-\infty$ a ∞ y deberemos calcular el límite con la holgura.
- En las restricciones donde no hay holgura, dentro del intervalo, el nuevo valor de la función objetivo viene dado por $z^* = z + (\text{precio sombra})(\hat{b}_i - b_i)$

4.1. Ejemplo 1

Una empresa fabrica encimeras de mármol y de granito en sus dos fábricas. En la primera puede producir como máximo, 1000 encimeras y en la segunda 800, debido a problemas de almacenaje. En la tabla se muestran los beneficios, en euros, para cada tipo de encimera según el punto de fabricación:

	Mármol	Granito
Fábrica 1	1000	1300
Fábrica 2	600	800

En la segunda fábrica tardan 2 horas en fabricar una encimera, mientras que en la primera las encimeras de mármol llevan 3 horas y las de granito 4. Si la empresa puede vender como máximo 900 encimeras de cada tipo y dispone de 4000 horas de trabajo con un coste de 20 euros la hora. Formula el problema que permita determinar cómo se debe distribuir la producción para obtener los máximos beneficios.

Llamemos x_{ij} al número de encimeras que se fabrican en la fabrica i $(i = 1, 2)$, del material j $(j = M, G)$ entonces tenemos que:

$$\text{máx } 1000x_{1M} + 1300x_{1G} + 600x_{2M} + 800x_{2G} - (3 \cdot 20)x_{1M} - (4 \cdot 20)x_{1G} - (2 \cdot 20)(x_{2G} + x_{2M}) =$$

$$940x_{1M} + 1220x_{1G} + 560x_{2M} + 760x_{2G}$$

s.a.

$$x_{1M} + x_{2M} \leq 900 \text{ (Encimeras de mármol en las dos fábricas)}$$
$$x_{1G} + x_{2G} \leq 900 \text{ (Encimeras de granito en las dos fábricas)}$$
$$3x_{1M} + 4x_{1G} + 2x_{2M} + 2x_{2G} \leq 4000 \text{ (Horas de trabajo total)}$$
$$x_{1M} + x_{1G} \leq 1000 \text{ (Encimeras en la fábrica 1)}$$
$$x_{2M} + x_{2G} \leq 800 \text{ (Encimeras en la fábrica 2)}$$

Procedemos a resolverlo:

```
coef <- c(940, 1220, 560, 760)
######## x1m x1g x2m x2g
A <- rbind (c(1,0,1,0),
            c(0,1,0,1),
            c(3,4,2,2),
            c(1,1,0,0),
            c(0,0,1,1))
dir <- rep("<=", 5)
```

```
b <- c(900,900,4000,1000,800)
sol <- lp("max", coef , A, dir , b, compute.sens = TRUE )
sol
```

```
## Success: the objective function is 1360000
```

```
sol$solution
```

```
## [1] 800   0   0 800
```

Análisis de sensibilidad de `c`

- Las variables básicas son x_{1M} y x_{2G}, es decir, la fábrica uno hace 800 encimeras de mármol y la fábrica 2, 800 de granito.
- Los intervalos de variación permitidos, para los coeficientes de la función objetivo son:

  ```
  sol$sens.coef.from
  ```

  ```
  ## [1]  9.150000e+02 -1.000000e+30 -1.000000e+30  6.266667e+02
  ```

  ```
  sol$sens.coef.to
  ```

  ```
  ## [1] 1.140000e+03 1.253333e+03 7.600000e+02 1.000000e+30
  ```

  - El coeficiente de x_{1M} podría variar entre 915 y 1140.
  - $(-\infty, 1253.3334)$, el coeficiente de x_{1G} tendría que aumentar hasta ≈ 1254 para que la base óptima cambie.
  - $(-\infty, 760)$, el coeficiente de x_{2M} tendría que aumentar hasta 761 para que la base óptima cambie.
  - El coeficiente de x_{2G} puede variar entre 626.6667 hasta infinito.
- Lo que deben de aumentar los coeficientes de las variables no básicas también se puede ver a través de los costes reducidos.

  ```
  sol$duals[6:9]
  ```

  ```
  ## [1]    0.00000  -33.33333 -200.00000    0.00000
  ```

  Por ejemplo, para que x_{2M} fuera variable básica, habría que mejorar su coeficiente en la función objetivo en al menos 200. Por lo que el coeficiente debe ser mayor de 760 (560+200).

