Topics On Optimization and Machine Learning
Homework 1

Adrià Lisa

Question 1: Non-linear optimization

Consider the following optimization problem.

$$\begin{aligned} \text{minimize} \quad & f_0(x_1, x_2) = e^{x_1}(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1) \\ \text{subject to} \quad & f_1(x_1, x_2) = x_1x_2 - x_1 - x_2 + 1.5 \leq 0 \\ & f_2(x_1, x_2) = -x_1x_2 - 10 \leq 0 \\ \text{var} \quad & x_1, x_2 \end{aligned}$$

Part (a): Identify whether it is convex or not.

The optimization problem will be convex if the objective function (named f_0) and the constraint functions in standard form (named f_1, f_2) are convex. To analyze f_0 we shall look at it with the following form.

$$f_0(x_1, x_2) = e^{x_1} \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 4 & 2 \\ 2 & 2 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + e^{x_1}(2x_2 + 1)$$

Note that the 2×2 matrix is positive definite, thus the quadratic form $4x_1^2 + 2x_2^2 + 4x_1x_2$ is strictly convex. Evidently, $2x_2 + 1$ is also convex, so we have that f_0 is convex w.r.t. x_2 because the sum of convex functions is convex. We now prove the convexity w.r.t. x_1 simply by differentiating:

$$\frac{\partial^2}{\partial x_1^2} f_0(x_1, x_2) = \left(4x_1^2 + (4x_2 + 16)x_1 + 2x_2^2 + 10x_2 + 9\right) e^{x_1} = p_{x_2}(x_1)e^{x_1} \tag{1}$$

Since $\lim_{x_1 \to \infty} p_{x_2}(x_1) = +\infty$ and the minimum value of the polynomial is located at the value x_1^* such that $8x_1^* + 4x_2 + 16 = 0$, where

$$\begin{aligned} p_{x_2}(x_1^*) &= (x_2^2 + 16x_2 + 64 - 2x_2^2 - 16x_2 - 8x_2 - 64 + 2x_2^2 + 10x_2 + 9) \\ &= (x_2^2 + 2x_2 + 9) > 0 \qquad \forall x_2 \in \mathbb{R} \end{aligned}$$

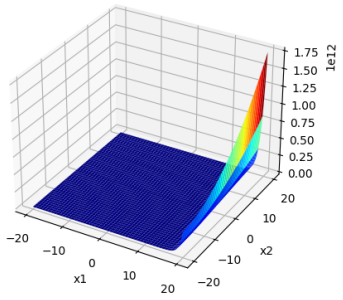
Therefore, we conclude that $\partial_{x_1}^2 f_0 > 0$ which implies that f_0 is convex in both variables.
However, the hessian matrices of the constraint functions f_1 and f_2 are, respectively,

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}, \begin{bmatrix} 0 & -1 \\ -1 & 0 \end{bmatrix} \not\succeq 0$$

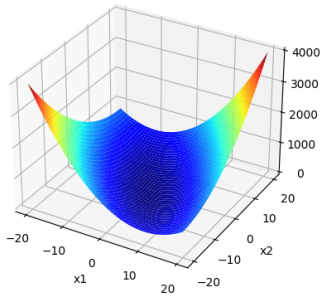
non-positive definite matrices, implying that f_1 and f_2 are non-convex.
In conclusion, the constraints of this optimization problem are not convex, but the objective function is.

Part (b): Examinining the objective function

If we attempt to plot f_0, we can quickly see that for large values of x_1 the exponential factor becomes very dominant. With limited resolution, the plot does not provide any information as it essentially becomes $f_0(x_1, x_2) \approx e^{x_1}$. Most of the information of this function is found at its polynomial part, $e^{-x_1} f_0(x_1, x_2)$. In fig. 1, we can see a plot of both functions.



(a) $f_0(x_1, x_2)$



(b) $e^{-x_1} f_0(x_1, x_2)$

Figure 1: Graphs of the objective function f_0 (a) and its polynomial part (b). It is clear that the exponential factor dominates the output of the objective function, and the polynomial part has a unique global minimum, located at $(0.5, -1)$.

If we use the proposed `scipy` module to solve the unconstrained problem, we can obtain the solution

$$f_0(-123324180, 123274249) = 0.0$$

However, it is clear that $f_0(x^*) = 0 \iff x^*$ is a root of the polynomial part, and the only zero of

$$4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1$$

is $(0.5, -1)$.
This error is due to a numerical issue, as $e^{-123324180}$ gets truncated to zero. Therefore, with the default numerical precision, `simpy` will think that all the points in $\{(x_1, x_2) \in \mathbb{R}^2 | x_1 \leq -123324180\}$ are optimal points!

Part (c): Solve the constrained problem using `scipy.optimize.minimize`.

