# AMPL

Iker[3]

October 17, 2022

## Contents

# 1 Introducción

AMPL (de A Mathematical Programing Lenguage) es un lenguaje de programación para la modelización algebráica de alto nivel orientado al cálculo matemático. Es software propietario con la posibilidad de adquisición de una versión demo gratuíta limitada.

En el presente manual buscamos dar unas pautas de inicio para la adquisición y empleo del programa.

# 2 Adquisición

Para adquirir el programa donde se procesarán los archivos podemos entrar en:

- Windows
- Linux
- MacOS

Se descargarán una serie de archivos y bibliotecas en un primer momento comprimidas (al menos si se usa el sistema operativo Linux). En dichos archivos tendremos dos ejecutables, uno será el `ampl`, el programa para usar AMPL desde la terminal, y el otro será `amplide`, la interfaz gráfica en la que se incluye un editor integrado y una consola de AMPL.[1]

[1] En caso de estar en sistema operativo Linux debemos ejecutar los programas desde la terminal, situándonos en la carpeta de los ejecutables para luego escribir `./ampl` o `./amplide`.

# 3 Interfaz Gráfica

La interfaz gráfica de AMPL (`amplide`) se encuentra en una subcarpeta denominada `amplide`. Dicha interfaz gráfica incluye un editor de texto(en Figura 1, recuadro de la derecha) muy útil al incluír resaltado de sintaxis para el lenguaje AMPL; una consola en la que ejecutar los comandos[2] propios de AMPL (en Figura 1, recuadro central); y un navegador de archivos para facilitar el trabajo (recuadro de la izquierda).

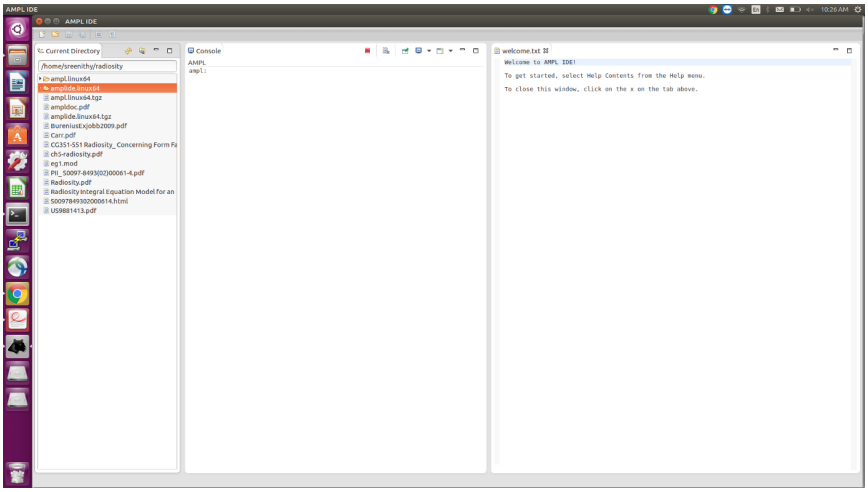


Figure 1: Interfaz gráfica de AMPL

En la interfaz gráfica tenemos ciertos atajos de teclado muy útiles como por ejemplo Ctl+r (presionar Control y la tecla **r** al mismo tiempo) que ejecuta el archivo seleccionado (ya sea en el navegador o en el editor) en la consola. Con hacer click una vez en un archivo del navegador de archivos o en un archivo abierto en el editor y presionar Ctl+r ejecutamos el archivo.

[2]Recordar **SIEMPRE** poner `punto y coma` al final de cada comando, de no hacerlo la consola esperará hasta que lo pongamos cambiando de `ampl:` a `ampl?`

# 4 Programas de AMPL

Los modelos que programaremos en AMPL se escriben en un archivo de extensión `.mod` (es recomendable escribirlos ya en el editor de la interfaz gráfica, así tendremos el resalado de sintaxis del lenguaje, i.e., palabras de colorines).

Antes de entrar en la explicación es preferible saber que para comentar en archivo de AMPL se emplea el numeral (#). Todo a la derecha de # queda comentado.

