MIRI-TOML Programming exercises

Marc Catrisse Pérez

April 4, 2021

1 Introduction

In this document, I'm going to solve the proposed optimization exercises. All of them are solved using Python 3.8 and the following optimization libraries:

- Scipy: `https://docs.scipy.org/doc/scipy/reference/tutorial/optimize.html`
- CVXpy: `https://www.cvxpy.org/`

2 Exercises

2.1 Ex 1

Before trying to solve the proposed optimization problem, we are going to determine if the problem is convex or not. Calculating the hessian Matrix we should be able to determine that the problem is convex if it results in a positive definite matrix.

$$H_x = \begin{bmatrix} \frac{\partial^2 f}{\partial^2 x_1} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial^2 x_2} \end{bmatrix}$$

$$\frac{\partial^2 f}{\partial^2 x_1} = e^{x_1} * (9 + 16x_1 + 10x_2 + 4x_1^2 + 2x_2^2 + 4x_1x_2)$$

$$\frac{\partial^2 f}{\partial x_1 \partial x_2} = e^{x_1} * (6 + 4x_2 + 4x_1)$$

$$\frac{\partial^2 f}{\partial x_2 \partial x_1} = e^{x_1} * (4x_2 + 4x_1 + 6)$$

$$\frac{\partial^2 f}{\partial^2 x_2} = 4e^{x_1}$$

$$det(H_x) = [4e^{2x_1} * (9 + 20x_1 + 12x_2 + 4x_1x_2)] - [e^2 x_1 * (64x_1 + 48x_2 + 16x_2^2 + 32x_1x_2 + 36)]$$

Just checking the constants we can determine that $det(H_x) < 0$ meaning that the Hessian Matrix isn't positive semi-define, hence the problem is not convex. Graphically, we can observe in Figure 1 that the problem isn't convex in the area delimited by the constraints (see the 2 black points and the line)

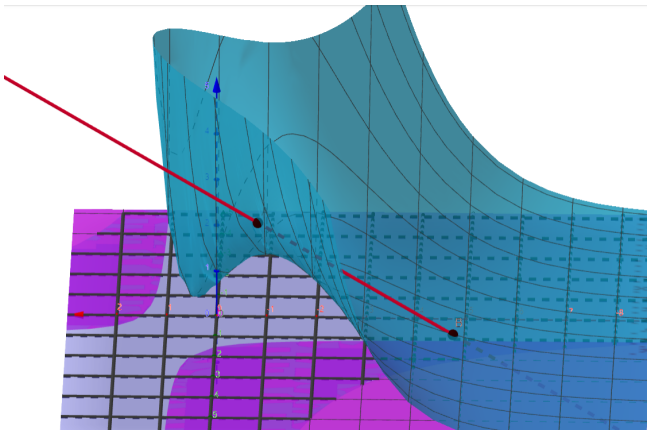

Figure 1: Graphical representation of the objective function (blue) + constraints(purple and grey)

Using Scipy we are going to try minimize the objective function(scipy.minimzie) with the SLSQP method. As initial points we are going to use the following: [0,0], [10,20], [-10,1], [-30,-30]. Obtaining the following output:

```
    X0 -> [0 0]
Optimization terminated successfully    (Exit mode 0)
            Current function value: 0.02355037962417486
            Iterations: 18
            Function evaluations: 56
            Gradient evaluations: 18
     fun: 0.02355037962417486
     jac: array([ 0.01839703, -0.00228436])
 message: 'Optimization terminated successfully'
    nfev: 56
     nit: 18
    njev: 18
  status: 0
 success: True
       x: array([-9.54740503,  1.04740503])
--------------------------------

X0 -> [10 20]
Inequality constraints incompatible    (Exit mode 4)
            Current function value: 44956016.68720051
            Iterations: 1
            Function evaluations: 3
            Gradient evaluations: 1
     fun: 44956016.68720051
     jac: array([48480251.5,  2687229. ])
 message: 'Inequality constraints incompatible'
    nfev: 3
     nit: 1
    njev: 1
```

```
  status: 4
 success: False
       x: array([10., 20.])
---------------------------------

X0 -> [-10   1]
Optimization terminated successfully    (Exit mode 0)
            Current function value: 0.0235503796241749
            Iterations: 4
            Function evaluations: 12
            Gradient evaluations: 4
     fun: 0.0235503796241749
     jac: array([ 0.01839703, -0.00228436])
 message: 'Optimization terminated successfully'
    nfev: 12
     nit: 4
    njev: 4
  status: 0
 success: True
       x: array([-9.54740503,  1.04740503])
---------------------------------

X0 -> [-30 -30]
Optimization terminated successfully    (Exit mode 0)
            Current function value: 0.02355037962417503
            Iterations: 14
            Function evaluations: 43
            Gradient evaluations: 14
     fun: 0.02355037962417503
     jac: array([ 0.01839703, -0.00228436])
 message: 'Optimization terminated successfully'
    nfev: 43
     nit: 14
    njev: 14
  status: 0
 success: True
       x: array([-9.54740503,  1.04740503])
---------------------------------
```