The performance of the method using the the proposed initial points is summarized in table 1.

x_0 value	[0,0]	[10,20]	[-10,1]	[-30,-30]
f_0 value	0.023	4.495e7	0.023	0.098
nº iterations	17	1	3	8
nº evaluations of f_0	54	3	10	40
nº evaluations of ∇f_0	17	1	3	8

Table 1: Performance of `sympy.optimize.minimize` without providing the gradient. The second row is the value for the objective function attained after the algorithmic search.

The execution with $x_0 = [10, 20]$ failed, with the message `Inequality constraints incompatible`. This is interesting since none of the initial values satisfy the constraints, but only the second produced a failure. I suppose that the algorithm decided that it would not be able to find a feasible point after an initial exploration. The returned value for the objective function is $f_0(10, 20) = 4.495 * 10^7$.
The best solution, $x = (-9.54740503, 1.04740503)$, was attained from the first and the third initial values. We can not guarantee their optimality without considering the Lagrangian, but we can obtain some insights if we observe the feasible region in fig. 2. We can see that x has the lowest value for the first coordinate amongst all the feasible points. As we discussed before, e^{x_1} is the dominant factor of $f_0(x_1, x_2)$, so there is an argument in favor of x being the optimum. Moreover, if we run the solver from a point inside the other connex component, say $x_0 = [0, 2]$, the solution obtained is a point closest to x^*, with an objective function value of $3.06 > 0.23$.
Since approaching x^*, which is equivalent to minimize the polynomial part of f_0, does not cut it, the best we can do is to minimize e^{x_1}.

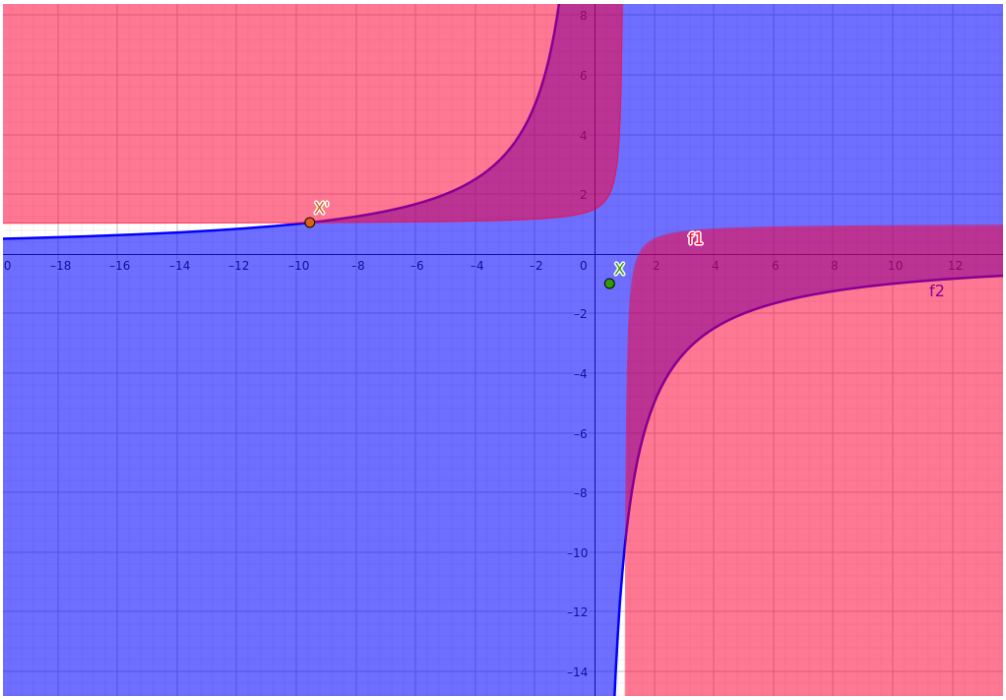


Figure 2: In red, there is the region $f_1 \leq 0$, and in blue, the region $f_2 \leq 0$. The point X represents the solution of the unconstrained problem, and X' the solution of the constrained problem.

Part (d): Try again after providing the Jacobian to the method.

The exact formula for the Jacobian for the objective function is:

$$\nabla f_0(x_1, x_2) = e^{x_1} \begin{bmatrix} 4x_1^2 + 4x_1x_2 + 8x_1 + 2x_2^2 + 6x_2 + 1 \\ 4x_1 + 4x_2 + 2 \end{bmatrix}$$

After running the optimization method again with the exact Jacobian, we save computational resources (thus time) by not having to approximate it. This can be seen in the generalized reduction of nº evaluations of f_0 in table 2 with respect to table 1.

x_0 value	[0,0]	[10,20]	[-10,1]	[-30,-30]
f_0 value	0.023	4.495e7	0.023	50.65
nº iterations	17	5	3	11
nº evaluations of f_0	20	1	4	57
nº evaluations of ∇f_0	17	1	3	11

Table 2: Performance of `sympy.optimize.minimize` after providing the gradient. In the second row we show the difference with the results of table 1.