En un nivel muy básico podemos diferenciar 4 partes de un programa de AMPL:

## 4.1 Declaración de conjuntos y parámetros

### 4.1.1 Conjuntos

Los `conjuntos` son agrupaciones de datos que no se repiten (es decir, un conjunto podría ser: 1 2 3; pero no: 1 1 2, al repetirse el 1). Pueden ser números, letras o palabras y los declaramos con `set` como:

```
# set nombre_conjunto := {valor1, valor2, ...};
set conjunto1:= {1,2,3,4,5,6,8,9,10};
set conjunto2:= 1..10;
set conjunto3:= {comida, bebida, gasolina};
```

También podemos definir el conjunto en un archivo de extensión `.dat`:
En el `.mod` declaramos el conjunto

```
set conjunto;
```

Y en el `.dat` lo definimos:

```
set conjunto:= coste beneficio ganancia;
```

### 4.1.2 Parámetros

Los `parámetros` son números, vectores o matrices constantes (permanecen inalterados durante la ejecución del programa). Los declaramos con `param` como:

```
# param nombre_parametro:= valor;
param coste:= 50;
param b := 0.6;
```

Para declarar vectores y matrices es mejor usar los archivos `.dat` para no saturar el archivo `.mod`.

En el `.mod` declaramos el parámetro:

```
# param nombre_parametro{índices};
param cajas {1..10};
param gastos {indices_gastos};    # tenemos que definir índices_gastos
  # como un conjunto con set

param costes {i in indices_costes}; # lo mismo que el anterior
param almacenes {(i,j) in {1..20, indices_y}} # aquí tenemos una matriz,
      # los subíndices los
      # podemos definir de cualquiera
      # de las formas descritas arriba
```

Y en el `.dat` lo definimos:

```
# param nombre_parametro:= indice1 valor1 indice2 valor2 ...;
param cajas:= 1 3 2 10 3 23 4 45 5 29 ...; # Hasta los 30 valores.
   # Primero disponemos el
   # índice y luego el valor
   # de cajas[índice]
```

- si queremos ver mejor, como AMPL ignora los espacios en blanco, podemos escribir lo mismo como:

  ```
  param cajas:=
  1 3
  2 10
  3 23
  4 45
  5 29
  ...;
  ```

  Donde cajas[1] =3, cajas[5] =29, etc...

- Ahora, si por ejemplo $\text{indices}_{\text{gastos}}$ fuera un conjunto de palabras {comida, bebida, gas} podemos hacer lo mismo:

```
param gastos:=
comida 300
bebida 120
gas 3000;
```

- para las matrices, con el mismo ejemplo arriba declarado, y suponindo que $\text{indices}_{\text{y}} = \{\text{A1, A2}\}$. Tras el nombre de la matriz escribimos : y ponemos los nombres de las columnas, después := y el nombre de la primera fila seguido de tantos valores como columnas hay; el de la segunda fila con sus valores, etc...:

```
param almacenes:  1  2  3  4  5  6  7  8  9  10:=
     A1  0 34  9 100 0  5  29 2  8  56
     A2 13 45  87 3  12 59 24 46 93 11;
```

Donde almacenes[A1,4] =100, almacenes[A2,10] = 11, etc...

## 4.2 Declaración de variables

Las `variables` son lo que conocemos usualmente como variables en matemáticas, datos desconocidos que queremos resolver. En la sintaxis de AMPL se declaran con `var` como:

```
# var nombre_variable;
var X1;
var Y;
```

Contamos con algunas palabras para especificar el tipo de dato que es nuestra variable:

```
var x, integer; # variable entera
var y, binary;  # variable binaria, sólo toma como valores 0 o 1.
```

También podemos definir las variables con subíndices:

```
var x {i in 1..13};
var y {i in 1..5}, binary;
var z {(i,j) in {1..9,1..7}}, integer;
```

### 4.3 Función objetivo

Para declarar la función objetivo a modelar usamos `maximize` o `minimize` (dependiendo del objetivo del problema), seguido del nombre que le queremos poner a la función, dos puntos y la función en sí. Por ejemplo:

```
var x; var y;

minimize C: 2*x - y;
```

El nombre de la restricción es muy útil para cuando queramos ejecutar el modelo. En caso de error nos señalará la función/restricción/parámetro/conjunto que no funciona o que presenta errores por el nombre que les demos.

### 4.4 Restricciones

Las restricciones son, como su nombre indica, sentencias que restringen los valores de las variables. En AMPL se declaran con `subject to` (sujeto a) seguido del nombre de la restricción, dos puntos y la restricción en si.

```
subject to Res: x + y <= 10;
```

Podemos declarar varias restricciones empleando subíndices, los cuales tenemos que indicar antes de los dos puntos:

```
var x{i in 1..4};
subject to Res {i in 1..4}: x[i] >= 0; # todas las variables
       # x[i] son no negativas
```

Para simplificar expresiones se puede emplear el **sumatorio** (`sum`), cuya sintaxis es (suponemos que `n` y `m` son un conjunto):

```
sum {i in n} # expresión del sumatorio
sum {(i,j) in {n,m}}
```

Por ejemplo:

```
var x {i in 1..5}; var y {i in 1..5};
var z {(i,j) in {1..3,1..10}};

subject to Res1: sum {i in 1..5} x[i] + y[i] < 0;
subject to Res2: sum {(i,j) in {1..3,1..10}} z[i,j] >= 3;
```

Por supuesto, podemos combinar la declaración de restricciones con subínidces y los sumatorios.

