Imanol Rojas Pérez

TOML

Programming exercises.

1. **Consider the following optimization exercise:**

$$\textbf{Minimize: } e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1)$$

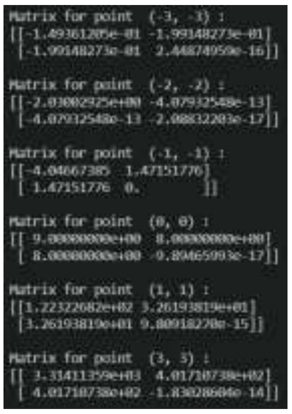
$$\textbf{Subject to: } \begin{array}{l} x_1x_2 - x_1 - x_2 \leq -1.5 \\ -x_1x_2 \leq 10 \end{array}$$

$$\textbf{Variables: } x_1, x_2$$

a. Identify whether is convex or not

First, the domain of the function must be analyzed. In this case, the domain is convex because any linear combination of two points of the function is a point inside of the function. We could say that the function does not have "holes" between two or more points where linear combinations of those points do not exist. In this problem this can be seen in the graph of the function (next section).

Due to the second order condition, the hessian of the objective function must be positive semidefinite at every real point existing. Next image shows the result of the Hessian matrix substituting by different points.

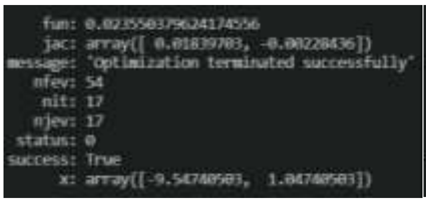


```
Matrix for point  (-3, -3) :
[[-1.49361205e-01 -1.99148273e-01]
 [-1.99148273e-01  2.44874959e-16]]

Matrix for point  (-2, -2) :
[[-2.03002925e+00 -4.07932548e-13]
 [-4.07932548e-13 -2.08832203e-17]]

Matrix for point  (-1, -1) :
[[-4.04667385  1.47151776]
 [ 1.47151776  0.        ]]

Matrix for point  (0, 0) :
[[ 9.00000000e+00  8.00000000e+00]
 [ 8.00000000e+00 -9.89465993e-17]]

Matrix for point  (1, 1) :
[[1.22322682e+02 3.26193819e+01]
 [3.26193819e+01 9.80918270e-15]]

Matrix for point  (3, 3) :
[[ 3.33411359e+03  4.01710738e+02]
 [ 4.01710738e+02 -1.83028604e-14]]
```

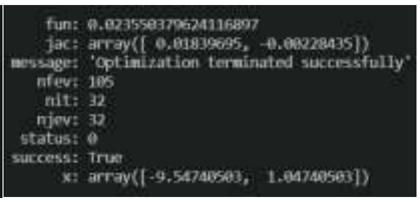
As it can be seen, for different points we have different Hessians with different values. Also, some of the matrices are not positive semidefinite, this means that the problem we ate looking at, is not convex.

b. Find the minimum. Use as initial points x_0 [0,0], [10,20], [-10,1], [-30,-30] and explain the difference in the solution if any. Choose which ones give an optimal point/value and give how long take 1 to converge to a result. Plot the objective curve and see whether the plot helps you to understand the optimization problem and results.

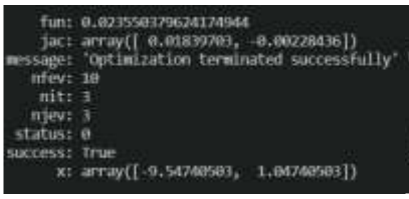
The next images show the different solutions found by the solver when changing the starting points.



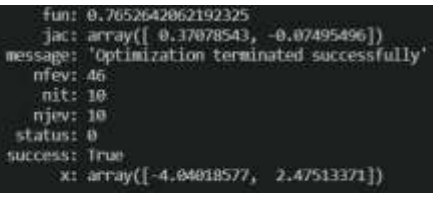
```
     fun: 0.023550379624174556
     jac: array([ 0.01839703, -0.00228436])
 message: 'Optimization terminated successfully'
    nfev: 54
     nit: 17
    njev: 17
  status: 0
 success: True
       x: array([-9.54740503,  1.04740503])
```



```
     fun: 0.023550379624116897
     jac: array([ 0.01839695, -0.00228435])
 message: 'Optimization terminated successfully'
    nfev: 105
     nit: 32
    njev: 32
  status: 0
 success: True
       x: array([-9.54740503,  1.04740503])
```