We only obtained a different solution at the last initial point, $x_0 = [-30, -30]$. Previously, it was a point in the left connex component, $x = [-7.647, 1.058]$, but now we get a point in the right connex component, $x' = [0.657, 2.461]$. If we use python to evaluate the jacobian at this inital point, we get:

$$\nabla f_0(-30, -30) \approx \begin{bmatrix} +8.02977626956175 \cdot 10^{-10} \\ -2.22711426658396 \cdot 10^{-11} \end{bmatrix}$$

Whereas the precise value is

$$\nabla f_0(-30, -30) = e^{-30} \begin{bmatrix} -360 \\ -238 \end{bmatrix}.$$

So we can see that the direction of the Jacobian gets messed up, due to the values being so close to zero.

Question 2

Consider the following optimization problem

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

Part (a): Identify whether it is convex or not.

The objective function is convex, as it is the sum of two convex functions. The first and the second constraints are affine functions, thus convex and concave. The hessians of the rest of the constraints are:

$$\begin{bmatrix} -2 & 0 \\ 0 & -2 \end{bmatrix}, \begin{bmatrix} -18 & 0 \\ 0 & -2 \end{bmatrix}, \begin{bmatrix} -2 & 0 \\ 0 & 0 \end{bmatrix}, \begin{bmatrix} 0 & 0 \\ 0 & -2 \end{bmatrix}$$

All their eigenvalues are zero or negative, therefore, all the constraints are concave. This implies that this is a concave optimization problem, in non-standard form.

Part (b): Try SLSQP method with a feasible and an unfesible intial point.

I chose $x_i = [0, 0]$ as the infeasible initial point, and $x_f = [2, 2]$ as the feasible point. For x_i, the method failed to find any feasible solution, and the output was the following:

```
message : Positive directional derivative for linesearch
success : False
 status : 8
    fun : 0.31254965127528844
      x : [ 5.000e-01 2.500e-01]
    nit : 13
    jac : [ 1.000e+00 5.001e-01]
   nfev : 52
   njev : 9
```

Since the algorithm was able to perform 13 iterations, the issue must come from it not being able to converge on a feasible solution.
There is no documentation on `scypy` on this error flag. The only accessible resource I could find on SLSQP was the Wikipedia [link], from where I could learn that the iterative direction d for this method is computed as the solution of the following sub-problem:

$$\min_d \quad f(x_k) + \nabla f(x_k)^T d + \tfrac{1}{2} d^T \nabla^2_{xx} \mathcal{L}(x_k, \lambda_k) d \tag{2}$$

$$\text{s.t.} \quad h(x_k) + \nabla h(x_k)^T d \geq 0 \tag{3}$$

$$\tag{4}$$

Where f is the objective function, h is the vector of constraint functions, and $\mathcal{L}$ is the Lagrangian. Since in this problem, the Lagrangian hessian

$$\nabla^2_{xx} \mathcal{L}(x, \lambda) = \begin{bmatrix} 2\lambda_3 + 18\lambda_4 + 2\lambda_5 + 2 & 0 \\ 0 & 2\lambda_3 + 2\lambda_4 + 2\lambda_6 + 2 \end{bmatrix} \succ 0$$

is by no means ill conditioned, I would guess that the source of the issue comes from the concave feasibility function h. In contrast, for x_f `SLSQP` was successful in finding the optimal solution $x^* = [1, 1]$ in 6 iterations.

Part (c): Study the convergence after inputting the Jacobian.

The Jacobian of the objective function is

$$\nabla f_0(x) = \begin{bmatrix} 2x_1 \\ 2x_2 \end{bmatrix}.$$

After providing it to the solver, the results obtained where the same on x_f, with the same nº of iterations, and some savings in the nº evaluations of f_0. As expected.
In contrast, the number of iterations went down on x_i from 13 to 5, and it reached a different (infeasible) solution. The output for x_i was the following:

```
message : Positive directional derivative for linesearch
success : False
 status : 8
    fun : 0.25
      x : [ 5.000e-01 0.000e+00]
    nit : 5
    jac : [ 1.000e+00 0.00e+00]
   nfev : 1
   njev : 1
```

I am not able to explain why the nº evaluations of f_0 is only 1, while 5 iterations were made. At this point, some deep down inspection/debugging of the source code would be necessary. It is writen in Fortran, and can be found at this link.

Question 3

Consider the following optimization problem.