As we can observe, the algorithm is able to find the same solution([-9.54, 1.05]) for all proposed initial points, except for [10, 20], because it violates the stated constraints. Notice that, we are obtaining the best performance (4 iterations) using an initial point ([-10,1]) quite close to the optimal one. Whereas [-30, -30] and [0,0], performs quite poorly, with 14 and 18 iterations respectively. Going back to the objective function plot (Figure 1) we can see that actually [0,0] it seems to be a local minimum, which could explain why it performs worse than [-30, -30]. Following, we are going to give the Jacobian as an input to the optimization function, in order to see if the method speeds up. Notice that now we are skipping the [10, 20] initial point, as it's incompatible with the constraint and it doesn't give any relevant information.

```
    X0 -> [0 0]
Optimization terminated successfully    (Exit mode 0)
            Current function value: 0.0235503796241745
            Iterations: 18
            Function evaluations: 20
            Gradient evaluations: 18
     fun: 0.0235503796241745
     jac: array([ 0.01839703, -0.00228436])
 message: 'Optimization terminated successfully'
    nfev: 20
     nit: 18
    njev: 18
  status: 0
 success: True
       x: array([-9.54740503,  1.04740503])
--------------------------------

X0 -> [-10   1]
Optimization terminated successfully    (Exit mode 0)
            Current function value: 0.02355037962417493
            Iterations: 4
            Function evaluations: 4
            Gradient evaluations: 4
     fun: 0.02355037962417493
     jac: array([ 0.01839703, -0.00228436])
 message: 'Optimization terminated successfully'
    nfev: 4
     nit: 4
    njev: 4
  status: 0
 success: True
       x: array([-9.54740503,  1.04740503])
--------------------------------

X0 -> [-30 -30]
Optimization terminated successfully    (Exit mode 0)
            Current function value: 3.060773017619694
            Iterations: 29
            Function evaluations: 69
            Gradient evaluations: 28
     fun: 3.060773017619694
     jac: array([11.22002239, -0.7473713 ])
 message: 'Optimization terminated successfully'
    nfev: 69
     nit: 29
    njev: 28
  status: 0
 success: True
       x: array([ 1.18249728, -1.73976691])
--------------------------------
```

All this output information can be resumed in the following tables

Initial point	Iterations	Func eval
[0,0]	18	46
[-10,1]	4	12
[-30,30]	14	43

Table 1: Performance indicators obtained for the default execution

When setting the jacobian as an input the overall number of function evaluations decrease, except for [-30, -30] point, which increases the number of iterations drastically and hence the number of function evaluations.

Initial point	Iterations	Func eval
[0,0]	18	20
[-10,1]	4	4
[-30,-30]	29	69

Table 2: Performance indicators obtained using the Jacobian

2.2 Ex 2

At first glance, we can assure that the proposed optimization problem ($f(x) = x_1^2 + x_2^2$) is convex. Because, it's a quadratic function, that can be represented as $x^T P x$ where $P = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$. The graphical representation (Figure 2) confirms it, showing an inverse cone that it's clearly convex.

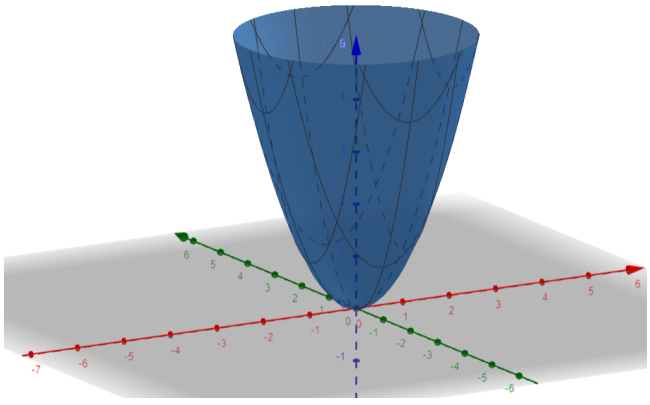

Figure 2: Graphical representation of the objective function (blue)

