Departamento de Engenharia Informática e de Sistemas

Metodologias de Otimização e Apoio à Decisão

Resolução de problemas de PL em Python

- Parte I -

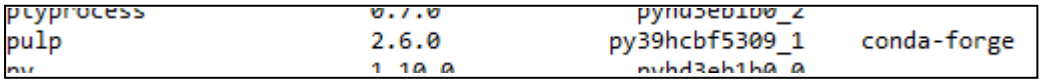
1. Programação linear e Python

Para resolver modelos de programação linear (PL) em Python, vamos recorrer à biblioteca **PuLP**. De entre as várias bibliotecas *open-source* de Python especializadas em interagir com *solvers* de programação linear, esta foi a selecionada pela simplicidade com que permite definir um dado problema.

Basicamente, a biblioteca PuLP permite descrever um problema de otimização como um modelo matemático, invocar um *solver* externo (CBC, GLPK, CPLEX, Gurobi, …) para resolver esse modelo e, em seguida, usar comandos em Python para manipular e mostrar a solução.

Para instalar esta biblioteca, aceder a https://anaconda.org/conda-forge/pulp e escolher uma das opções de instalação. Uma possibilidade é executar `conda install -c conda-forge pulp` a partir da consola do Spyder.

Para verificar se a instalação foi bem sucedida, na mesma consola digitar `conda list` e constatar que *pulp* já consta da lista de pacotes.



Seguem-se alguns exemplos que ilustram a resolução de problemas básicos de PL, em Python, recorrendo à biblioteca PuLP.

EXEMPLO 1

Considere-se o exemplo do fazendeiro (retirado do Capítulo 2 de Investigação Operacional):

"*Um fazendeiro deseja otimizar as plantações de arroz e milho da sua quinta, ou seja, quer saber que áreas deve plantar de arroz e milho de modo a ser máximo o lucro obtido das plantações.*

O lucro por unidade de área plantada de arroz e de milho é de, respetivamente, 5 e 2 unidades monetárias (UM).

As áreas plantadas de arroz e milho não devem ser maiores que 3 e 4 unidades de área, respetivamente.

O consumo total de mão-de-obra (medido em homens/hora) nas duas plantações não deve ser maior do que 9. Cada unidade de área plantada de arroz necessita de 1 homem/hora e cada unidade de área plantada de milho necessita de 2 homens/hora."

O modelo matemático de programação linear é o seguinte:

Maximizar $z = 5\,x_1 + 2\,x_2$	(Lucro a obter das plantações)
sujeito a	
$x_1 \leq 3$	(Área para plantação de arroz)
$x_2 \leq 4$	(Área para plantação de milho)
$x_1 + 2\,x_2 \leq 9$	(Mão-de-obra)
$x_1 \geq 0,\ x_2 \geq 0$	

O processo de modelação em PuLP envolve os seguintes passos:

- Inicializar o modelo
- Definir as variáveis de decisão
- Definir a função objetivo
- Definir as restrições
- Resolver o modelo

Para iniciar este processo escrever o código que se segue no editor do Spyder.

O início do ficheiro deve ser destinado a uma pequena secção de comentários descrevendo o propósito do programa. Por exemplo:

```
"""
Problema do fazendeiro / Farmer problem
"""
```

Depois, há que importar as classes e funções da PuLP que se vão utilizar:

```
from pulp import LpMaximize, LpProblem, LpStatus, lpSum, LpVariable
```

Alternativamente, pode optar-se por importar todas as classes e funções:

```
from pulp import *
```

Seguidamente, há que inicializar uma instância da classe *LpProblem* para representar o modelo:

```
# Criar modelo / Create model
model=LpProblem("Fazendeiro",LpMaximize)
```

O primeiro parâmetro (*name*) é uma *string* que serve para identificar o problema. O segundo parâmetro (*sense*) varia entre *LpMinimize* ou *LpMaximize* (que são constantes), dependendo do tipo de otimização em causa.

Depois de criado o modelo, há que definir as variáveis de decisão como instâncias da classe *LpVariable* que tem quatro parâmetros: *name*, *lowBound*, *upBound* e *cat*. O primeiro é uma *string* que define o nome da variável. O segundo e terceiro representam os limites inferior e superior da variável, cujos valores por omissão, são $-\infty$ e $+\infty$, respetivamente. O quarto parâmetro representa a categoria da variável, podendo variar entre *LpContinuous* ou *LpInteger*, sendo, por omissão, *LpContinuous*.