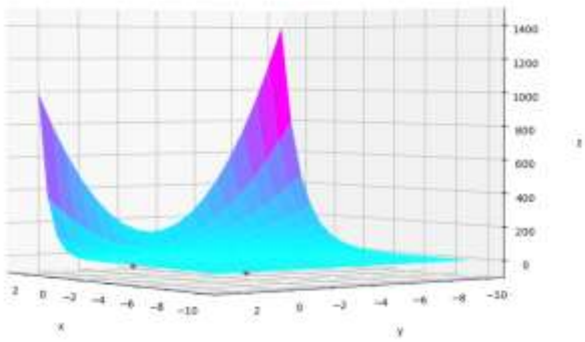
```
     fun: 0.023550379624174944
     jac: array([ 0.01839703, -0.00228436])
 message: 'Optimization terminated successfully'
    nfev: 10
     nit: 3
    njev: 3
  status: 0
 success: True
       x: array([-9.54740503,  1.04740503])
```



```
     fun: 0.7652642062192325
     jac: array([ 0.37078543, -0.07495496])
 message: 'Optimization terminated successfully'
    nfev: 46
     nit: 10
    njev: 10
  status: 0
 success: True
       x: array([-4.04018577,  2.47513371])
```

As it can be seen, there is more than one solution for the model. The first three starting points ([0,0], [10,20] and [-10,1]) have one solution, the last point ([-30,-30]) has a different solution. This means that the optimization exercise is not convex. To be convex, the solution must be unique no matter if the solver starts from different initial guesses. The fact that it has more than one solution means that it has local minimums and a global minimum. Also, the number of function evaluations depends on how close the starting point is from the next minimum, the further from the minimum, the longer it takes to find the solution.

The next plot shows how the solutions are disposed on the plane that the model conforms.



Three different solutions can be seen in the plot corresponding to the images seen previously in this same section.

c. Give as input the Jacobian (exact gradient) to the method and repeat and check whether the method speeds up.

The next table shows how many iterations does the solver do to converge to a solution. The most important column is the Function evaluations column.

	Number of iterations		**Function evaluations**		**Gradient Evaluations**	
Initial Point	**Normal**	**Jacobian**	**Normal**	**Jacobian**	**Normal**	**Jacobian**
(0, 0)	17	17	54	20	17	17
(10, 20)	32	27	105	40	32	27
(-10, 1)	3	3	10	4	3	3
(-30, -30)	10	44	46	110	10	43

When the Jacobian is given to the solver, it needs less function evaluations to find the solution. This happens because it does not have to calculate the gradient. In a normal case the gradient is calculated at every step, this supposes more function evaluations. Giving the Jacobian to the solver makes faster the process of solving the problem.

2. Consider the problem of fining x1 and x2 that solves the following optimization exercise:

$$\textbf{Minimize: } x_1^2 + x_2^2$$

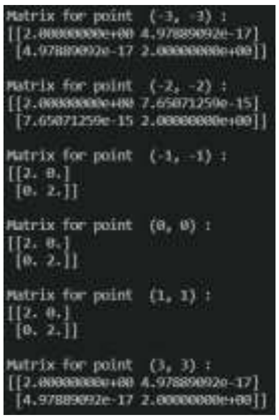
$$\textbf{Subject to: } \left\{ \begin{array}{l} 0.5 \le x_1 \\ -x_1 - x_2 + 1 \le 0 \\ -x_1^2 - x_2^2 + 1 \le 0 \\ -9x_1^2 - x_2^2 + 9 \le 0 \\ -x_1^2 + x_2 \le 0 \\ -x_2^2 + x_1 \le 0 \end{array} \right\}$$

$$\textbf{Variables: } x_1, x_2$$

a. Identify whether is convex or not

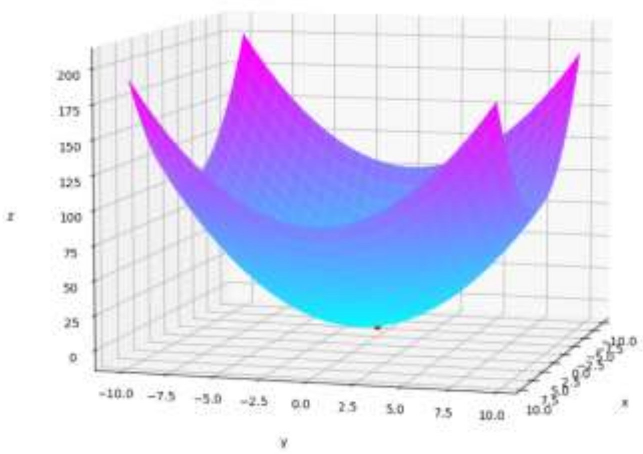
As in the last exercise, the domain of the function is convex for the same reasons. The linear combinations of any two points of the function give a point that is also inside the function.