Aquí un ejemplo en el que se incluyen parámetros (unitarios y vectores), conjuntos, variables (tanto enteras como binarias), sumatorios (de una y varias variables) y la declaración de restricciones con subíndices:

```
param m:= 10;
set n:=1..m;
var x {(i,j) in {n,1..5}}, integer;
var y {i in 1..5}, binary;
param b {i in 1..5} ;
param gasto {i in n};   # definimos los valores en un .dat
# (para el ejemplo no importan sus valores)

maximize C: sum{(i,j) in {n, 1..5}} x[i,j]; # la función objetivo es
    # la suma de todos los x[i,j]
subject to Res {j in 1..5}: sum {i in n} gasto[i]*x[i,j] <= b[j]*y[j];
subject to NonegX {i in n}: x[i,j] >= 0; # restricciones de no
 # negatividad para cada x[i,j]
```

# 5 Consola

La consola de AMPL puede ser convocada ya sea ejecutando el programa `ampl` previamente descargado junto al resto de archivos y lbrerías (el cual abrirá la terminal en caso de estar en Linux o un ejecutable que simula una terminal en caso de Windows) o bien en la interfaz gráfica de AMPL (`amplide`) siendo la región central del programa nada más lo abrimos.

## 5.1 Comandos de Consola

La consola consta simplemente de una pantalla y un prompt (línea de texto al lado del punto donde se introducen los comandos). El prompt tiene básicamente dos estados:

- `ampl:` : Para indicar que el programa está listo para que se introduzcan comandos.
- `ampl?` : AMPL está esperando a que completes el comando anterior introducido (generalmente es por no incluír punto y coma al final del comando)

Los comandos básicos para trabajar con AMPL en la consola son:

- `reset;` : reinicia la consola borrando los datos ya introducidos (muy útil usarlo tras cada ejecución de un programa para que no interfiera con el siguiente programa).
- `quit;` : sale de AMPL (en el caso de ejecutarlo en la interfaz gráfica ya no podremos emplear la consola hasta reiniciar AMPL).
- `model nombre-del-modelo.mod ;` [3]: modeliza el programa que hemos escrito en el archivo nombre-del-modelo.mod y, en caso de error de sintaxis, indica (de forma muy burda y errrática) donde se sitúa.
- `data nombre-archivo-datos.dat ;` [3]: incluye los datos al último programa modelizado.

[3] A la hora de poner los nombres de los archivos nos encontramos con dos problemas: en caso de estar en la interfaz gráfica tenemos que situar el navegador de archivos en la carpeta donde se encuentre el archivo a ejecutar; y si estamos en el programa `ampl` debebmos ejecutarlo en la carpeta donde se sitúen los archivos que ejecutaremos. Podemos evitar ésto poniendo la ruta absoluta al archivo.

- `include nombre-archivo-comandos.run ;` [3]: ejecuta todos los comandos incluídos en el archivo con extensión `.run`.
- `option` : modifica cómo se comporta AMPL. El más común al nivel de los ejercicios propuestos es:
  - `option solver nombre-solver ;` : modifica el solver a utilizar (`nombre-solver`) en la resolución del problema.
- `solve;` : soluciona el problema modelizado (no tiene mucho misterio). En caso de algún error en la ejecución lo deja indicado.
- `display nombre-de-variables ;` ; Imprime en la consola el valor de las variables que indiquemos (separadas por comas).

### 5.2 Archivos .run

Para evitar pérdidas de tiempo y esfuerzo reescribiendo una larga lista de comandos en la consola para ejecutar un programa podemos escribir dichos comandos en un archivo con extensión `.run` y luego ejecutar dicho archivo en la consola de AMPL con el comando `include` seguido del nombre del archivo.

Pongamos un ejemplo de un archivo `.run` de un modelo con sólo dos variables (ejecutaríamos el archivo en la consola de la interfaz gráfica):

```
reset;                  # eliminamos cualquier cosa de la memoria del programa
model ejemplo.mod;      # modelizamos el problema
data ejemplo.dat;       # incluímos los datos
option solver minos;    # cambiamos el solver por el MINOS
solve;                  # resolvemos el problema
display x1, x2;         # imprimimos por pantalla los valores de las variables
# quit; si quisiéramos salir de AMPL
```