Report Project 1

Purpose

The purpose of this project is to do some exercise about optimization in order to understand how it works and how we must approach the problems.
For doing the exercises, we used Python and some libraries like scipy and CVXPY in order to compare the results and make conclusions.
For the scipy version, we solve the exercise in three ways:

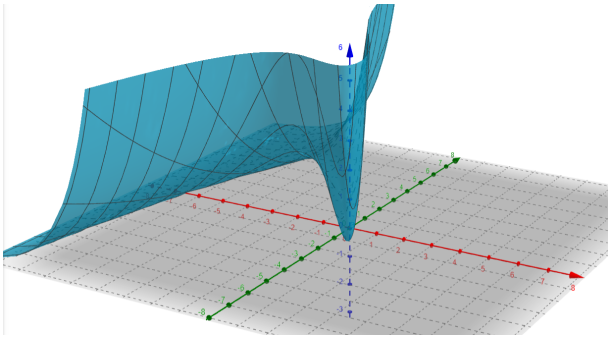
- using SLSQP;
- using SLSQP and the jacobian;

Exercise 1

$$\begin{aligned} \text{minimize} \quad & e^{x_1}(4 * x_1^2 + 2 * x_2^2 + 4 * x_1 * x_2 + 2 * x_2 + 1) \\ \text{subject to} \quad & x_1 * x_2 - x_1 - x_2 \leq -1.5 \\ & -x_1 * x_2 \leq 10 \\ \text{var} \quad & x_1, x_2 \end{aligned}$$

What can we say about the function?

- It is a product of an exponential function and a polynomial;
- the exponential function is a convex function;
- however the polynomial is geometric. So we can find a local minimum but we don't know if we can find the global one, it depends on the start point;
- So, we can conclude that the function we have to minimize is not a convex function.



We can prove it by seeing the graph, in fact the points where the function change concavity
are clearly visible.
Another proof is the hessian: even though it is a positive definite matrix (and so it is possible that the problem could be convex), but by seeing the determinant:

$$H_{00} = \frac{\partial^2 f(x)}{\partial x1 \partial x1} = e^{x1} * (4 * x1^2 + 2 * x2^2 + 4 * x1 * x2 + 2 * x1 + 8 * x1 + 4 * x2 + 9)$$

$$H_{01} = \frac{\partial^2 f(x)}{\partial x1 \partial x2} = e^{x1} * (4 * x2 + 4 * x0 + 6)$$

$$H_{10} = \frac{\partial^2 f(x)}{\partial x \partial x} = e^{x1} * (4 * x2 + 4 * x1 + 6)$$

$$H_{11} = \frac{\partial^2 f(x)}{\partial x \partial x} = e^{x1} * 4$$

$$det(H) = H_{00} * H_{11} - H_{10} * H_{01} =$$

$$= 4 * e^{2*x1} * (20 * x1 + 12 * x2 + 4 * x1 * x2 + 9) +$$

$$- e^{2*x1} * (64 * x1 + 48 * x2 + 16 * x2^2 + 32 * x1 * x2 + 36)$$

As we can see in the formula det(H), the second part grow faster than the first, but the minus makes this one negative! So the determinant of the hessian is less or equal to zero, we can

conclude that the function is not convex.

And about the constraint?

The two inequalities are affine but not convex, also there are no intersections neither with each other nor with the function we have to minimize.
So we can conclude that there are no feasible points.

The results

This exercise is solved using scipy and CVXPY. For the scipy version, we considered the points (0,0), (10,20), (-10,1) and (-30,-30), all of these are not feasible in this problem.
In the table below we can see the results.