No exemplo do fazendeiro, como as variáveis têm restrição de não-negatividade, é apenas necessário indicar um limite inferior igual a 0. O terceiro e o quarto parâmetros podem omitir-se porque assumem os valores padrão.

```
# Criar variáveis de decisão / Create decision variables
x1=LpVariable("x1",lowBound=0)
x2=LpVariable("x2",lowBound=0)
```

O próximo passo consiste em criar a função objetivo e as restrições, atribuindo-as ao modelo. Não é necessário criar listas ou matrizes, bastando escrever as expressões em Python e usar o operador **+=** para anexá-las ao modelo.

```
# Adicionar função objetivo ao modelo / Add objective function to model
model+=5*x1 + 2*x2

# Adicionar restrições ao modelo / Add constraints to model
model+= x1 <= 3,"Arroz/Rice"
model+= x2 <= 4,"Milho/Corn"
model+= x1 + 2*x2 <= 9,"Mao-de-obra/Labor"
```

Nota: A utilização do operador **+=** para esta finalidade só é possível, porque a sua classe, *LpProblem*, implementa o método especial *.__iadd__()* que especifica o comportamento desse operador.

As *strings* colocadas a seguir à virgula, quer na função objetivo, quer nas restrições, dão uma breve indicação do que estas representam. Atenção! O uso de alguns caracteres proibidos (tais como: ~, ç, ´,`), originam um erro.

Para problemas maiores, pode ser preferível usar a função *lpSum()* (que calcula a soma de uma lista de expressões lineares), do que repetir o operador **+**. Por exemplo, a adição da função objetivo anterior, ao modelo, poderia ter sido feita com a instrução seguinte:

```
# Adicionar função objetivo ao modelo / Add objective function to model
model+=lpSum([5*x1,2*x2])
```

Neste momento, já é possível ver a definição completa do modelo criado, digitando o nome do modelo na consola, ou adicionando a instrução *print(model)* ao código anterior. O resultado surgirá em seguida:

```
Fazendeiro:
MAXIMIZE
5*x1 + 2*x2 + 0
SUBJECT TO
Arroz/Rice: x1 <= 3

Milho/Corn: x2 <= 4

Mao_de_obra/Labor: x1 + 2 x2 <= 9

VARIABLES
x1 Continuous
x2 Continuous
```

Também é possível guardar o modelo num ficheiro *.mps* para uso posterior. Este tipo de ficheiro é criado no formato Mathematical Programming System (MPS), sendo especificamente usado para problemas de programação linear. Os modelos são guardados como texto ASCII com os parâmetros em formato de colunas. Na código que se segue, o método *writeMPS()* cria o ficheiro cujo nome é passado como argumento, por omissão, na mesma pasta do projeto Python.

```
# Guarda modelo num ficheiro MPS / Save model in a MPS file
model.writeMPS("Farmer.mps")
```

Também é possível fornecer o *link* completo.

```
# Guarda modelo num ficheiro MPS / Save model in a MPS file
model.writeMPS("C:\\Users\\Teresa\\Arquivos\\MOAD\\AULAS PRÁTICAS\\PYTHON\\Farmer.mps")
```

Para recuperar o modelo, usa-se a instrução seguinte.

```
# Recuperar modelo de um ficheiro MPS / Retrieve model from a MPS file
var,model=LpProblem.fromMPS("Farmer.mps",LpMaximize)
```

Note-se que na recuperação do modelo foi necessário indicar que se trata de um problema de maximização (*LpMaximize*), uma vez que essa informação não é guardada no ficheiro. Por omissão, é considerado um problema de minimização.