Doing the same as in the last exercise, we have that the hessian for different points is:



```
Matrix for point  (-3, -3) :
[[2.00000000e+00 4.97889092e-17]
 [4.97889092e-17 2.00000000e+00]]

Matrix for point  (-2, -2) :
[[2.00000000e+00 7.65071259e-15]
 [7.65071259e-15 2.00000000e+00]]

Matrix for point  (-1, -1) :
[[2. 0.]
 [0. 2.]]

Matrix for point  (0, 0) :
[[2. 0.]
 [0. 2.]]

Matrix for point  (1, 1) :
[[2. 0.]
 [0. 2.]]

Matrix for point  (3, 3) :
[[2.00000000e+00 4.97889092e-17]
 [4.97889092e-17 2.00000000e+00]]
```

The hessian does not change with the points and is symmetric positive definite thus, the problem is convex in this case. This also can be seen if the solutions are represented in a graph:



The problem has a single solution and has a convex form.

b. Propose an initial point that is not feasible and an initial point that is feasible and check what happens with SLSQP as method.

The feasible set of points is the set that contains all the points inside the constraints that give a solution to the problem. The best feasible point is [1,1] because it is the closest to the solution. If we choose [1,1] as the initial point, the solution is reached faster.

The point [0,0] is not between the boundaries of the constraints of the model so the method cannot calculate the solution for the problem. This point is infeasible point.

c. Repeat giving as input the Jacobian. Check the convergence time (or number of steps).

To make a good comparison, three initial points are going to be compared. Those points are the following: [3,3], [2,2] and [1,1]. The next table shows the evaluations and the iterations of the method to find the optimal solution of the problem. The *Function evaluations* column shows the difference in terms of steps of the problem when the Jacobian is not given (Normal) and when the Jacobian is given (Jacobian).

	Number of iterations		**Function evaluations**		**Gradient Evaluations**	
Initial Point	**Normal**	**Jacobian**	**Normal**	**Jacobian**	**Normal**	**Jacobian**
(3, 3)	7	7	32	10	7	7
(2, 2)	6	6	18	6	6	6
(1, 1)	1	1	3	1	1	1

As mentioned before, if the Jacobian is given to the solver, the function evaluations are reduced.

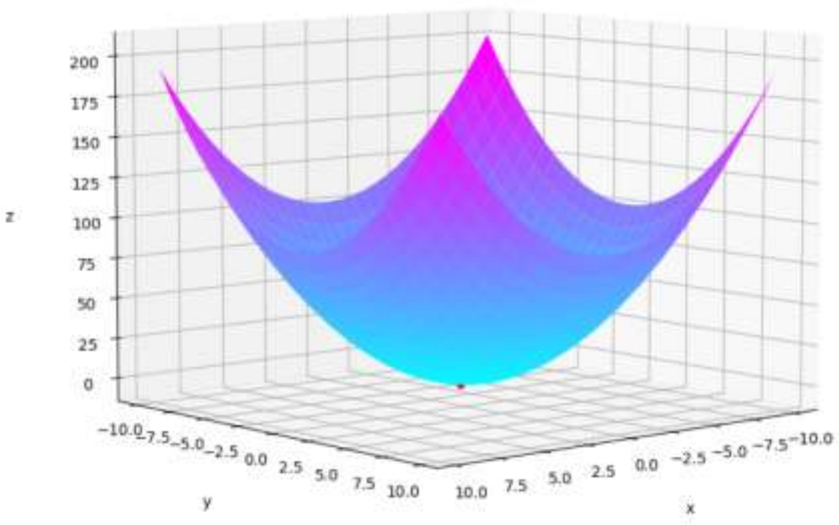
3. Consider the following non-linear problem. Say whether is a COP and solve it using the scipy library. Check convergence. Learn how to use the CVX toolbox to program it.