Only SLSQP

Start	Optimal point	Optimal value	Iterations	Evaluations (function + gradient)	Status
(0,0)	(-9.55, 1)	0.0257	16	50+16	Completed
(10,20)	(-9.55, 1)	0.0257	35	139+34	Completed with overflow
(-10,1)	(-9.55, 1)	0.0257	3	10+3	Completed
(-30,-30)	(-9.55, 1)	0.0257	14	43+14	Completed

As we can see in the table above, the solver can find a solution even though the points are not feasible. The only point which presents an anomaly is (10,20) because it causes an overflow in the exponential function. Even though it belongs to the same optimal point with a major number of iterations and evaluations. In any case, the point (-10,1) reaches the optimal one with the least number of iterations and evaluations, so it is the best result!

SLSQP and Jacobian

Start	Optimal point	Optimal value	Iterations	Evaluations (function + gradient)	Status
(0,0)	(-9.55, 1)	0.0257	21	34+21	Completed
(10,20)	(10,20)	31740137	5	1+1	Positive directional derivative
(-10,1)	(-9.55, 1)	0.0257	3	4+3	Completed
(-30,-30)	(1.23, -1.24)	21.1	100	998+99	Limit reached

Solving the problem with SLSQP and the jacobian gives different results! In fact, the point (10,20) doesn’t belong to the optimal one because there is a positive direction in the

linesearch that doesn't allow the descent. Moreover, the point (-30,-30) doesn't complete the descent because it reaches the maximum number of iterations. Respect to (10,20), we have a solution with (-30,-30), it is not optimal but it is a result! This point could belong to the optimal point if we adjust the parameter.
The points (0,0) and (-10,1) belong to the optimal point, in particular the last one has an improvement in terms of performance, passing from 13 evaluations to 7!
Instead, the point (0,0) has more iterations and less evaluations, so we can say there is a balance in terms of performance.

Jacobian and Hessian

Start	Optimal point	Optimal value	Iterations	Evaluations (function + gradient)	Status
(0,0)	(-9.55, 1)	0.0257	21	34+21	Completed
(10,20)	(10,20)	31740137	5	1+1	Positive directional derivative
(-10,1)	(-9.55, 1)	0.0257	3	4+3	Completed
(-30,-30)	(1.23, -1.24)	21.1	100	998+99	Limit reached

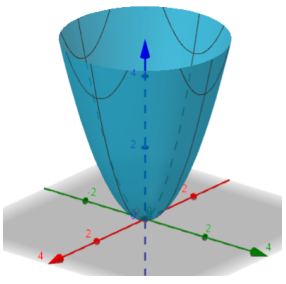
About this last experiment, We can make some conclusions because these data and the previous ones are the same. This happened because the minimize function of scipy use SLSQP in certain conditions, this method doesn't use the hessian to do the descent and so it uses only the jacobian.

Exercise 2

$$\begin{aligned} \text{minimize} \quad & x_1^2 + x_2^2 \\ \text{subject to} \quad & 0.5 \leq x_1 \\ & -x_1 - x_2 + 1 \leq 0 \\ & -x_1^2 - x_2^2 + 1 \leq 0 \\ & -9x_1^2 - x_2^2 + 9 \leq 0 \\ & -x_1^2 + x_2 \leq 0 \\ & -x_2^2 + x_1 \leq 0 \end{aligned}$$

The problem

The function we have to minimize in the second problem is a sum of square, it is a quadratic function that can represented as $x^t P x$ where P is a matrix with diagonal 1 and 0 in other parts.
We can also prove the convexity because the sum of convex functions is a convex function too, so the function is convex because it is the sum of squares, which are convex!
The convexity is also visible on the graph, so we can easily conclude that the function we have to minimize is convex.



