

**ASIGNATURA: TECNICAS DE OPTIMIZACIÓN**

**PRÁCTICA: OPTIMIZACIÓN CON IpsolveAPI + R**

**SESIONES: 1**

**SOFTWARE: IpSolveAPI + R**

## R

---

**R** es un lenguaje y entorno para computación estadística y gráficos. Es un proyecto GNU que es similar al lenguaje y entorno S que fue desarrollado en Bell Laboratories (anteriormente AT&T, ahora Lucent Technologies) por John Chambers.

R proporciona una amplia variedad de técnicas estadísticas y de investigación operativa (modelado lineal y no lineal, pruebas estadísticas clásicas, análisis de series temporales, clasificación, agrupamiento, ...) y gráficas, y es altamente extensible. El lenguaje S suele ser el vehículo elegido para la investigación en metodología estadística, y R proporciona una ruta de código abierto para desarrollar esta tarea.

R está disponible como Software Libre bajo los términos de la Licencia Pública General GNU de la Free Software Foundation en forma de código fuente. Compila y se ejecuta en una amplia variedad de plataformas UNIX y sistemas similares (incluidos FreeBSD y Linux), Windows y MacOS.

Existen varias librerías en R que permiten resolver modelos de optimización lineal (IpSolve, IpSolveAPI, Rglpk, Rsymphony, linprog, ompr, entre otros (ver el [anexo I](#) de este mismo documento).

## IpSolveAPI en R

---


IpSolveAPI es una de las librerías disponibles en R para resolver modelos de programación lineal y entera. Puedes encontrar los detalles de esta librería [aquí](#) y una lista de las librerías de optimización en R en los anexos de este documento.

La librería IpSolveAPI proporciona un API para la librería lp\_solve que es un optimizador para resolver problemas de programación lineal pura y programación entera/binaria (mixta). Utiliza el algoritmo simplex revisado para resolver los problemas de programación lineal pura y algoritmos Branch and Bound para manejar variables enteras.

Es posible llamar a IpSolveAPI desde R a través de una extensión o módulo. Como tal, parece que IpSolveAPI está completamente integrado con R. Las matrices se pueden transferir directamente entre R e IpSolveAPI en ambas direcciones. La interfaz completa está escrita en C por lo que tiene el máximo rendimiento.

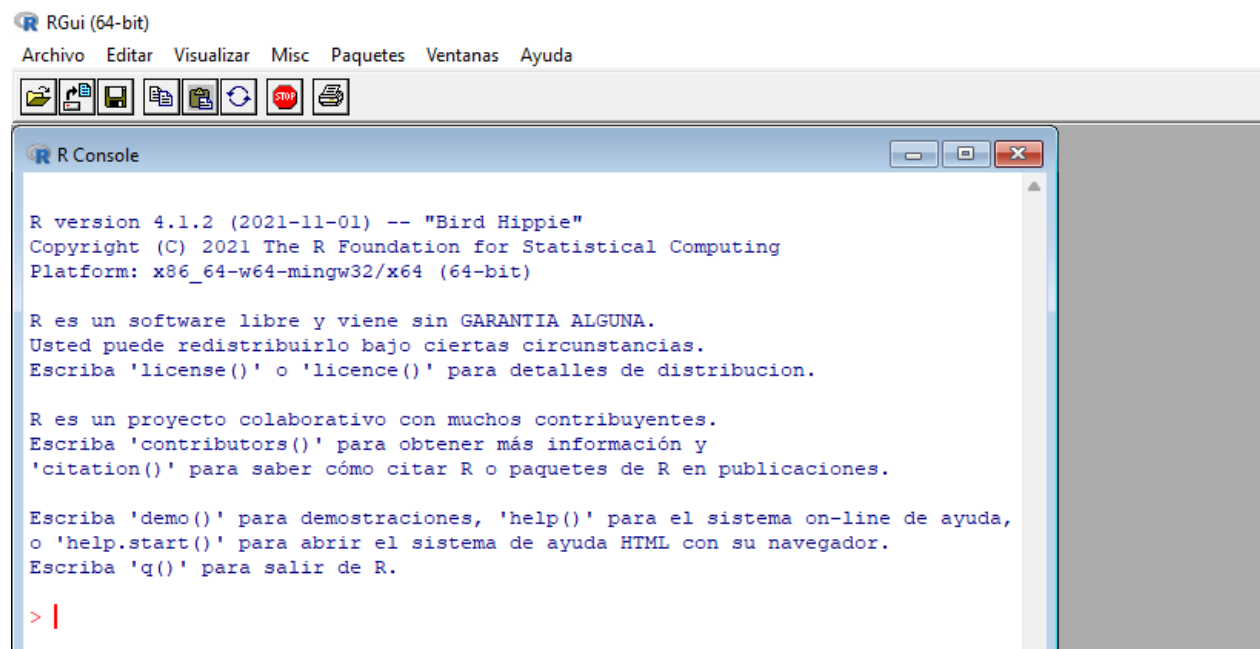
En esta práctica se utilizará la línea de comandos de R, para lo cual es necesario tener abierta la aplicación R.