Minimize: $x_1^2 + x_2^2$

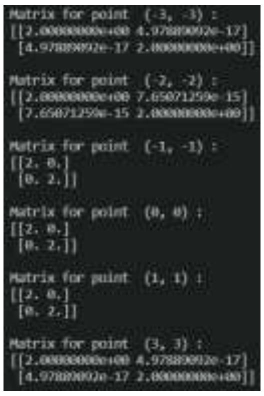
3Subject to: $$\begin{aligned} x_1^2 + x_1x_2 + x_2^2 &\le 3 \\ 3x_1 + 2x_2 &\ge 3 \end{aligned}$$

Variables: x_1, x_2

The domain of the function is convex, any combination of two poins of the function gives another point that is in the function. There is only one optimal solution. This can be seen in the next graph.



Also, if the Hessian is analyzed, the conclusion is that it is positive definite and the same for different initial points. This ensures that the problem is convex.



```
Matrix for point  (-3, -3) :
[[2.00000000e+00 4.97889092e-17]
 [4.97889092e-17 2.00000000e+00]]

Matrix for point  (-2, -2) :
[[2.00000000e+00 7.65071259e-15]
 [7.65071259e-15 2.00000000e+00]]

Matrix for point  (-1, -1) :
[[2. 0.]
 [0. 2.]]

Matrix for point  (0, 0) :
[[2. 0.]
 [0. 2.]]

Matrix for point  (1, 1) :
[[2. 0.]
 [0. 2.]]

Matrix for point  (3, 3) :
[[2.00000000e+00 4.97889092e-17]
 [4.97889092e-17 2.00000000e+00]]
```

4. Consider the following optimization problem. Use the CVX toolbox to program it.

Minimize: $x^2 + 1$

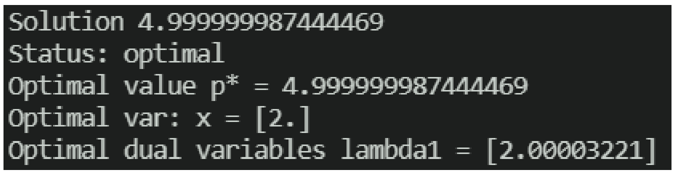
Subject to: $(x-2)(x-4) \leq 0$

Variables: x

The next code implements the model of this exercise using the CVXPY library. As in the last exercise, first, the variables of the model are defined, then the only constraint that the problem has is defined in DCP format. Finally

```
from cvxpy import *

x = Variable(1, name = "x") # Definition of the variable x


f1 = power(x, 2) - 6*x + 8 # Definition of the constraint that the model
has
constraints = [f1 <= 0]

f0 = power(x, 2) + 1 # Definition of the Objective function
obj = Minimize(f0)

prob = Problem(obj, constraints) # Creation of the model

print("Solution " + str(prob.solve()))  # Returns the results
print("Status: " + str(prob.status))
print("Optimal value p* = " + str(prob.value))
print("Optimal var: x = " + str(x.value))
print("Optimal dual variables lambda1 = " +
str(constraints[0].dual_value))
```

When the code is executed, the following results are obtained:



```
Solution 4.999999987444469
Status: optimal
Optimal value p* = 4.999999987444469
Optimal var: x = [2.]
Optimal dual variables lambda1 = [2.00003221]
```

These results agree with the solutions given on the exercises guide. There is only one lambda due that we have just one constraint in the model.

5. **Consider the following optimization problem. Use the CVX toolbox to program it.**

$$\textbf{Minimize: } x_1^2 + x_2^2$$

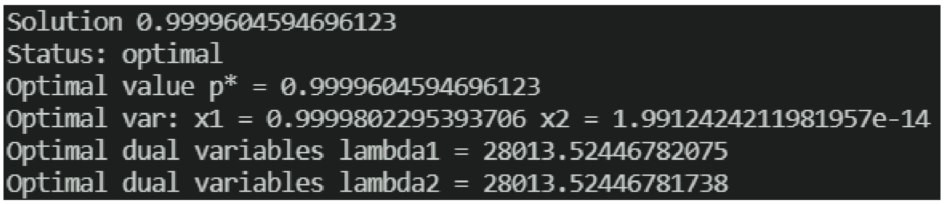
$$\textbf{Subject to: } \begin{array}{l} (x_1 - 1)^2 + (x_2 - 1)^2 \le 1 \\ (x_1 - 1)^2 + (x_2 + 1)^2 \le 1 \end{array}$$