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

Part (a): Identify whether is convex or not

The objective function is convex, as it is the sum of two convex functions. The two constraints in standard form are:

$$\begin{aligned} h_1(x_1, x_2) &= x_1^2 + x_1 x_2 + x_2^2 - 3 \leq 0 \\ h_2(x_1, x_2) &= -3x_1 - 2x_2 + 3 \leq 0 \end{aligned}$$

We can easily compute their hessian matrices:

$$\nabla^2 h_1 = \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} \succ 0, \quad \nabla^2 h_2 = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix} \succeq 0$$

Both are semi-positive definite so it is a convex optimization problem.

Part (b): Solve it with `scipy` and check the convergence.

The SLSQP algorithm is able to converge independently of the initial point, as the problem has the property of being convex. Moreover, it will always converge towards the unique optimal solution, namely $x^* = [0.692307690.46153846]$ (see fig. 3).
I will not delve into a mathematical analysis of the convergence rates of SLSQP, but experimentation with several different initial points suggest it is quite fast.

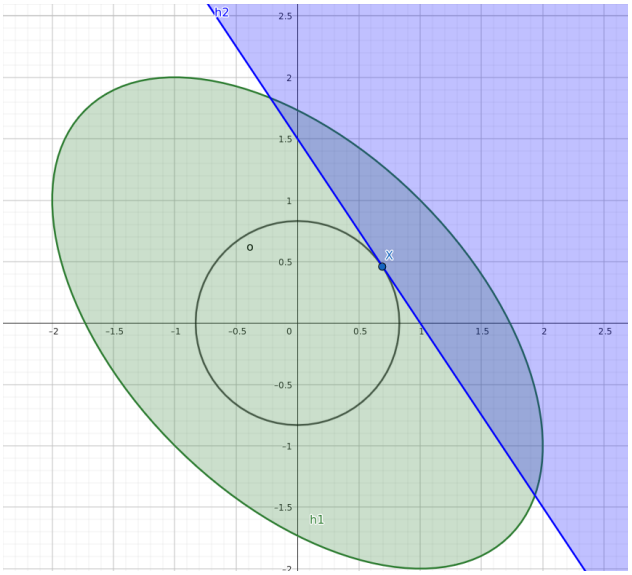


Figure 3: The feasibility region of the third exercise is the intersection between the green ellipse ($h_1(x) \leq 0$) and the blue semi-plane ($h_2(x) \leq 0$). The point X is the optimal solution, with objective function value equal the square of the radius of the gray circle intersecting with it.

Part (c): Program it with the CVX toolbox, and obtain the Lagrange multipliers.

An interesting issue with `CVXPY` occurred when I was coding the first constraint. If one encodes the first constraint as

```
x[0]**2 + x[0]*x[1] + x[1]**2 <= 3
```

they would get the following error:

```
DCPError:  Problem does not follow DCP rules.
```

This must happen because `CVXPY` reads every summand sepparately, and the term x_1x_2 is not a convex function by itself. But as was proved before, the whole function is a convex quadratic form. A workaround for this is to encode h_1 using `CVXPY`'s method for describing quadratic forms. The following is a code snippet on how I was able to correctly encode the third problem.

```
import cvxpy as cp
x = cp.Variable(2)
Q = [[1, .5], [.5, 1]]
constraints = [cp.quad_form(x, Q) <= 3, 3*x[0] + 2*x[1] >= 3]
prob = cp.Problem(cp.Minimize(x[0]**2 + x[1]**2), constraints=constraints)
prob.solve()
```

The optimal solution was reached, with Lagrange multipliers $\lambda_1 \simeq 0$ and $\lambda_2 = 0.461550969108317$ respectively for the first and the second constraints.
The Lagrangian, and the gradient of the Lagrangian for this problem are:

$$\mathcal{L}(x, \lambda) = x_1^2 + x_2^2 + \lambda_1 \left(x_1^2 + x_1x_2 + x_2^2 - 3\right) + \lambda_2 \left(-3x_1 - 2x_2 + 3\right)$$

$$\nabla_x \mathcal{L}(x, \lambda) = \begin{bmatrix} 2x_1 + \lambda_1 \cdot (2x_1 + x_2) - 3\lambda_2 \\ 2x_2 + \lambda_1 \left(x_1 + 2x_2\right) - 2\lambda_2 \end{bmatrix}$$

If we substitute with $\lambda_1 = 0$ and $\lambda_2 = 0.461550969108317$, we obtain that

$$\nabla_x \mathcal{L}(x, (0, 0.461550969108317) = 0 \iff \begin{array}{c} x_1 = \frac{3}{2} \cdot 0.461550969108317 \\ x_2 = 0.461550969108317 \end{array}$$

which is precisely, the optimal solution.

Question 4

Consider the following optimization problem.

$$\begin{aligned} \text{minimize} \quad & x^2 + 1 \\ \text{subject to} \quad & (x-2)(x-4) \leq 0 \end{aligned}$$

Part (a): Solve it by hand (primal and dual).