Cuando se muestre el símbolo  a lo largo del enunciado de la práctica, significa que debes introducir en R el comando que se indique y observar el resultado.

Lo primero será instalar el paquete para a continuación cargarlo en la sesión y dejarlo listo para su uso, si es que el equipo donde trabajas no tiene instalado R. En esta práctica se asume que en el equipo donde trabajas ya está instalado R.

Al ejecutar , aparecerá la siguiente ventana:



## Instalación del controlador lpSolveAPI en R

La instalación del paquete desde CRAN requiere un único comando en el que hay que seleccionar un *Secure CRAN mirror*:



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

## Cargando el controlador lpSolveAPI en R

Pero instalar el paquete no es suficiente. R también debe cargarse en el espacio de memoria antes de que se pueda usar. Esto se puede hacer con el siguiente comando:



```
> library(lpSolveAPI)
```

## Obtención de ayuda

Utilizando el sistema de ayuda integrado de R es posible acceder a documentación para cada función en el paquete lpSolveAPI. Por ejemplo, el comando



```
> ??add.constraint
```

mostrará la documentación para la función `add.constraint`.

Es posible visualizar el listado completo de funciones incluidas en la librería lpSolveAPI con el comando:



```
> ls("package:lpSolveAPI")
```

## Convenciones usadas en la librería lpSolveAPI

La sintaxis utilizada para programar lpSolve a través de la librería lpSolveAPI requiere

- crear en primer lugar un objeto tipo modelo de programación lineal (**L**inear **P**rogram **M**odel **O**bject - **LPMO**),
- resolver el modelo de programación lineal y
- utilizar los métodos `get` para cargar los elementos de la solución.

El primer argumento de prácticamente todas las funciones de la librería lpSolveAPI es el objeto **LPMO** sobre el que se aplicará la función.

Vamos a resolver el siguiente modelo de programación lineal:

$$\text{MAX } Z = x_1 + 9 x_2 + 3 x_3$$

s.a:

$$x_1 + 2 x_2 + 3 x_3 \geq 1$$

$$3 x_1 + 2 x_2 + 2 x_3 \leq 15$$

$$x_1, x_2 \geq 0$$

El modelo se ha resuelto con LINGO® obteniéndose los siguientes informes de solución óptima y análisis de sensibilidad:

Global optimal solution found.			Ranges in which the basis is unchanged:			
Objective value:			Objective Coefficient Ranges:			
Variable	Value	Reduced Cost	Variable	Current Coefficient	Allowable Increase	Allowable Decrease
X1	0.000000	12.50000	X1	1.000000	12.50000	INFINITY
X2	7.500000	0.000000	X2	9.000000	INFINITY	6.000000
X3	0.000000	6.000000	X3	3.000000	6.000000	INFINITY
			Righthand Side Ranges:			
Row	Slack or Surplus	Dual Price	Row	Current RHS	Allowable Increase	Allowable Decrease
1	67.50000	1.000000	R1	1.000000	14.00000	INFINITY
R1	14.00000	0.000000	R2	15.00000	INFINITY	14.00000
R2	0.000000	4.500000				

A lo largo de esta práctica vamos a reproducir estos informes de Lingo® utilizando lpSolveAPI y R.

### IMPORTANTE:

Conforme avanzamos en la práctica y vamos obteniendo información de la solución óptima y análisis de sensibilidad, **identifica** su equivalencia en el informe Lingo®.