Análisis de sensibilidad de `b`

- Calculamos las holguras:

```
round (abs(A %*%sol$solution -b),4)
```

```
##      [,1]
## [1,]  100
## [2,]  100
## [3,]    0
## [4,]  200
## [5,]    0
```

- En las restricciones con holgura, los términos independientes pueden variar:
  - $(900 - 100, \infty) = (800, \infty)$ Total encimeras de mármol
  - $(900 - 100, \infty) = (800, \infty)$ Total encimeras de granito
  - $(1000 - 200, \infty) = (800, \infty)$ Total encimeras en la fábrica 1
- En las restricciones sin holgura utilizamos los precios sombra:

```
sol$duals[c(3,5)] # Precio Sombra
```

```
## [1] 313.3333 133.3333
```

```
sol$duals.from [c(3,5)]
```

```
## [1] 1600  650
```

```
sol$duals.to[c(3,5)]
```

```
## [1] 4300  900
```

  - Las horas de trabajo total pueden moverse entre 1600 y 4300. Por cada hora de trabajo que se añada, la función objetivo aumenta 313.3333. Por lo tanto, este sería el precio máximo que la empresa estaría dispuesta a pagar por una hora extra de trabajo.
  - Si se ponen, por ejemplo, 4300 horas de trabajo. La función objetivo aumenta 94000€.

Preguntas clásicas:

1. ¿Cuál es la cantidad máxima que la empresa tendría que estar dispuesta a pagar por una hora extra de trabajo? ¿Cuál sería el beneficio si se dispusiera de 3000 horas de trabajo?

   La cantidad máxima de la empresa estaría dispuesta a pagar por una hora extra de trabajo sería: 313.3333 (precio sombra). Si se dispusiera de 3000 horas de trabajo en lugar de 4000 el beneficio seria 1046667 en lugar de 1360000. Este resultado sale de esta operación: $1360000 + 313.3333(3000 - 4000)$. *Como 3000 pertenece al intervalo (1600, 4300), no varían las variables básicas.*

2. ¿Cuánto tendría que valer el beneficio de una encimera de mármol en la fábrica 2 para que fuera rentable su fabricación?

   $x_{2M} = 0$, *para que sea variable básica con valor distinto de cero, su coeficiente en la función objetivo debe aumentar en 200 como mínimo (que es el valor del coste reducido de esa variable). Por lo que el coeficiente debe ser mayor de 760 (560+200=760).*

3. ¿Cuál sería la ganancia si el beneficio de las encimeras de mármol en la primera fábrica fuera de 1300? ¿Cambian las variables básicas?

   El intervalo para este coeficiente es $(915, 1140)$. *En este caso el coeficiente tomaba valor 940=1000-60 (restando el coste de las horas). Si el beneficio son 1300, entonces el coeficiente tomaría valor 1300-60=1240. Como 1240 no pertenece al anterior intervalo calculado, las variables básicas cambian.*

4. ¿Cómo varía la solución si el beneficio de las encimeras de granito en la fábrica 1 es de 1400? ¿Permanece la base actual óptima?

   El intervalo para este coeficiente es $(-\infty, 1253, 333)$. *En este caso el coeficiente tomaba valor 1220=1300-80 (restando el coste de las horas). Si el beneficio son 1400, entonces el coeficiente tomaría valor 1400-80=1320. Como 1320 no pertenece al anterior intervalo calculado, las variables básicas cambian.*

5. Debido a la demanda se quiere estudiar posibles variaciones en la cantidad de fabricación de encimeras de granito, determina el intervalo de valores en el cual la base permanece óptima

   Calculando las holguras vemos que hay 100 de holgura por lo tanto $(800, \infty)$ es el intervalo en el que la base permanece óptima.

6. Si se pueden producir hasta 1000 encimeras en la segunda fábrica ¿varia la solución?

   El intervalo permitido es (650,900) por tanto si se pudiera producir hasta 1000 encimeras en la segunda fábrica cambiarían las variables básicas

4.2. Ejemplo 2

Una bodega produce vino rosado y tinto. Este año dispone de 90Ton de uva blanca y de 145Ton de uva negra. Para producir un litro de rosado se necesitan 3 kilos de uvas blancas y 2 kilos de negras; mientras que para producir un litro de tinto se necesitan 5 kilos de blanca

y 4 de negra. Por la situación actual del mercado, supone que puede vender 50000 litros de cada tipo de vino como máximo a un precio de 3.2 euros el litro rosado y a 5.5 euros el litro de tinto. Además, si fuera necesario, debido a la mala cosecha se puede comprar vino a pequeños fabricantes para su posterior venta a 5 euros el tinto y 2.7 el rosado. Formular el problema que permita a la bodega maximizar sus beneficios. Llamemos:

- x_R litros que se fabrican de vino rosado.
- x_T litros que se fabrican de vino tinto.
- x_{CR} litros que se compran de vino rosado.
- x_{CT} litros que se compran de vino tinto.