About the constraints

The constraints of this problem are affine and some of these are also convex. Every constraint intersects the others, so at this time we have a feasible set.
For this problem we chose the following points:

- feasible points: (1,1), (4,3), (2,4);
- non-feasible points: (-1,-1), (0,0), (-3,2)

Results

Only SLSQP

Start	Optimal point	Optimal value	Iterations	Evaluations (function + gradient)	Status
(1,1)	(1,1)	2	1	3+1	Completed
(4,3)	(1,1)	2	11	39+10	Completed
(2,4)	(1,1)	2	8	21+7	Completed
(-1,-1)	(-0.9,-0.9)	1.8	8	12+4	Positive derivative linesearch
(0,0)	(-8.11e-15,3.25e-17)	6.58e-29	10	68+6	Positive derivative linesearch
(-3,2)	(-2.17,2.36)	2.572	12	34+12	Positive derivative linesearch

In the table above we can see the results given by each point. We can see that all the feasible points are completely successful, the gradient descent and everyone give the same optimal point as a result. In this case the point (1,1) became the best in terms of performances because it is the optimal point, in fact the other two feasible points belong to it. If we don't consider the point (1,1), the point (2,4) allows us to get the best performances because it is closer to the optimal point.

SLSQP and Jacobian

Start	Optimal point	Optimal value	Iterations	Evaluations (function + gradient)	Status
(1,1)	(1,1)	2	1	1+1	Completed
(4,3)	(1,1)	2	11	18+10	Completed
(2,4)	(1,1)	2	8	7+7	Completed
(-1,-1)	(-0.9,-0.9)	1.8	17	76+13	Positive derivative linesearch
(0,0)	(6.58e-16,	4.36e-31	18	144+14	Positive

	-4.76e-17)				derivative linesearch
(-3,2)	(1.00000001 ,1.00000001)	2	22	37+20	Positive derivative linesearch

Using SLSQP with the jacobian give the same results but with some differences:

- the feasible points give the same result but with an improvement in term of performances;
- The number of iterations and evaluations for the non-feasible points is largely grown;
- The point (-1,-1) give the same solution of the previous experiment, so there is a lack of performances;
- Even with more iterations and evaluations, the non-feasible point (-3,2) is practically equal to the optimal point!

Exercise 3

For exercise 3, we have to minimize the same function of the previous exercise, but with different constraints.
We know that the function is convex, but what about the constraints? The first one is convex and the second one is affine. The constraints intersect each other, so we have a feasible set.

$$\begin{aligned} \text{minimize} \quad & x_1^2 + x_2^2 \\ \text{subject to} \quad & x^2 + x_1 x_2 + x_2^2 \leq 3 \\ & 3x_1 + 2x_2 \geq 3 \\ \text{var} \quad & x_1, x_2 \end{aligned}$$

Results

We do this exercise both with scipy and with CVXPY. With scipy we use the points (1,1), (1,0) and (1,0.5) as feasible points and (7,-7), (5, -5) and (10,-10) as non-feasible ones.

SLSQP

Start	Optimal point	Optimal value	Iterations	Evaluations (function + gradient)	Status
(1,1)	(0.69, 0.46)	0.69	4	12+4	Completed
(1,0)	(0.69, 0.46)	0.69	2	7+2	Completed
(1,0.5)	(0.69, 0.46)	0.69	4	12+4	Completed
(5,-5)	(0.69, 0.46)	0.69	4	13+4	Completed
(7,-7)	(0.69, 0.46)	0.69	4	13+4	Completed
(10,-10)	(0.69, 0.46)	0.69	4	13+4	Completed

As we can see from the results in the table, all the points allow us to get the optimal one, independently of whether it is feasible or not. We can see there are two optimal points, it happens because of their similarity, in fact they are very close.

SLSQP and jacobian

Start	Optimal point	Optimal value	Iterations	Evaluations (function + gradient)	Status
(1,1)	(0.69, 0.46)	0.69	4	4+4	Completed
(1,0)	(0.69, 0.46)	0.69	2	3+2	Completed
(1,0.5)	(0.69, 0.46)	0.69	4	4+4	Completed
(5,-5)	(0.69, 0.46)	0.69	4	5+4	Completed
(7,-7)	(0.69, 0.46)	0.69	4	5+4	Completed
(10,-10)	(0.69, 0.46)	0.69	4	5+4	Completed

Using the Jacobian with SLSQP doesn't give some changes with respect to the previous experiments, in fact the only change we can see regards the performances.