Para crear un objeto **LPMO**, utilizaremos el comando `make.lp(m, n)`. Si el objeto LPMO es un modelo con 3 variables (indicaremos 0 restricciones porque vamos a ir añadiendo de forma dinámica las restricciones):



```
> m <- make.lp(0, 3)
```

Hay que tener en cuenta que la función objetivo y el lado derecho de las restricciones no se tienen en cuenta en la dimensión del objeto **LPMO**.

El objeto **LPMO** que hemos creado está inicializado y vacío. Lo podemos visualizar:



```
> m
```


El siguiente paso es utilizar métodos tipo `set` para definir el modelo de programación lineal como se describe en los siguientes apartados.

## Creación y resolución de programas lineales mediante el paquete lpSolveAPI de R

Una vez que se ha creado el objeto **LPMO**, se le van añadiendo


1. los coeficientes de las variables en las restricciones,
2. sentido de las restricciones ( $\leq$ ,  $\geq$  o  $=$ ) y
3. coeficientes en la función objetivo:

1. Una vez creado el objeto **LPMO**, se indican los **coeficientes de las variables en la función objetivo** (utilizaremos el método `set.objfn`):

```
 > set.objfn(model, c(1, 9, 3))
```

```
 > m
```


2. **Restricciones**: coeficientes lado izquierdo, sentido de la restricción y valor del segundo miembro:

```
 > add.constraint(model, c(1, 2, 3), ">=", 1)  
> add.constraint(model, c(3, 2, 2), "<=", 15)
```

```
 > m
```

Por defecto, todas las variables de decisión se crean con valor en el intervalo  $[0, \dots, +\infty)$

Como observamos, el **sentido de la función objetivo** debe ser Maximizar y por tanto, lo debemos cambiar ya que por defecto el objeto **LPMO** se crea con Minimización. Utilizamos la función `lp.control` para indicar el criterio de la función objetivo:

```
 > lp.control(m, sense="max")
```



```
> m
```

## EJECUTAMOS IpSolveAPI:

Una vez definido el modelo, para resolverlo se usa la función `solve ()` que devuelve el código

- 0 Optimal
- 1 Sub-optimal
- 2 Infeasible
- 3 Unbounded
- 4 Degenerated
- 5 Numeric Failure
- 6 Aborted Process
- 7 Timeout
- 9 Model Solved by pre-solver



```
> solve(m)
```

El retorno del valor 0 indica que el modelo se ha resuelto de forma satisfactoria.

A continuación, accedemos a los resultados del modelo utilizando para ello los métodos

- `get.objective`,
- `get.variables` y
- `get.constraints`:

Visualizamos el valor óptimo de la Función Objetivo:



```
> get.objective(m)
```

A continuación, visualizamos el valor óptimo de las variables



```
> get.variables(m)
```

y el valor que toma la parte izquierda de las restricciones una vez obtenida la solución óptima



```
> get.constraints(m)
```

Compara los resultados obtenidos con los proporcionados por Lingo®.

## Costes Reducidos y Costes de Oportunidad

Para obtener los valores de los coste reducidos de las variables decisión y de los costes de oportunidad de las restricciones, debemos utilizar el método `get.dual.solution`:



```
> get.dual.solution(m)
```

Compara los resultados obtenidos con los proporcionados por Lingo®.

Que nos muestra en primer lugar los costes de oportunidad (incluyendo a la función objetivo con 1.0) y después los costes reducidos de las variables decisión.

De modo que, si tengo “m” restricciones y “n” variables,

- los primeros m valores corresponden a los **COSTES DE OPORTUNIDAD** y
- los siguientes n valores son los **COSTES REDUCIDOS**.

**NOTA IMPORTANTE:** el signo del COSTE REDUCIDO y el COSTE DE OPORTUNIDAD indica el efecto **final** sobre el valor óptimo de la función objetivo, de modo que, si es positivo, significa un aumento del valor óptimo de la función objetivo y si es negativo, significa un decremento. Por tanto, **la interpretación es diferente a como lo hace Lingo®**.

## Análisis de sensibilidad

Para obtener los intervalos de análisis de sensibilidad de los **coeficientes de las variables en la función objetivo** utilizaremos la función `get.sensitivity.obj`:



```
> get.sensitivity.obj(m)
```

Compara los resultados obtenidos con los proporcionados por Lingo®.

