# Introducción a SCIP y ZIMPL

Facundo Gutiérrez - Nazareno Faillace

Investigación Operativa - 2do. cuatrimestre 2018

---

## 1. Instalando y ejecutando SCIP

Es conveniente instalar SCIP en Linux. Para esto, dirigirse a la sección de descargas de la página de SCIP: `http://scip.zib.de/` . En esa sección se encuentran diversos instaladores de SCIP. Se explicará cómo instalar SCIP con el instalador `SCIPOptSuite-6.0.0-Linux.sh`.
Una vez descargado el archivo, moverlo a la carpeta donde se desea instalar. Hacer click derecho sobre el archivo, ir a Propiedades, luego a Permisos y marcar la casilla que dice *Permitir ejecutar el archivo como un programa*.
Abrir la consola en la ubicación del archivo y ejecutar el siguiente comando:

```
./SCIPOptSuite-6.0.0-Linux.sh
```

"Leer" los términos y condiciones. Finalmente nos hace dos preguntas, a las que responderemos con `y` las dos veces. Se creará en la ubicación del archivo una carpeta llamada `SCIPOptSuite-6.0.0-Linux` . Entrar en esa carpeta y luego entrar a la carpeta `bin`. Abrir la terminal en esta ubicación y ejecutar el comando:

```
./scip
```

Cada vez que se desee usar SCIP, debe dirigirse a la carpeta `bin` y ejecutar el comando `./scip` en la consola. Si aparece algún mensaje de error relacionado a que no encuentra librerías o paquetes, intentar ejecutando alguno (o ambos) de los siguientes comandos:

```
sudo apt-get install libatlas3-base
sudo apt-get install libgfortran3
```

## 2. SCIP - Correr archivo `.zpl`

Una vez que ejecutamos SCIP, leemos el archivo `.zpl` de la siguiente manera:

`read` *ruta/archivo.zpl*

Por ejemplo:

```
read /home/usuario/Documentos/problema.zpl
```

Luego, le pedimos a SCIP que lo resuelva con el comando `optimize`. Finalmente, para que SCIP nos muestre la solución óptima (si existe) escribimos `display solution`. También podemos guardar en un archivo a la solución, de la siguiente manera:

`write solution` *ruta/nombre del archivo*

Por ejemplo:

```
write solution /home/usuario/Documentos/solucion.txt
```

Con el comando `help` se puede ver la lista de comandos y sus descripciones.

## 3. Utilizando ZIMPL

### 3.1. Sintaxis y expresiones básicas

Manual de ZIMPL : `http://zimpl.zib.de/download/zimpl.pdf`
Cada línea debe terminar con `;`
Para comentar utilizamos el `#` : todo lo que venga después del `#` en la misma línea será ignorado.

Strings: se escriben entre comillas. Ej.: `"Bill Jobs"`
Operaciones: `a + b`, `a * b`, `a * *b`, `a/b` ... (Tablas 2 y 3 del manual)
Expresiones booleanas: $<=, <, ==, !=, >=, >$ . Se pueden combinar con `and`, `or` y `xor` y negar con `not`.
Ej.: `a + b <= 8  and  3 * a <= 12`

### 3.2. Tuplas y conjuntos

- Cada conjunto consiste de un número finito de tuplas con igual número de componentes.
- Los datos son delimitados por { y }
- Las tuplas son delimitadas por < y > Si la tupla tiene un sólo elemento, es posible obviar < >
- Ejemplos:

```
set A := {1, 2, 3} ;  # Se define el conjunto A
```

- Podemos utilizar operaciones de conjuntos (ver Tabla 4 del manual). Ejemplos:

```
set D:= A cross B ; # A×B (también se puede esribir como A*B)
set E:= A without B ; # A -B
```

### 3.3. Conjuntos condicionales y conjuntos indexados

Se pueden restringir un conjunto a las tuplas que cumplen cierta restricción booleana usando `with`. Ejemplos:

- `set F := { < i, j > in Q with i > j and i < 5} ;`
- `set A := {"x","y","z"} ;`
  `set B := {1,2,3} ;`
  `set V := {< a,2 > in A*B with a == "x" or a == "z"} ;`
  `# Da como resultado V = {< "x",2 >,< "z",2 >}`

Se puede indexar un conjunto a partir de otro conjunto. Para acceder a un conjunto indexado, se usan `[` y `]` (por ejemplo, `S[7]`). Ejemplos:

```
set I:= {1..3} ;
set A[I] := < 1 > {"a","b"}, < 2 > {"c","e"}, < 3 > {"f"} ; # A = {{"a","b"}, {"c","e"}, {"f"}}
set B[< i > in I] := {3 * i}  # B = {3,6,9}
```

### 3.4. Parámetros

- Pueden ser declarados de dos maneras: con o sin un conjunto indexado. Si no se indexa sobre un conjunto, el parámetro es simplemente un valor numérico o un string.
- El parámetro se declara con `param` seguido del nombre del parámetro y opcionalmente el conjunto sobre el que será indexado entre corchetes. Después del `:=` se escribe una lista de pares cuyo primer elemento es la tupla del conjunto de índices y el segundo elemento es el valor del parámetro para ese índice.
- Ejemplos:

```
param q:= 5 ;  # parámetro no indexado
set A:={3..10} ;
```

```
param U[A]:=< 5 > 14, < 8 > 19 default 11 ;
# en este caso U[5]=14, U[8]=19, U[4]=11 pero U[1], U[2], #U[11], etc. no están definidos.
param c[<i>in {1..10} with i mod 2 == 0] := 3*i
```

### 3.5. Sumas y comando `forall`

Forma general para la suma:

`sum` *index* `do` *term*

Donde *index* tiene la siguiente pinta:

*tuple* `in` *set* `with` *boolean-expression*

Está permitido usar `:` en vez de `do` y `|` (barra vertical) en vez de `with`. El `with` del *index* es opcional. Ejemplos:

```
sum <a> in A : u[a] * y[a]
sum <a,b,c> in C with a in E and b >3 : -a/2 * z[a,b,c]
```

La forma general de `forall` (*'para todo'*) es:

`forall` *index* `do` *term*

La estructura de *index* es la misma que para el comando `sum`. Ejemplo:

$\sum_{k \in K} k \cdot c_{ij} \quad \forall i \in X \, \forall j \in X$ se escribe como:

```
forall <i,j> in X cross X do sum <k> in K : k*c[i,j];
```

### 3.6. Comando `do print`

- El comando `do print` permite imprimir el valor numérico, string, expresión booleana, conjunto o tupla que le siga.
- Ejemplos:
```
set I :=  1..10 ;
do print I;
do print " Cardinality of I:", card(I);
do forall <i> in I with i > 5 do print sqrt(i)
```
- Es útil para chequear si, por ejemplo, un conjunto tiene los elementos que esperamos que tenga.

### 3.7. Variables

- Las variables pueden estar indexadas, como los parámetros.
- Pueden ser de tres tipos: `real, integer, binary` (default: `real`)
- Podemos especificar las cotas de las variables (default: $[0, +\infty)$). Las cotas pueden ser `infinity` y `-infinity`
- Ejemplos:
```
var x1;
var x2 binary;
var x3 integer >= -infinity;
var y[A] real >= 2 <= 18;
```

### 3.8. Objetivo

- Puede ser `maximize` o `minimize`

- Luego del objetivo se escribe el nombre, dos puntos y la expresión de la función objetivo.
- Ejemplos:

```
maximize profit: sum <i>in I : c[i] * x[i];
minimize cost: 12 * x1 -4.4 * x2 + 5 +
sum <a> in A : u[a] * y[a] +
sum <a,b,c>in C with a in E and b >3 : -a/2 * z[a,b,c];
```

### 3.9. Restricciones

- La forma general de una restricción es:

  `subto` *nombre*: *término* desigualdad/igualdad *término*

  Ejemplo: `subto time: 3 * x1 + 4 * x2 <= 7;`
- También puede tener la siguiente estructura (rango):