$$\textbf{Variables: } x_1, x_2$$

The next code implements the model of the exercise. The implementation is the same as in the last exercise but with another constraint.

```python
from cvxpy import *
import numpy as np

x = Variable(2, name = "x")

f1 = power(x[0] - 1,2) + power(x[1] - 1, 2) # Define both constraints of
the problem
f2 = power(x[0] - 1,2) + power(x[1] + 1, 2)

constraints = [f1 <= 1., f2 <= 1.]

f0 = power(x[0], 2) + power(x[1], 2) # Define the objective function
obj = Minimize(f0)

prob = Problem(obj, constraints) # Create the model of the problem

print("Solution " + str(prob.solve()))  # Returns the optimal value.
print("Status: " + str(prob.status))
print("Optimal value p* = " + str(prob.value))
print("Optimal var: x1 = " + str(x[0].value) + " x2 = " +
str(x[1].value))
print("Optimal dual variables lambda1 = " +
str(constraints[0].dual_value))
print("Optimal dual variables lambda2 = " +
str(constraints[1].dual_value))
```

The results will have two lambdas in this case because the problem has two constraints. The results are the following:



```
Solution 0.9999604594696123
Status: optimal
Optimal value p* = 0.9999604594696123
Optimal var: x1 = 0.9999802295393706 x2 = 1.9912424211981957e-14
Optimal dual variables lambda1 = 28013.52446782075
Optimal dual variables lambda2 = 28013.52446781738
```

These results agree with the ones given in the exercises guide.

6. **The objective of this exercise is to test how a descent gradient algorithm works. Let us assume that we want to minimize an objective function f(x) without constraints. We start with a random point on the function and move in the negative direction of the gradient of the function to reach the local/global minima. It is to say, remember that the gradient descent algorithm assumes that:**

$$x^{(k+1)} = x^{(k)} + t\Delta x = x^{(k)} + t\nabla f\left(x^{(k)}\right)$$

Make a program in python in which you calculate the Gradient Descent Method using the Backtracking Line Search. Make the program to work with two simple examples:

- $f(x) = 2x^2 - 0.5$**, with initial point x0 = 3, with accuracy criterion of** $\eta = 10^{-14}$**. Give the result, the final accuracy, and the number of steps.**
- $f(x) = 2x^4 - 4x^2 + x - 0.5$**, with initial points x0 = -2, -0.5, 0.5 and 2 with accuracy criterion of** $\eta = 10^{-14}$**. Give the result, the final accuracy, and the number of steps.**
- **Repeat the previous exercise using Newton's Method. Compare the number of steps and time to find the solution.**

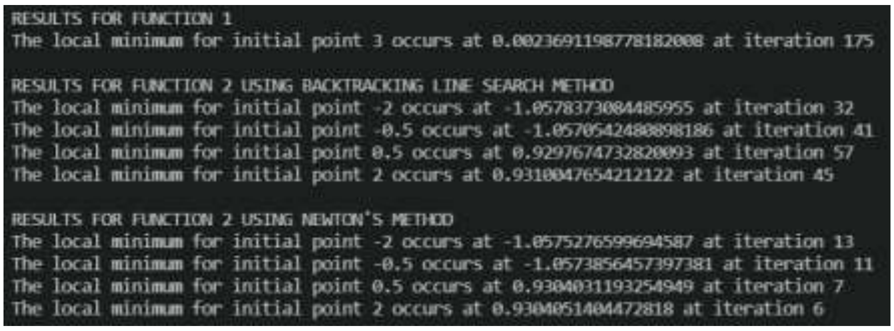
The next code implements the gradient descent method algorithm:

```
def GradientDescendMethod(cur_x, rate, precision, previous_step_size,
max_iters, func):
    import numdifftools as nd
    initial = cur_x # Save the initial point to print it later
    iters = 0 #iteration counter
    df = nd.Gradient(func) #Gradient of our function
    while previous_step_size > precision and iters < max_iters:
        prev_x = cur_x #Store current x value in prev_x
        cur_x = cur_x - rate * df(prev_x) #Grad descent
        previous_step_size = abs(cur_x - prev_x) #Change in x
        iters = iters+1 #iteration count

    print("The local minimum for initial point", initial, "occurs at",
cur_x, "at iteration", iters)
```