Que nos muestra los límites del análisis de sensibilidad de los coeficientes de la función objetivo.

De forma similar, se accede al análisis de sensibilidad del **lado derecho de las restricciones** mediante el método `get.sensitivity.rhs`:



```
> get.sensitivity.rhs(m)
```

Esta función nos muestra en primer lugar los costes de oportunidad y costes reducidos y a continuación los límites de análisis de sensibilidad.

**Compara los resultados obtenidos con los proporcionados por Lingo®.**

## Variables enteras

Por defecto, las variables son continuas y no negativas. Si alguna es entera o binaria se declara con la función `set.type()`.

Supongamos que se desea indicar que el modelo incluye variables de naturaleza **entera**. En particular, queremos indicar que la segunda y la tercera variable son enteras:



```
> set.type(m, 2, "integer")  
> set.type(m, 3, "integer")
```



```
> m
```

## Variables binarias

También es posible especificar que alguna variable del modelo es de naturaleza **binaria**. En particular si queremos indicar que la primera variable es binaria, usaremos la función `set.type` pero esta vez indicaremos `"binary"`:



```
> set.type(m, 1, "binary")
```



```
> m
```

## Variables acotadas

Para indicar que una variable tomará valores acotados entre ciertos límites (distintos de  $[0, \dots, +\infty)$ ), se utilizará la función `set.bounds()`.

## Etiquetas para variables y restricciones

Por último, es posible etiquetar las variables y las restricciones. Para ello deberemos crear una



lista con los nombres que deseamos utilizar y asignarlas usando la función `dimnames`



```
> colNames<-c("x1", "x2", "x3")  
> rowNames<-c("res1", "res2")  
> dimnames(m)<-list(rowNames,colNames)
```



```
> m
```

Por tanto, las funciones a utilizar para formular el modelo matemático

$$\text{MAX } Z = x_1 + 9 x_2 + 3 x_3$$

s.a:

$$\begin{aligned} x_1 + 2 x_2 + 3 x_3 &\geq 1 \\ 3 x_1 + 2 x_2 + 2 x_3 &\leq 15 \\ x_1, x_2 &\geq 0 \end{aligned}$$

y obtener la solución óptima de **lpSolveAPI** y **R** son las siguientes:

### Construcción del modelo matemático

```
> m <- make.lp(0, 3)  
> set.objfn(m, c(1, 9, 3))  
> add.constraint(m, c(1, 2, 3), ">=", 1)  
> add.constraint(m, c(3, 2, 2), "<=", 15)  
> lp.control(m, sense="max")  
  
> colNames<-c("x1", "x2", "x3")  
> rowNames<-c("res1", "res2")  
> dimnames(m)<-list(rowNames,colNames)
```

### Obtención solución óptima y análisis de sensibilidad

```
> solve(m)  
> get.objective(m)  
> get.variables(m)  
> get.constraints(m)  
  
> get.dual.solution(m)  
> get.sensitivity.obj(m)  
> get.sensitivity.rhs(m)
```

Estas funciones son las que se utilizarán en el siguiente apartado.

## Separando datos del modelo

Es posible formular el modelo matemático separando los datos del modelo matemático:

```
> m <- make.lp(0, 3)
> aij <- matrix(c(1, 2, 3, 3, 2, 2), nrow=2, byrow=TRUE)
> ci <- c(1,9,3)
> bi <- c(1,15)

> set.objfn(m,ci)
> add.constraint(m, aij[1,], ">=", bi[1])
> add.constraint(m, aij[2,], ">=", bi[2])
```

## Escribir y leer un modelo matemático en formato MPS

Para escribir el modelo matemático en un fichero que pueda ser leído por otro software de optimización, se utiliza el formato MPS diseñado por IBM. Usaremos la función `write.lp`:



```
> write.lp(m, 'modelo.mps', type="mps")
```

El fichero se habrá creado en el directorio de trabajo de `r`. Para saber cual es el directorio de trabajo de `r`, llamaremos a la función `getwd`:



```
> getwd()
```

**Busca el fichero generado y observa el formato MPS** (el tipo de ficheros `.MPS` pueden ser abiertos desde el *Bloc de notas*). De esta forma, conociendo el formato MPS, desde otra aplicación se podrá generar un fichero con el modelo matemático para que `lpSolveAPI` en R pueda obtener la solución óptima.