CVXPY

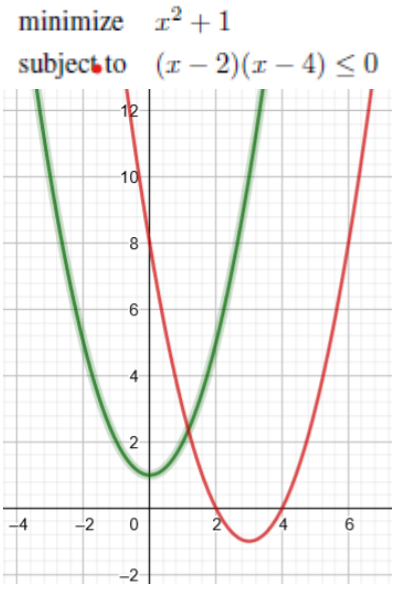
The experiment with CVXPY works in a different way: the start point is taken randomly, so we don't have to choose it. Respect to scipy, CVXPY gives us the solution to the dual problem, so we have the dual optimal point.

Optimal point	Optimal value	Lambdas	Iterations
(0.69, 0.46)	0.69	(1.39e-10, 0.462)	8

Comparing these results with the previouses, we can see that each other finds the optimal point, moreover the number of iterations is quite similar.

Exercise 4

The exercise 4 consists in the minimization of the function $x^2 + 1$ subject to one constraint. What can we say about the function? We can consider it as a quadratic function because the square of 1 is always 1, so the function is convex.
Another proof of convexity is in the graph, in fact it is clearly visible without difficulties.



The only constraint the problem has is a product of polynomials, is it convex or not? This is a particular case of parabola, we can get it with the square function, that is convex. So we can conclude that the constraint is convex and represent the feasible set of the problem.

Result

For this exercise we choose 2,3 and 4 as feasible points and 1,5,6,7 as non feasibles, which result do we get?

Only SLSQP

Start	Optimal point	Optimal value	Iterations	Evaluations (function + gradient)	Status
2	(2,2)	5	1	3+1	Completed
3	(2,3)	4.99	7	22+7	Completed
4	(2,4)	4.99	7	22+7	Completed
0	(2,2)	4.99	6	18+6	Completed
1	(2,2)	4.99	6	19+6	Completed
5	(2,3.96)	4.99	7	22+7	Completed
6	(2,3.98)	4.99	7	22+7	Completed
7	(2,3.99)	4.99	7	22+7	Completed

Using only SLSQP, all the points belong to the optimal one with more or less the same number of iterations and evaluations, independently of whether it is feasible or not. The feasible point 2 is the optimal point and, for this reason, it allows the best performances.

Jacobian with SLSQP

Start	Optimal point	Optimal value	Iterations	Evaluations (function + gradient)	Status
2	(2,2)	5	1	1+1	Completed
3	(2,3)	4.99	7	8+7	Completed
4	(2,4)	4.99	7	8+7	Completed
0	(2,2)	4.99	6	6+6	Completed
1	(2,2)	4.99	5	5+5	Completed
5	(2,3.96)	4.99	7	8+7	Completed
6	(2,3.98)	4.99	7	8+7	Completed

7	(2,3.99)	4.99	7	8+7	Completed

This experiment give essentially the same results but with a minor number of iterations and evaluations.