The Lagrangian of this problem is

$$\mathcal{L}(x, \lambda) = x^2 (\lambda + 1) - 6x\lambda + 8\lambda + 1.$$

A point x^* minimizing the Lagrangian would satisfy:

$$\frac{\partial}{\partial x}\mathcal{L}(x^*, \lambda) = -6\lambda + 2x^* (\lambda + 1) = 0 \iff x^* = \frac{3\lambda}{\lambda + 1}$$

which is well defined as the lagrangian multiplier satisfies $\lambda \geq 0$.
Therefore, we take $g(\lambda) := \mathcal{L}(x^*, \lambda)$, and the dual problem is defined as

$$\begin{aligned} \text{maximize} \quad & g(\lambda) = -\frac{9\lambda^2}{\lambda + 1} + 8\lambda + 1 \\ \text{subject to} \quad & \lambda \geq 0 \end{aligned}$$

The function $g(\lambda)$ has two critical points, -4 and 2. Thus, the solution for the dual is $\lambda_{opt} = 2$, and the solution for the primal is

$$x_{opt} = \frac{3 \cdot 2}{2 + 1} = 2.$$

Part (b): Solve it with the CVX toolbox, and give the Lagrange multipliers.

I encoded the problem in *CVXPY* using the following code snippet.

```
import cvxpy as cp

x = cp.Variable()
constraints = [x**2 - 6*x + 8 <= 0]
obj = cp.Minimize(x**2 + 1)

prob = cp.Problem(obj, constraints)
prob.solve()
```

The solutions for the primal and the dual that were obtained are:

$$x = 1.9999999968611173 \qquad \lambda = 2.0000322081789013$$

Which is consistent with the optimal solutions obtained in part (a).

Question 5

Consider the following optimization problem.

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

Part (a): Program it with the CVX toolbox.

This problem has a unique feasible point, $x^* = [1, 0]$, and as such, it is the optimal solution.
This pathological case makes `CVXPY` yield the result of 'infeasible problem'. Four different solvers, `SCS`, `ECOS`, `CVXOPT` and `CLARABEL`, were tried on the `solve()` method, all yielding the same result. One intuition for this is that any exploratory or stochastic method has a probability of hitting a single point virtually equal to zero.
Moreover, one can can even set a 'warm start' by letting the initial value as $x_0 = [1, 0]$, but the problem will still be considered infeasible by the solver.

Question 6: Gradient descent algorithm

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

- $f(x) = 2x^2 - 0.5$, with initial point $x^{(0)} = 3$ and accuracy (stop criterion) $\eta = 10^{-4}$. Give the final result, the final accuracy, and the number of steps.
- $f(x) = 2x^4 - 4x^2 + x - 0.5$, try several initial points $x^{(0)} = -2$, $x^{(0)} = -0.5$, $x^{(0)} = 0.5$, and $x^{(0)} = 2$. The accuracy (stop criterion) again is $\eta = 10^{-4}$. Give the final result, the final accuracy, and the number of steps.

Part (a): My program of the Gradient descent algorithm (GDA)

I implemented a python routine for the GDA using backtracking line search with the Armijo-Wolfe conditions. It requires some initial condition x_0, a callable function f, and its gradient or derivative df.

```
def GradDescent (x_0, f, df, stop_criterion=1e-4, alpha=0.1, beta=0.1):
    x = x_0
    steps = 0
    while(abs(df(x)) > stop_criterion):
        stepsize = 1
        while(f(x - stepsize * df(x))> f(x) - alpha*stepsize*df(x)*df(x)):
            stepsize = stepsize * beta
        x = x - stepsize * df(x)
        steps = steps + 1
    return x, steps
```

Part (b): First example results

The optimal solution for the first example is $x^* = 0$. The following figures compare the accuracy and the number of steps after running the GDA for different values of the Wolfe parameters, namely, α and β. The mean results for the number of steps and error across all the values for β, given the four considered values for α are summarized in table 3.
Looking at fig. 4, one can see that for very small values of β the algorithm behaves poorly. There are certain values of β that "hit the jackpot", allowing GDA to reach the solution within 1 or 2 steps. This behaviour is entirely due to chance, and will not replicate for any other problem or initial condition.
We see the same behaviour in fig. 5, where by chance the error can become smaller than 10^{-9}. The accuracy of the method remains good for all the parametrizations because we impose that $|\nabla f| < 10^{-4}$. However, table 3 reveals that there is some trade-off between speed and precision in the choice of α.