Para leer con `lpSolveAPI` en R un fichero creado por otra aplicación usaremos la función `read.lp`:



```
> m <- read.lp('modelo.mps', type="mps")
```

## Hazlo tú mismo

En una empresa se fabrican tres productos A, B y C. Los tres productos comparten en sus procesos de producción cuatro máquinas M1, M2, M3 y M4. El producto A utiliza tres operaciones en las máquinas M1, M3 y M4. El producto B utiliza sólo dos operaciones en las máquinas M1 y M3 o en las máquinas M2 y M4. El producto C puede fabricarse utilizando las máquinas M1 y M3 o las máquinas M2, M3 y M4.

El tiempo necesario en minutos por unidad producida, para cada posibilidad de producción en cada máquina, el coste variable de producción por minuto, la capacidad diaria de producción de cada máquina y las demandas diarias mínimas de los tres productos se presentan en la siguiente tabla.

**Datos técnicos y económicos**

Producto	Proceso	Tiempo (min/unidad)				Demanda diaria mínima
		M1	M2	M3	M4	
A	1	10		6	3	36
B	B1	8		10		45
	B2		6		9	
C	C1	8		16		10
	C2		10	3	8	
Coste variable por min (um)		40	50	24	30	
Capacidad diaria en min		480	480	480	480	

El objetivo consiste en determinar el esquema de producción que minimice el coste variable total. El modelo matemático en formato Lingo® es el siguiente:

[OBJ] MIN = 634\*A+560\*B1+570\*B2+704\*C1+812\*C2;

[DEMA] A >= 36;

[DEMB] B1 + B2 >= 45;

[DEMC] C1 + C2 >= 10;

[CAPM1] 10\*A + 8\*B1 + 8\*C1 <= 480;

[CAPM2] 6\*B2 + 10\*C2 <= 480;

[CAPM3] 6\*A + 10\*B1 + 16\*C1 + 3\*C2 <= 480;

[CAPM4] 3\*A + 9\*B2 + 8\*C2 <= 480;

**Sigue los pasos descritos anteriormente con lpSolve en R para obtener la solución óptima y el análisis de sensibilidad del problema y rellena los informes que hubiera obtenido Lingo®.**

Objective value:

Variable	Value	Reduced Cost
A		
B1		
B2		
C1		
C2		
Row	Slack or Surplus	Dual Price
OBJ	55464.00	
DEMA		
DEMB		
DEMC		
CAPM1		
CAPM2		
CAPM3		
CAPM4		

Objective Coefficient Ranges			
Variable	Current Coefficient	Allowable Increase	Allowable Decrease
A	634.0000		
B1	560.0000		
B2	570.0000		
C1	704.0000		
C2	812.0000		

Righthand Side Ranges			
Row	Current RHS	Allowable Increase	Allowable Decrease
DEMA	36.00000		
DEMB	45.00000		
DEMC	10.00000		
CAPM1	480.0000		
CAPM2	480.0000		
CAPM3	480.0000		
CAPM4	480.0000		