A partir do momento em que existe um modelo criado, este pode ser resolvido por um *solver*. Existem algumas funções auxiliares que permitem consultar os *solvers* possíveis de usar na PuLP e os que se encontram atualmente disponíveis.

```
In [18]: listSolvers()
Out[18]:
['GLPK_CMD',
 'PYGLPK',
 'CPLEX_CMD',
 'CPLEX_PY',
 'GUROBI',
 'GUROBI_CMD',
 'MOSEK',
 'XPRESS',
 'PULP_CBC_CMD',
 'COIN_CMD',
 'COINMP_DLL',
 'CHOCO_CMD',
 'MIPCL_CMD',
 'SCIP_CMD']

In [19]: listSolvers(onlyAvailable=True)
Out[19]: ['GLPK_CMD']
```

Para resolver o modelo definido anteriormente:

```
# Resolver modelo / Solve model
model.solve()
```

Como neste caso os parenteses após o *solve*() são deixados vazios, será usado o *solver* escolhido pela PuLP (por omissão). Neste caso, o GLPK_CMD (único disponível). Para confirmar que foi este o *solver* usado:

```
In [32]: model.solver
Out[32]: <pulp.apis.glpk_api.GLPK_CMD at 0x2a4222f9700>
```

Alternativamente, ao resolver o modelo, pode optar-se por especificar o *solver* a usar:

```
# Resolver modelo / Solve model
solver=getSolver("GLPK_CMD")
model.solve(solver)
```

Finalmente, há que apresentar os resultados da resolução:

```
# Visualizar resultados / Visualize results
print("-------- Resultados / Results ---------")
print(f"Status = {model.status} <=> {LpStatus[model.status]}")
print(f"z* = {value(model.objective)}")
for var in model.variables():
    print(f"{var.name}* = {var.value()}")
for name,constraint in model.constraints.items():
    print(f"{name}: {constraint.value()}")
```

Na sequência do bloco de instruções anterior, surgirá na consola:

```
-------- Resultados / Results ---------
Status= 1 <=> Optimal
z* =  21.0
x1 = 3.0
x2 = 3.0
Arroz/Rice: 0.0
Milho/Corn: -1.0
Mao_de_obra/Labor: 0.0
```

Interpretação dos resultados:

Os valores obtidos para as variáveis de decisão e função objetivo, significam que o fazendeiro deverá plantar **3** unidades de área de arroz e **3** unidades de área de milho, de forma a conseguir um lucro máximo de **21** unidades monetárias. Os valores **0** correspondentes às 1ª e 3ª restrições, significam que os recursos (área de arroz e mão-de-obra) serão usados na totalidade. O valor **-1** correspondente à 2ª restrição, significa que uma unidade do recurso disponível (área para plantação de milho) não será utilizada.

Analisemos, em seguida, as instruções do código anterior:

```
print(f"Status = {model.status} <=> {LpStatus[model.status]}")
```

Esta instrução imprime o valor do *status* (atributo do objeto *model*) e em seguida imprime a descrição desse *status* acedendo ao dicionário *LpStatus* e usando como chave o valor do próprio *status*. Os pares chave-valor deste dicionário são: **1** – ***Optimal***: existe solução ótima e foi encontrada; **0** – ***Not solved***: é o valor por omissão antes de o problema ter sido resolvido; **-1** – ***Infeasible***: o problema não tem soluções admissíveis, ou seja, é impossível; **-2** – ***Unbounded***: solução ótima no infinito; **-3** – ***Undefined***: solução admissível não foi encontrada (mas pode existir).

```
print(f"z* = {value(model.objective)}")
```

A instrução anterior mostra o valor ótimo da função objetivo, aplicando a função v*alue()* ao atributo *objective* do objeto *model*. Se não se aplicasse esta função, seria mostrada a expressão da função objetivo e não o seu valor.

```
for var in model.variables():
    print(f"{var.name}* = {var.value()}")
```

O método .*variables()* da classe *LpProblem* devolve a lista de variáveis do problema (objetos da classe *LpVariable*). Assim, para todas as variáveis dessa lista, imprime o nome e respetivo valor ótimo, através do parâmetro *name* e do método .*value()*.

```
for name,constraint in model.constraints.items():
    print(f"{name}: {constraint.value()}")
```

O atributo *constraints* do objeto *model*, é um dicionário de restrições (objetos da classe *LpConstraint*) indexado pelo nome da restrição. O método .*items()*, devolve uma lista de pares (*chave*,*valor*) desse dicionário (neste caso, a *chave* é o *nome da restrição* e o *valor* é a *expressão linear* correspondente). Ao usarmos duas variáveis no ciclo *for*, em cada iteração a primeira assumirá o *nome da restrição* e a segunda, a *expressão linear*. Para obter o valor desta última, recorre-se ao método .*value()*.