CVXPY

Optimal point	Optimal value	Lambda	iterations
1.99	4.99	2	9

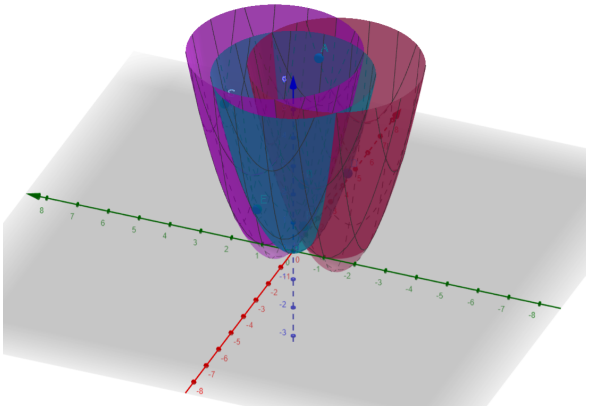
Also for this exercise, we can see that the results are practically the same.

Exercise 5

In the fifth exercise we have to minimize the same function of the exercises 2 and 3. From these exercises we know that the function is convex, so we concentrate ourselves in the analysis of the constraints:

$$\begin{aligned} \text{minimize} \quad & x_1^2 + x_2^2 \\ \text{subject to} \quad & (x_1 - 1)^2 + (x_2 - 1)^2 \leq 1 \\ & (x_1 - 1)^2 + (x_2 + 1)^2 \leq 1 \end{aligned}$$

- The constraints are the sum of quadratic forms, so they are convex because the sum conserves this property. The convexity is also visible in the graph (red and purple functions);
- In the graph we can see that there are no negative points where there are the constraint functions, so this problem doesn't have a feasible space.



Results

SLSQP

Start	Optimal value	Optimal point	Iterations	Evaluations (function + gradient)	Status
(2.33, 0.11)	1	(9.99e-01,-243e-04)	82	732+78	Positive directional linesearch
(1.25, 2.18)	1	(0.99,0.009)	28	189+24	Positive directional linesearch
(-0.12, 2.18)	0.9998	(9.999e-01, -6.765e-04)	58	506+54	Positive directional linesearch

(0.68, -1.49)	1	(1, 2.66e-04)	100	1024+96	Positive directional linesearch
(-1.28, 0.66)	0.999	(9.999e-01,-2.931e-06)	23	169+19	Positive directional linesearch
(0.69,-0.06)	1.01	(1.001,-0.053)	38	316+34	Positive directional linesearch

As we can see in the table, every point doesn't complete the gradient descent, this happens because the points are not feasible. In any case, the points converge in (1,0), probably it is not the optimal point but it is the best we can get in these conditions! Regarding the performances, each point does a great number of iterations and evaluations, is there a better way?

SLSQP with Jacobian

Start	Optimal value	Optimal point	Iterations	Evaluations (function + gradient)	Status
(2.33, 0.11)	0.9997	(9.999e-01, 3.692e-04)	66	484+62	Positive directional linesearch
(1.25, 2.18)	0.9997	(0.9998,0.008)	64	561+60	Positive directional linesearch
(-0.12, 2.18)	0.99995	(9.9997e-01, -3571e-04)	83	669+79	Positive directional linesearch
(0.68, -1.49)	0.99993	(9.9996e-01, 3.155e-05)	38	285+34	Positive directional linesearch
(-1.28, 0.66)	0.999	(9.9998e-01, 2.7043e-04)	79	554+75	Positive directional linesearch
(0.69,-0.06)	1.01	(9.998e-01, 3.692e-04)	67	460+63	Positive directional linesearch

Using SLSQP with the jacobian, we get the same results but with some differences about the performance. In fact for some points there was a great improvement, an example is (0.68, -1.49) that passed from 1024 evaluations to 285. Instead other points had a lack of performances like (1.25, 2.18). In general the use of the jacobian generates a lack of performances, so for these points it isn't convenient.

CVXPY

Optimal point	Optimal value	Lambda	iterations
(9.9998e-01,1.991e-14)	0.99	(28013.52,28013.52)	24

The exercise with CVXPY gives the same results, the principal difference regards the performances which are in media better than scipy.