Entonces tenemos que:

$\text{máx}\, 3.2x_R + 5.5x_T + (3.2 - 2.7)x_{CR} + (5.5 - 5)x_{CT} =$

$3.2x_R + 5.5x_T + 0.5x_{CR} + 0.5x_{CT}$

s.a.

$$\begin{aligned} 3x_R + 5x_T &\leq 90000 \ \text{(Kilos uva blanca)} \\ 2x_R + 4x_T &\leq 145000 \ \text{(Kilos uva negra)} \\ x_R + x_{RC} &\leq 50000 \ \text{(litros vino rosado)} \\ x_T + x_{TC} &\leq 50000 \ \text{(litros vino tinto)} \end{aligned}$$

Resolvemos:

```
coef <- c(3.2, 5.5, 0.5, 0.5)
A <- rbind (c(3,5,0,0),
            c(2,4,0,0),
            c(1,0,1,0),
            c(0,1,0,1))
dir <- rep ("<=", 4)
b <- c(90000,145000,50000,50000)
sol <- lp("max", coef , A,dir ,b, compute.sens = TRUE )
sol
```

```
## Success: the objective function is 140000
```

```
sol$solution
```

```
## [1]     0 18000 50000 32000
```

```
round (abs(A %*%sol$solution -b)) # Holguras
```

```
##       [,1]
## [1,]     0
## [2,] 73000
## [3,]     0
## [4,]     0
```

```
sol$duals [1:4] # Precio Sombra
```

```
## [1] 1.0 0.0 0.5 0.5
```

```
sol$duals [5:8] # Coste reducido
```

```
## [1] -0.3  0.0  0.0  0.0
```

```
sol$sens.coef.from
```

```
## [1] -1e+30  5e+00  2e-01  0e+00
```

```
sol$sens.coef.to
```

```
## [1] 3.5e+00 1.0e+30 1.0e+30 1.0e+00
```

```
sol$duals.from[c(1,3,4)]
```

```
## [1]     0     0 18000
```

```
sol$duals.to[c(1,3,4)]
```

```
## [1] 1.8125e+05 1.0000e+30 1.0000e+30
```

Preguntas

1. ¿Cuál es la solución óptima y cuál es el beneficio máximo esperado?

   La solución óptima es producir 18.000 litros de vino tinto, comprar 50000 litros de vino rosado y otros 32000 litros de vino tinto. El beneficio es de 140000

2. ¿Cuál es la cantidad máxima que la bodega estaría dispuesta a pagar por otros 100 kilos de uva blanca? ¿y por otros 100 kilos de uva negra?

   El precio sombra para la primera restriccion es 1, por tanto estaría dispuesto a pagar 1 x 100 = 100€ por los 100 kg de uva blanca.

   Por otros 100 kg de uva negra no pagaría nada porque nos sobran 73.000kg de uva negra.

3. ¿Cuál es el beneficio si sólo se pueden vender 40000 litros de vino tinto?

   El intervalo es $(18000, \infty)$, como 40000 pertenece al intervalo las variables básicas no cambian. El beneficio sería 140000- 0.5*10000=135000€

4. Si se disponen solo de 60 Ton de uvas blancas ¿cuál es la ganancia de la bodega? ¿Se modifica la base óptima?

   El intervalo es $(0, 181250)$, como 60000 pertenece al intervalo las variables básicas no cambian. El beneficio sería 140000- 1*30000=110000€

5. Si se compra el tinto a 4.8 euros el litro ¿se modifica la solución?

   El coeficiente en la función objetivo sería 55 - 4.8 = 0.7. El intervalo para este coficiente es (0,1), como 0.7 pertenece al intervalo las variables básicas son las mismas.

   La solución sería 3.2*0+5.5*18000+0.5*50000+0.7*32000=146400