  `subto` *nombre: expr desigualdad término desigualdad expr*

  Ejemplo: `subto space: 50 >= sum <a>in A: 2 * u[a] * y[a] >= 5;`
- *nombre* puede ser cualquier string que comience con una letra. *término* es como el que se utiliza en el objetivo. *expr* es cualquier expresión válida que de como resultado un número.
- Ejemplos:

```
subto weird: forall <a>in A: sum <a,b,c>in C: z[a,b,c]==55;
subto c21: 6*(sum <i>in A: x[i] + sum <j>in B : y[j]) >= 2;
subto c40: x[1] == a[1] + 2 * sum <i>in A do 2*a[i]*x[i]*3+4;
```

### 3.10. Tabla de parámetros

Una tabla de parámetros está delimitada en los márgenes por | y necesita una primera fila con el índice de las columnas (que debe ser unidimensional) y una columna con el índice de las filas (que puede ser multidimensional). Cada columna se separa con coma. Ejemplos:

```
set I := { 1 .. 10 };
set J := { "a", "b", "c", "x", "y", "z" };

param h[I*J] :=   | "a", "c", "x", "z"   |
                |1|  12,  17, 99,     23 |
                |3|   4,   3,-17, 66*5.5 |
                |5| 2/3, -.4,  3, abs(-4)|
                |9|   1,   2,  0,      3 | default -99;

param g[I*I*I] :=     | 1, 2, 3 |
                    |1,3| 0, 0, 1 |
                    |2,1| 1, 0, 1 |;

param k[I*I] :=  |  7,  8,  9 |
               |4| 89, 67, 55 |
               |5| 12, 13, 14 |, <1,2> 17, <3,4> 99;
```

### 3.11. Inicializar parámetros y conjuntos desde un archivo

La sintaxis para cargar los valores de un parámetro o un conjunto desde un archivo es:

`read` *filename* `as` *template* `[skip` *n*`]` `[use` *n*`]` `[match` *s*`]` `[comment` *s*`]`

Donde *filename* es el nombre del archivo a leer. *template* es un string con la estructura de las tuplas que serán generadas. Cada linea del archivo está separada en campos. Los delimitadores de los campos pueden ser: espacio, tabulación, coma, punto y coma o dos puntos (:).
Hay varios modificadores para *template* y pueden ser utilizados sin importar el orden. Para esta clase, el que resulta más interesante es `skip`, que permite ignorar las primeras $n$ líneas del archivo.

Ejemplos:
Se inicializa el conjunto P a partir de la lectura de `notes.txt`.

```
set P := { read "nodes.txt" as "<1s>" };
nodes.txt:
   Hamburg            → <"Hamburg">
   München            → <"München">
   Berlin             → <"Berlin">
```

`<1s>` indica que se lee el primer campo y se lo interpreta como un string.

```
set Q := { read "blabla.txt" as "<1s,5n,2n>" skip 1 use 2 };
blabla.txt:
   Name;Nr;X;Y;No      → skip
   Hamburg;12;x;y;7    → <"Hamburg",7,12>
   Bremen;4;x;y;5      → <"Bremen",5,4>
   Berlin;2;x;y;8      → skip
```

Según el *template*, las tuplas estarán formadas por el primer campo interpretado como string, el quinto interpretado como un valor numérico y luego el segundo, interpretado también como valor numérico. Además, se omite la primera línea (`skip 1`) y se utilizan sólo las primeras 2 líneas a partir de que comienza la lectura del documento (`use 2`)

```
param cost[P] := read "cost.txt" as "<1s> 2n" comment "#";
cost.txt:
   # Name Price        → skip
   Hamburg 1000        → <"Hamburg"> 1000
   München 1200        → <"München"> 1200
```

En este caso, se puede inicializar el parámetro `cost` a partir del conjunto P utilizando un template adecuado. Además, el modificador `comment` indica que toda línea del documento que comience con `#` debe ser omitida.