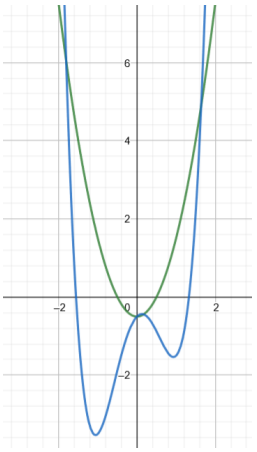
Exercise 6

The sixth exercise consists in the implementation of a gradient descent in two ways:

- the classic gradient descent;
- Newton's method.

First, what is a gradient descent method? How does it work?
A gradient descent is a method allowing us to optimize a function calculating its maximum or minimum, it works iteratively calculation the point at the next step in this way:

$$x^{(k+1)} = x^{(k)} + t * d^{(k)}$$



In the formula, $x^{(k)}$ is the point at the k-th step, t is the direction and $d^{(k)}$ is the descent direction, this last variable is always equal to the negative gradient of the function.
Before starting to analyze the code, we tank a moment about the functions we have to optimize:

- $f(x) = 2 * x^2 - 0.5$ is the green function in the graph, it is clearly visible that this function is convex. We have to test the point 3 of this function;
- $g(x) = 2 * x^4 - 4 * x^2 - x - 0.5$ is a geometric function because it is a polynomial, so it is not convex. We can confirm its convexity by seeing the blu function in the graph. For this function we have to test the points -2, -0.5, 0.5 and 2.

The code

Now we analyze the code. We implemented a class doing the gradient descent effectively, also we did some support classes to simplify the work and avoid the repetitions.
The GradientDescent and NewtonDescent classes implement these methods:

- a constructor the take a function and a ε value as input, the last one will be useful in the stop conditions;
- the method getT() implements the backtracking line search, so, it takes a point x and the descend direction d as input and computes a value t;
- the method getDirection() returns the negative gradient in base of a point x;
- The method descend() takes a point x0 as input and does the gradient descent starting from this point.

Backtracking line search

The backtracking line search is a method to calculate the best step size t in base of the point x, the descent direction d and two hyperparameters α and β.
How does it work precisely? The value t start from 1 and the algorithm continues to scale t by β until $f(x + t * d) - f(x) * \alpha * t * \bigtriangledown f(x)^t d$ is less or equal to zero.
For this exercise we chose the α=0.155 and β=0.45, in particular the last one allows us to get interesting results with respect to 0.1 or 0.8 as recommended by many people.

```
def getT(self,x,d): #Backtracking line search
    a=(31/2)*0.01
    b=0.45
    count=0
    def cond(x0,t0,d0):
        c=self.fun.applyGrad(x0)[0]
        left=self.fun.applyFun(x0+t0*d)
        right=self.fun.applyFun(x0)+a*t0*c*d0
        return left-right>0
    t=1
    while cond(x,t,d):
        t=b*t
        count+=1
    return t,count
```

Classic gradient descent

The classic gradient descent uses the negative gradient as the descent direction.
How does it work? Starting from a point x0, it continues to compute a new point x using the formula we explained at the beginning of this section.
When does the algorithm stop? It stops when the l2-norm of the function is less or equal to ε.

```
def getDirection(self,x):
    return -self.fun.applyGrad(x)[0]

def descend(self,x0): #calc the optimal point beginning from x0
    print(x0)
    x=x0 #step 1
    count=0
    while self.fun.twoNorm(x)>self.eps: #step 5
        d=self.getDirection(x)#step 2
        t,ct=self.getT(x,d) #step 3
        x=x+t*d #step 4
        count+=(ct+1)
        #print("descend")
    return GradResult(x0,x,self.fun.applyFun(x),count)
```

Results

Considering the image of the right and the graph at the beginning of this section, we can conclude that:

- the point 3 reach the optimal point, that is 0, this happen because the function is convex and the are no constraints, so the feasible is is the domain of the function;
- In the second function, the negative points reach the global minimum that is the point -1.06 and the positive one, that is 0.93.
- However there could be cases where some positive points reach the global minimum, for example point 2. This happens because the computation of the new point is influenced by α and β, so different values give different results!

```
Results:
    Start: 3
    Opt. point: 5.095068912299994e-06
    Opt. value: -0.49999999994808053
    Num of it.: 24

    Start: -2
    Opt. point: -1.0574579017490782
    Opt. value: -3.529507282390682
    Num of it.: 41

    Start: -0.5
    Opt. point: -1.0574550422563354
    Opt. value: -3.529507282536184
    Num of it.: 42

    Start: 0.5
    Opt. point: 0.9304050741229585
    Opt. value: -1.5334970165410375
    Num of it.: 23

    Start: 2
    Opt. point: -1.057456881027739
    Opt. value: -3.529507282460297
    Num of it.: 36
```

Newton gradient descent

Newton's method is an optimized descent method that allows us to get the optimal value in general by doing less steps than the gradient descent.
Which are the differences with respect to the gradient descent?
There are principally two differences:

- the descent direction is not the negative gradient but it is computed as the division of the square gradient and the gradient of the function;
- The stop condition is different, now we have to calculate a value λ as we can the in the function getDec(), it is the newton decrement.

```
def getDec(self,x):
    a=self.fun.applyGrad(x)[0]
    b=self.fun.applySqGrad(x)[0]
    return (a*a)/b

def getDirection(self,x):
    a=self.fun.applySqGrad(x)[0]
    b=self.fun.applyGrad(x)[0]
    return -b/a

def descend(self,x0):
    x=x0
    count=0
    while self.getDec(x)/2>self.eps:
        d=self.getDirection(x)
        t,ct=self.getT(x,d)
        x=x+t*d
        count+=(ct+1)
        print("descend")
    return GradResult(x0,x,self.fun.applyFun(x),count)
```

Results

Analyzing the results on the right, we can see clearly the efficiency of the newton's method respecto to the classic gradient descent. However, this efficiency has a cost: some points such as 0.5 and -0.5 could not belong to the optimal point.
In any case we can say:

- point 3 has improved in terms of performance, passing from 24 to 1 iteration. We can say the same thing for the point -2!
- point 2 now belongs to the global minimum and does it with a minor number of iterations.

```
Results:
    Start: 3
    Opt. point: 0.0
    Opt. value: -0.5
    Num of it.: 1

    Start: -2
    Opt. point: -1.0577905282955613
    Opt. value: -3.529506214116978
    Num of it.: 4

    Start: -0.5
    Opt. point: -0.5
    Opt. value: -1.875
    Num of it.: 0

    Start: 0.5
    Opt. point: 0.5
    Opt. value: -0.875
    Num of it.: 0

    Start: 2
    Opt. point: 0.9331433745699511
    Opt. value: -1.5334488905735097
    Num of it.: 4
```

Conclusions

How convenient is Newton's method with respect to the classic gradient descent? It could be convenient when the number of iterations in the gradient descent is very high and we can accept a similar value instead of the optimal one. In fact the classic gradient descent gives the optimal results most of the times, so our purpose is the optimization. Instead with Newton's method our purpose is the performances: so we can accept a closer point to the optimal one saving resources.

Exercise 7

Exercise 7 consists to solving of a network utility problem using CVXPY. What can we say about this problem? Given a network with L, each one with capacity Cl, and some routes

which are subsets of L, we have to maximize the utility of the network. How can we do this? We define a function Ur(xr), which is the utility to transmit a route r at rate xr from a source:

$Ur(x_r) = log(x_r)$

As we can see in the formula, Ur(xr) is a logarithm function that is not convex.