The next code implements the newtons method to find the result of the optimization. In this case the Scipy library has been used to have

```
def newton_method(x0):
    g= newton(func=f_1,x0=x0,fprime=f_2, tol=0.0001, full_output=True) #
Newtons method using the library
    r= g[1]
    print("The local minimum for initial point", x0, "occurs at", r.root,
"at iteration", r.iterations)
```

The results are the following ones:



```
RESULTS FOR FUNCTION 1
The local minimum for initial point 3 occurs at 0.0023691198778182008 at iteration 175

RESULTS FOR FUNCTION 2 USING BACKTRACKING LINE SEARCH METHOD
The local minimum for initial point -2 occurs at -1.0578373084485955 at iteration 32
The local minimum for initial point -0.5 occurs at -1.0570542480898186 at iteration 41
The local minimum for initial point 0.5 occurs at 0.9297674732820093 at iteration 57
The local minimum for initial point 2 occurs at 0.9310047654212122 at iteration 45

RESULTS FOR FUNCTION 2 USING NEWTON'S METHOD
The local minimum for initial point -2 occurs at -1.0575276599694587 at iteration 13
The local minimum for initial point -0.5 occurs at -1.0573856457397381 at iteration 11
The local minimum for initial point 0.5 occurs at 0.9304031193254949 at iteration 7
The local minimum for initial point 2 occurs at 0.9304051404472818 at iteration 6
```

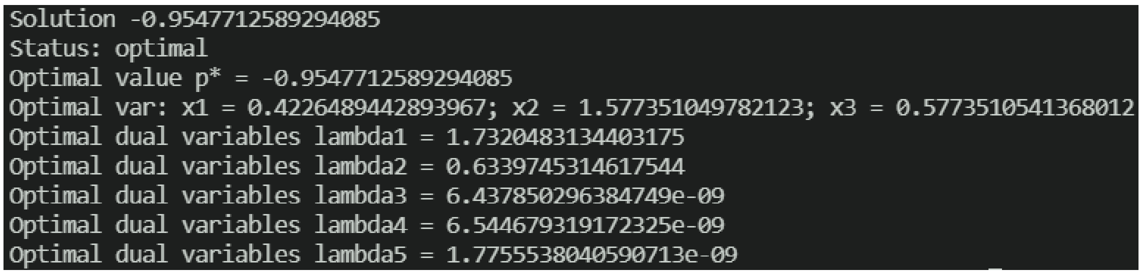
The Newton’s method arrives to the same results but it does it faster than the gradient descent method.

7. **Consider a Network Utility problem such as the one given at class, slide 12 of Topic 12, in which 3 sources and 5 links with capacity $(C_1, C_2, C_3, C_4, C_5) = (1, 2, 1, 2, 1)$ respectively are considered and solve it using CVX. Source 1 traverses link 1 and 2, source 2 traverses link 2 and source 3 traverses link 1 and 5. Also obtain the Lagrange multipliers.**

Next code implements the problem of optimizing the Network Utility.

```python
from cvxpy import *

x = Variable(3, name = "x")
c = [1, 2, 1, 2, 1] # Channel capacity array

f1 = x[0] + x[2] # x1+x3
f2 = x[0] + x[1] # x1+x2
f3 = x[2] # x3

constraints = [f1 <= c[0], f2 <= c[1], f3 <= c[4], x[0] >= 0, x[1] >= 0, x[2] >= 0] # Define all constraints

f0 = 0
for i in range(3):
    f = cvxpy.log(x[i])
    f0 = f0 + f

obj = Maximize(f0) # Define the objective function

prob = Problem(obj, constraints) # Create the model

print("Solution " + str(prob.solve()))  # Returns the solutions and the lagrange multipliers
print("Status: " + str(prob.status))
print("Optimal value p* = " + str(prob.value))
print("Optimal var: x1 = " + str(x[0].value) + "; x2 = " + str(x[1].value) + "; x3 = " + str(x[2].value))
print("Optimal dual variables lambda1 = " + str(constraints[0].dual_value))
print("Optimal dual variables lambda2 = " + str(constraints[1].dual_value))
print("Optimal dual variables lambda3 = " + str(constraints[2].dual_value))
print("Optimal dual variables lambda4 = " + str(constraints[3].dual_value))
print("Optimal dual variables lambda5 = " + str(constraints[4].dual_value))
```

The solutions are the following.



```
Solution -0.9547712589294085
Status: optimal
Optimal value p* = -0.9547712589294085
Optimal var: x1 = 0.4226489442893967; x2 = 1.577351049782123; x3 = 0.5773510541368012
Optimal dual variables lambda1 = 1.7320483134403175
Optimal dual variables lambda2 = 0.6339745314617544
Optimal dual variables lambda3 = 6.437850296384749e-09
Optimal dual variables lambda4 = 6.544679319172325e-09
Optimal dual variables lambda5 = 1.7755538040590713e-09
```

The solutions agree with the ones given in the problems guide.