## ANEXO I: Software de optimización en R

- LP (Linear programming): [boot](#), [glpkAPI](#), [limSolve](#), [linprog](#), [lpSolve](#), [lpSolveAPI](#), [quantreg](#), [rcdd](#), [Rcplex](#), [Rglpk](#), [Rmosek](#), [Rsymphony](#)
- GO (Global Optimization): [DEoptim](#), [DEoptimR](#), [GenSA](#), [GA](#), [pso](#), [rgenoud](#), [cmaes](#), [nloptr](#), [NMOF](#), [OOR](#)
- SPLP (Special models of linear programming like transportation, multi-index, etc.): [clue](#), [lpSolve](#), [lpSolveAPI](#), [quantreg](#), [TSP](#)
- BP (Boolean programming): [glpkAPI](#), [lpSolve](#), [lpSolveAPI](#), [Rcplex](#), [Rglpk](#)
- IP (Integer programming): [glpkAPI](#), [lpSolve](#), [lpSolveAPI](#), [Rcplex](#), [Rglpk](#), [Rmosek](#), [Rsymphony](#)
- MIP (Mixed integer programming and its variants MILP for LP and MIQP for QP): [glpkAPI](#), [lpSolve](#), [lpSolveAPI](#), [Rcplex](#), [Rglpk](#), [Rmosek](#), [Rsymphony](#)
- QP (Quadratic programming): [kernlab](#), [limSolve](#), [LowRankQP](#), [quadprog](#), [Rcplex](#), [Rmosek](#)
- SDP (Semidefinite programming): [Rcsdp](#), [Rdsdp](#)
- CP (Convex programming): [cccp](#)
- COP (Combinatorial optimization): [adagio](#), [CEoptim](#), [TSP](#), [matchingR](#)
- MOP (Multi-objective and goal programming): [caRamel](#), [GPareto](#), [mco](#), [emoa](#), [rmoo](#)
- NLP (Nonlinear programming): [nloptr](#), [alabama](#), [Rsolnp](#), [Rdonlp2](#)
- GRAPH (Programming involving graphs or networks): [igraph](#)
- IPM (Interior-point methods): [kernlab](#), [glpkAPI](#), [LowRankQP](#), [quantreg](#), [Rcplex](#)
- RGA (Methods of reduced gradient type): [stats](#) ([optim\(\)](#)), [gsl](#)
- QN (Methods of quasi-Newton type): [stats](#) ([optim\(\)](#)), [gsl](#), [lbfgs](#), [lbfgsb3c](#), [nloptr](#), [optimParallel](#), [ucminf](#)
- DF (Derivative-free methods): [dfoptim](#), [minqa](#), [nloptr](#)

## ANEXO II: Software de optimización en R más utilizado

### IpSolve

IpSolve is a MILP solver, that runs in a large variety of languages. Models can be passed via input files, an API (application programming interface) or an IDE (integrated development environment). It is also embedded into R via the **IpSolve** and **IpSolveAPI** packages. The reference guide is quite exhaustive, and a good place to start using this software.

[R package IpSolve](#)

[R package IpSolveAPI](#)

### GLPK

GLPK stands for GNU Linear Programming Kit. It is a set of callable libraries written in C intended to solve large scale LP, ILP and MILP models. It is developed by Andrei Makhorin, of the Moscow Aviation Institute. GLPK contains an standalone solver, **glpsol**, that can be called from the command line. It can read models written in a variety of languages, among them CPLEX, convenient for relatively small modelos and GNU MathProg, a standard based on the AMPL format adequate to write large models with regular structure.

GLPK can be used in R via the **Rglpk** package. For windows users is available **GUSEK**, an IDE running the GLPK libraries.

[GLPK homepage](#)

[R package Rglpk](#)

[GUSEK IDE](#)

### SYMPHONY

Symphony is an open-source solver of MILP modelos written in C. It is an initiative of the [COIN-OR](#) project. It can read modelos written in MPS and MathProg formats. It is also implemented in R through the Rsymphony package.

[SYMPHONY homepage](#)

[SYMPHONY user's manual](#)

[R package Rsymphony](#)

### linprog

A library for solving linear programming models, quite popular in MATLAB. There is also a R package available. This library does not solve ILP/MILP models.

[linprog in MATLAB](#)

[linprog R package](#)

## ANEXO III: Formato MPS

MPS es un formato optimizado para tarjetas perforadas: es un archivo ASCII orientado a columnas, normalmente escrito en mayúsculas. Algunas de las secciones utilizadas en el archivo son las siguientes

- 'NOMBRE' (almacena un nombre elegido para el problema)
- 'FILAS' (define los nombres de todas las restricciones)
- 'COLUMNAS' (contiene las entradas de la matriz A- matriz)
- 'RHS' (permite definir los vectores del lado derecho)
- 'BOUNDS' (especifica los límites inferior y superior de los valores individuales)
- 'RANGES' (especifica las desigualdades dobles.)

**Software compatible:** los archivos MPS son compatibles con la mayoría de los programas comerciales de programación lineal, así como con el sistema de código abierto COIN-OR.

**Formatos relacionados:** Existe un formato MPS ampliado (XMPS) que añade soporte para programas lineales y enteros para incluir los PNL (Programas No Lineales), así como el formato [NL](#) que ha quitado mucha popularidad a los formatos MPS.