α	0.1	0.2	0.3	0.4
mean n° steps	26.16	19.27	16.36	15.46
mean error	$4.91 \cdot 10^{-6}$	$5.07 \cdot 10^{-6}$	$5.43 \cdot 10^{-6}$	$8.45 \cdot 10^{-6}$

Table 3: Mean number of steps and error for the GDA with gradient tolerance 10^{-4}. The means are computed over a range of 10000 values of $\beta \in (0, 1)$

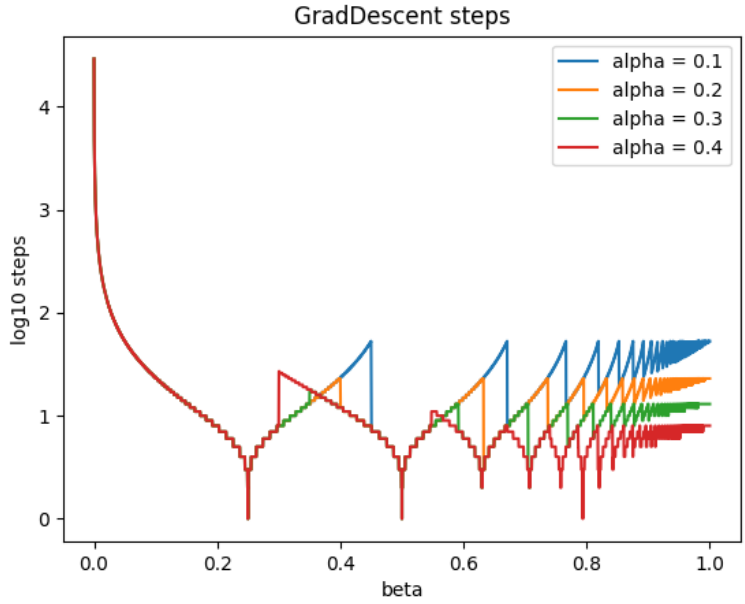


Figure 4: Comparison on the number of steps of GDA for $\alpha = 0.1, 0.2, 0.3, 0.4$ and 10000 values of β.

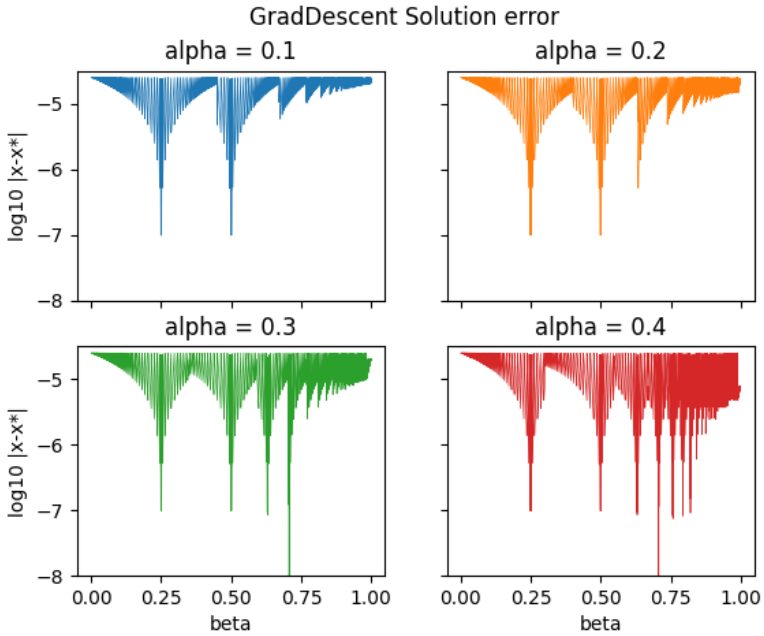


Figure 5: Comparison on the accuracy of the solutions obtained with GDA for 10000 values of β.

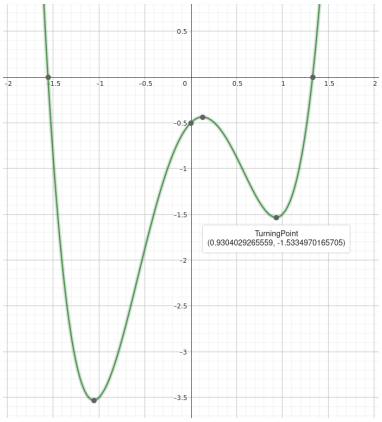


Figure 6: Plot of the curve $f(x) = 2x^4 - 4x^2 + x - 0.5$.

Part (c): Second example results

For the second example, we chose the values $\alpha = 0.1$ and $\beta = 0.8$ and ran the GDA from four different initial values. Taking into account that the optimal value is (appearently) an irrational number, with a very complicated exact expression, we will use the following approximation to compute the error:

$$x^* \approx -1.05745377073837789925780703035427198$$

The following table summarizes the results for this example. We can see that the errors are of the order of 10^{-6}, the same

x_0	-2	-0.5	0.5	2
n° steps	22	24	24	26
absolute error	$4.28 \cdot 10^{-6}$	$4.23 \cdot 10^{-6}$	$3.59 \cdot 10^{-6}$	$3.62 \cdot 10^{-6}$

Table 4: Result for the GDA applied to the second example, with gradient tolerance 10^{-4}.

as in the previous example, and the number of steps is also very similar with $\alpha = 0.1$.