8. **Consider a Resource Allocation problem such as the one given at class, like slide 12 of Topic 13 but with a wireless network. Give the traffic allocated to each user xi and the percentage of time each link Rij is used. Also give the dual variables. The variables Rij represent a slot in which node i is scheduled to transmit to node j.**

$$\textbf{Maximize} \sum_{i}^{N} \log(x_i)$$

$$\textbf{Subject to} \quad \begin{aligned} &x_1 + x_2 \le R_{12} \\ &x_1 \le R_{23} \\ &x_3 \le R_{32} \\ &R_{12} + R_{23} + R_{32} \le 1 \end{aligned}$$

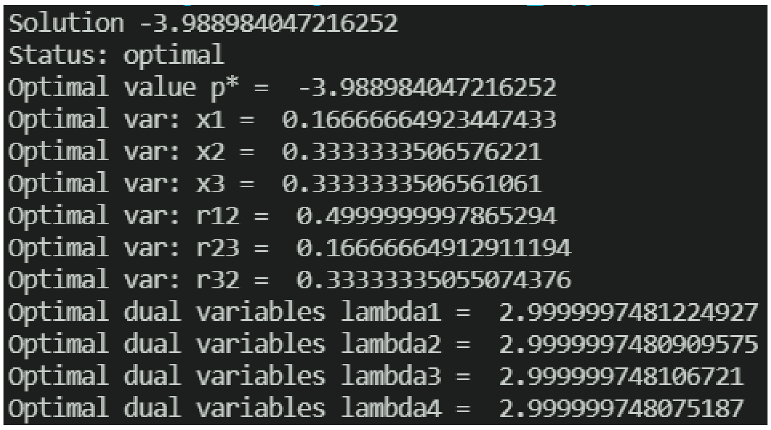
$$\textbf{Variables} \; x_i, R_{jk}$$

The next code implements the exercise. The structure is the same as all the problems implemented with the CVXPY library.

```python
from cvxpy import *


x = Variable(3, name='x') # Define the two variables
r = Variable(3, name='r')

f1 = x[0] + x[1] # Define all the constraints of the model
f2 = x[0]
f3 = x[2]
f4 = r[0] + r[1] + r[2]

constraints = [f1 <= r[0], f2 <= r[1], f3 <= r[2], f4 <= 1]

f0 = sum(log(x)) # Define the objective function
prob = Problem(Maximize(f0), constraints) # Create the model

print("solve", prob.solve())  # Returns the optimal value.
print("status:", prob.status)
print("optimal value p* = ", prob.value)
print("optimal var: x1 = ", x[0].value)
print("optimal var: x2 = ", x[1].value)
print("optimal var: x3 = ", x[2].value)
print("optimal var: r12 = ", r[0].value)
print("optimal var: r23 = ", r[1].value)
print("optimal var: r32 = ", r[2].value)
print("optimal dual variables lambda1 = ", constraints[0].dual_value)
print("optimal dual variables lambda2 = ", constraints[1].dual_value)
print("optimal dual variables lambda3 = ", constraints[2].dual_value)
print("optimal dual variables lambda4 = ", constraints[3].dual_value) #µ1
```

Imanol Rojas Pérez

The solutions of the problem are the following:



```
Solution -3.988984047216252
Status: optimal
Optimal value p* =  -3.988984047216252
Optimal var: x1 =  0.16666664923447433
Optimal var: x2 =  0.3333333506576221
Optimal var: x3 =  0.3333333506561061
Optimal var: r12 =  0.4999999997865294
Optimal var: r23 =  0.16666664912911194
Optimal var: r32 =  0.33333335055074376
Optimal dual variables lambda1 =  2.9999997481224927
Optimal dual variables lambda2 =  2.9999997480909575
Optimal dual variables lambda3 =  2.999999748106721
Optimal dual variables lambda4 =  2.999999748075187
```

These solutions agree with the ones given in the problems guide.