This is the utility of a route, is there a way to get the total utility of the network? Yes, you have to define a function W(x) which is the sum of each utility:

$$W(x) = \sum_r U(x_r) = \sum_r log(x_r)$$

So, W(x) is the function we have to maximize. What about the constraints? We get the constraints from the routes, in our example:

- the route S0 uses the links 1 and 2;
- the route S1 uses the link 2;
- the route S2 use the links 1 and 5;

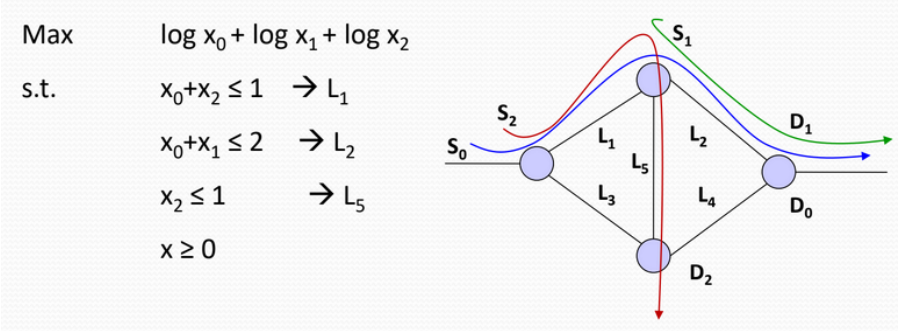
So we sum the rates of each route, this value have to been less or equal to the capacity of the link:

- $x0 + x2 <= C1$
- $x0 + x1 <= C2$
- $x2 <= C5$

Moreover, every rate xi have to be more or equal to 0:

$x >= 0$

We can summary the problem as we can see in the image below:



Results

Using the network configuration in the previous image, we can see that there are more dual optimal values than constraints, this happens because we didn't consider the constraint in the links 3 and 4, which no routes use.

These constraints are the followings:

- $0 <= C3$
- $0 <= C4$

We can clearly see that these constraints are always true, so we can remove them from our problem.

```
xstar: [0.42264894 1.57735105 0.57735105]
pstar: 0.9547712589294085
lambdas: [1.7320483134403175, 0.6339745314617544, 6.437850296384749e-09,
6.544679319172325e-09, 1.7755538040590713e-09, 4.7795891196703965e-09]
numIt: 15
```

Exercise 8

The last exercise is similar to the previous one, in this case we need to implement a resource allocation problem, always using CVXPY.
What can we say about the problem? It is practically the same problem we explained previously, but here we have a wireless network and so we have to consider the memory allocation.
How can we represent the memory allocation? We can represent as a variable Rij, it indicates a slot of memory used by the node i to send data to the node j.

$$\begin{aligned} \text{maximize} \quad & \sum_{i}^{N} log(x_i) \\ \text{subject to} \quad & x_1 + x_2 \leq R_{12} \\ & x_1 \leq R_{23} \\ & x_3 \leq R_{32} \\ & R_{12} + R_{23} + R_{32} \leq 1 \\ \text{subject to} \quad & x_i, R_{jk} \end{aligned}$$

Considering this variable, we have to rewrite the constraints in base of the routes:

- $x1 + x2 <= R12$
- $x1 <= R23$
- $x3 <= R32$

Also we have to consider that the resources are limited, so the sum of memory allocated doesn't have to overcome the total amount of memory:
$R12 + R23 + R32 <= 1$

Results

In the results below, we use only one variable because our support classes we created use only one variable. It's better to do an interpretation:

- the first three point in x* are respectively x1, x2 and x3;
- the other points in x* are R12, R23, R32.

```
xstar: [0.16666664923447433, 0.3333333506576221, 0.3333333506561061, 0.4999999997865294, 0.16666664912911194,
 0.33333335055074376]
pstar: 3.988984047216252
lambdas: [2.9999997481224927, 2.9999997480909575, 2.999999748106721, 2.999999748075187]
numIt: 15
```

Note

All the code is in this repository: https://github.com/MarcoCarry97/Topic-On-Optimization-And-Machine-Learning