Part (d): Compare previous results with Newton's method.

The "Newton-CG" method implementation of the `scipy.optimize.minimize` module was used. The gradients of the objective functions were always provided to the method, and the tolerance was set to 10^{-4}. Otherwise the comparison will not be fair.
For the first example, convergence towards the optimal solution was reached in just two iterations, with an exact value of 0.0 (thus, no numerical error).
The results for the second example are summarized in table 5. We can see that the performance in terms of accuracy and number of iterations is far superior for the negative initial points. However, for the positive initial points the method converges to a different solution, namely $x = 0.9304029$, that is actually a local minimum of the objective function (see fig. 6). This makes sense, as it is known that Newton's method does not have the capability of escaping local minima.

x_0	-2	-0.5	0.5	2
n° steps	6	5	6	5
solution	-1.05745377	-1.05745377	0.93040293	0.93040293
absolute error	$4.829 \cdot 10^{-10}$	$2.486 \cdot 10^{-10}$	1.9878	1.9878

Table 5: Results for the Newton-CG method applied to the second example.

In conclusion, Newton's method seems far superior than GDA for convex unconstrained optimization, but it is not robust for problems where the objective function may have several local minima.

Question 7

Network utility Consider the Network utility problem:

$$\begin{aligned} \text{maximize} \quad & W(x) = \sum_{r \in S_i} U_r(x_r) = \sum_{r \in S_i} log(x_r) \\ \text{subject to} \quad & Ax \leq C \\ & x \geq 0 \end{aligned}$$

Where $C = (1, 2, 1, 2, 1)^T$ and

$$A = \begin{bmatrix} 1 & 0 & 1 \\ 0 & 1 & 1 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \\ 1 & 0 & 0 \end{bmatrix}$$

Part (a): The Dual Problem

Clearly, the third and fourth row of the matrices A and C are redundant. But still, I wanted to try to encode the problem using the matrix notation. The Lagrangian of the problem considering the "useless" rows is

$$\mathcal{L}^1 = \lambda_1 (x_1 + x_3 - 1) + \lambda_2 (x_2 + x_3 - 2) - \lambda_3 - 2\lambda_4 + \lambda_5 (x_1 - 1) - \lambda_6 x_1 - \lambda_7 x_2 - \lambda_8 x_3 - \log(x_1) - \log(x_2) - \log(x_3).$$

If we take out the third and fourth row, it would become

$$\mathcal{L}^2 = \lambda_1 (x_1 + x_3 - 1) + \lambda_2 (x_2 + x_3 - 2) + \lambda_5 (x_1 - 1) - \lambda_6 x_1 - \lambda_7 x_2 - \lambda_8 x_3 - \log(x_1) - \log(x_2) - \log(x_3)$$

The derivatives w.r.t. the primal variables of both coincide:

$$\nabla_x \mathcal{L}^1 = \nabla_x \mathcal{L}^2 = \begin{bmatrix} \lambda_1 + \lambda_5 - \lambda_6 - \frac{1}{x_1} \\ \lambda_2 - \lambda_7 - \frac{1}{x_2} \\ \lambda_1 + \lambda_2 - \lambda_8 - \frac{1}{x_3} \end{bmatrix}$$

If we removed the dual variables corresponding to the non-negativity constraint, namely $\lambda_6, \lambda_7, \lambda_8$, we would get that the dual and the primal solutions are related by the following system of equations:

$$\lambda_1^* + \lambda_5^* = \frac{1}{x_1^*} \tag{5}$$

$$\lambda_2^* = \frac{1}{x_2^*} \tag{6}$$

$$\lambda_1^* + \lambda_2^* = \frac{1}{x_3^*} \tag{7}$$

Part (b): Solution using "useless" rows

The encoding of the problem using `CVXPY` was the following:

```
import cvxpy as cp

A = [[1, 1, 0, 0, 0],
     [0, 1, 0, 0, 0],
     [1, 0, 0, 0, 1]]
C = [1, 2, 1, 2, 1]
x = cp.Variable(3)
constaints = [A@x <= C, x >= 0]
prob = cp.Problem(cp.Maximize(cp.log(x[0]) + cp.log(x[1]) + cp.log(x[2])),
                  constraints=constraints)
prob.solve()
```

This approach yielded the solution

$$x_1 = 0.42264971, \quad x_2 = 1.57735029, \quad x_3 = 0.57735029$$

and the following values for the dual variables:

$$\lambda_1 = 1.73205063, \quad \lambda_2 = 0.633974601, \quad \lambda_5 = 3.25763413e-09,$$
$$\lambda_3 = 1.41565492e-09, \quad \lambda_4 = 7.35642950e-10,$$
$$\lambda_6 = 3.37549243e-09, \quad \lambda_7 = 9.16253816e-10, \quad \lambda_8 = 2.39939889e-09$$

Which is consistent with the system of equations derived in part a. Note that all the multipliers are basically zero, except for $\lambda_1, \lambda_2, \lambda_5$. The solution for the primal (and dual) problem that was obtained is

$$d^* = p^* = -0.95477125606744.$$

Part (c): Solution of simplified problem

The encoding of the problem removing the third and the fourth row was the following: The encoding of the problem using `CVXPY` was the following:

```
y = cp.Variable(3)
simp_cons = [y[0] + y[2] <= 1,
             y[0] + y[1] <= 2,
             y[2] <= 1, y>= 0]
prob = cp.Problem(cp.Maximize(cp.log(y[0]) + cp.log(y[1]) + cp.log(y[2])), constraints=simp_cons
prob.solve()
```

This approach yielded the solution

$$x_1 = 0.42264894, \quad x_2 = 1.57735105, \quad x_3 = 0.57735105$$

and the following information of the dual variables:

$$\lambda_1 = 1.7320483134403175, \quad \lambda_2 = 0.6339745314617544, \quad \lambda_5 = 6.437850296384749e-09,$$
$$\lambda_6 = 6.54467932e-09, \quad \lambda_7 = 1.77555380e-09, \quad \lambda_8 = 4.77958912e-09$$

The solution for the primal (and dual) problem that was obtained is

$$d^* = p^* = -0.9547712589294085.$$

We can only see a difference in any the values obtained in part b starting as low as the fourth decimal point.
In conclusion, it is clear that both formulations are practically equivalent, but we have also seen that the inclusion/exclusion of the two redundant rows can provoke some (very) small numerical differences in the result.

Question 8: Resource Allocation in wireless network

Consider the resource allocation problem

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{3} log(x_i) \\ \text{subject to} \quad & x_1 + x_2 \leq T_{12} \\ & x_1 \leq T_{23} \\ & x_3 \leq T_{32} \\ & T_{12} + T_{23} + T_{32} \leq 1 \\ \text{var} \quad & x_1, x_2, x_3, T_{12}, T_{23}, T_{32} \end{aligned}$$

and solve it using CVX.

Part (a): Primal solution

I used `CVXPY`, and encoded the problem as follows.

```
import cvxpy as cp
x = cp.Variable(3)
T = cp.Variable(3) #[T_12, T_23, T_32]
cons = [x[0] + x[1] <= T[0],
              x[0] <= T[1],
              x[2] <= T[2],
              T[0] + T[1] + T[2] <= 1]
prob = cp.Problem(cp.Maximize(cp.log(x[0]) + cp.log(x[1]) + cp.log(x[2])),
                  constraints=cons)
prob.solve()
```

The traffic allocated to each user are:

$$x_1 = 0.1666... = \frac{1}{6}, \quad x_2 = 0.333... = \frac{1}{3}, \quad x_3 = 0.333... = \frac{1}{3}$$

The percentage of time each link is used are:

$$T_{12} = 0.5 = \frac{1}{2}, \quad T_{23} = 0.1666... = \frac{1}{6}, \quad T_{32} = 0.333... = \frac{1}{3}$$

And the primal solution obtained is:

$$p^* = -3.988984047216252$$

Part (b): Dual solution

The Lagrangian for this problem is

$$\mathcal{L}(x, T, \lambda) = \lambda_1 (-T_{12} + x_1 + x_2) + \lambda_2 (-T_{23} + x_1) + \lambda_3 (-T_{32} + x_3) + \lambda_4 (T_{12} + T_{23} + T_{32} - 1) - \log(x_1) - \log(x_2) - \log(x_3)$$

We can find an expression of the dual solutions by differentiating $\mathcal{L}$ with respect to the primal variables,

$$\nabla_{x,T} \mathcal{L}(x, T, \lambda) = \begin{bmatrix} \lambda_1 + \lambda_2 - \frac{1}{x_1} \\ \lambda_1 - \frac{1}{x_2} \\ \lambda_3 - \frac{1}{x_3} \\ -\lambda_1 + \lambda_4 \\ -\lambda_2 + \lambda_4 \\ -\lambda_3 + \lambda_4 \end{bmatrix}$$

And by equating to zero, this tells us that $\lambda_4^* = \lambda_3^* = \lambda_2^* = \lambda_1^*$, and that the variables T_{ij} are actually some sort of "slack" variables. Therefore,

$$\lambda_4^* = \lambda_3^* = \lambda_2^* = \lambda_1^* = \frac{1}{x_2^*} = \frac{1}{x_3^*}.$$

The solution we got from `CVXPY` for the Lagrangian multipliers is $\lambda_i = 2.999... = 3 \quad \forall i \in \{1, 2, 3, 4\}$, which is consistent with our theoretical analysis.