In this exercise, we are going to test the solver with 2 different initial points, [0,1] and [6, 10]. As you may notice, the [0, 1] point is unfeasible, because it doesn't satisfy the following constraint.

$$-x_1^2 + x_2 <= 0$$

When running the optimization solver with the default parameters and SLSQP as method, we are obtaining the following output:

```
    X0 -> [0 1]
     fun: 2.000000008876196
     jac: array([2., 2.])
 message: 'Optimization terminated successfully'
    nfev: 12
     nit: 4
    njev: 4
  status: 0
 success: True
       x: array([1., 1.])
---------------------------------

X0 -> [ 6 10]
     fun: 2.0000000707510583
     jac: array([2.00000003, 2.00000003])
 message: 'Optimization terminated successfully'
    nfev: 27
     nit: 10
    njev: 9
  status: 0
 success: True
       x: array([1.00000002, 1.00000002])
---------------------------------
```

Actually, we expected [0,1] optimization to fail,because it's unfeasible. But it worked, and solved the problem in just 4 iterations. In the other hand when running the optimization problem giving the Jacobian as input, we are getting te following ouput.

```
    X0 -> [0 1]
     fun: 1.0
     jac: array([0., 2.])
 message: 'Positive directional derivative for linesearch'
    nfev: 1
     nit: 5
    njev: 1
  status: 8
 success: False
       x: array([0., 1.])
---------------------------------

X0 -> [ 6 10]
     fun: 2.000000015193007
     jac: array([2.00000001, 2.00000001])
 message: 'Optimization terminated successfully'
    nfev: 22
     nit: 12
    njev: 11
```

```
  status: 0
 success: True
       x: array([1., 1.])
--------------------------------
```

As you may notice, now the solver returns an error for the [0,1] point: "Positive directional derivative for linesearch". Showing that something is wrong with the Jacobian, and the direction is setting isn't helping to solve the problem. As a summary we can see the performance of each execution in the following tables

Initial point	Iterations	Func eval
[0,1]	4	12
[6,10]	10	27

Table 3: Performance indicators obtained using the default parameters

Initial point	Iterations	Func eval
[0,1]	5	1
[6,10]	12	22

Table 4: Performance indicators obtained using the Jacobian

As we can observe, the Jacobian doesn't clearly improve the performance in [6,10] case, which it's estrange, as it seems that the direction it's not set well...

2.3 Ex 3

In this exercise, we are going to change the solver library. Now we are going to use the CVXPy library (`https://www.cvxpy.org/`), that solves Convex Optimization Problems. The objective function it's the same from the previous exercise, so we can confirm that its convex, for the reasons stated in ex 2.

First we are going to solve the problem using Scipy and $x_0 = [6, 10]$ and then cvxpy.

```
    ----------Scipy-------------
X0 -> [ 6 10]
Optimization terminated successfully    (Exit mode 0)
            Current function value: 0.6923076923076923
            Iterations: 4
            Function evaluations: 12
            Gradient evaluations: 4
     fun: 0.6923076923076923
     jac: array([1.38461538, 0.92307697])
 message: 'Optimization terminated successfully'
    nfev: 12
     nit: 4
    njev: 4
  status: 0
 success: True
```

```
       x: array([0.69230768, 0.46153848])

    ----------CVXPy-------------
solve 0.6923076920047873
status: optimal
optimal value p* =  0.6923076920047873
optimal var: x1 =  0.6923081958456881  x2 =  0.4615377063422878
optimal dual variables lambda1 =  [1.39314258e-10]
optimal dual variables lambda2 =  0.46154291380933704
```

As expected, we are obtaining practically the same results using CVXPy.

2.4 Ex 4

In this exercise, we are still practising a bit using CVXPy. Now we are going to solve a single variable quadratic objective function $(x^2 + 1)$ that, as stated before, it is a Convex Optimization Problem. This time, we are getting the following output:

```
solve 4.999999979918552
status: optimal
optimal value p* =  4.999999979918552
optimal var: x1 =  1.9999999968611173
optimal dual variables lambda1 =  [2.00003221]
```

2.5 Ex 5

This exercise is quite similar to the last one, we have a Quadratic objective function with also quadratic constraints. But this time the problem contains 2 different variables: x_1 and x_2 We are getting the following output:

```
solve 0.9999567689686263
status: optimal
optimal value p* =  0.9999567689686263
optimal var: x1 =  0.9999762883789091  x2 =  -4.1230398417949826e-11
optimal dual variables lambda1 =  [22956.06950301]
optimal dual variables lambda2 =  [22956.06950647]
```

2.6 Ex 6

This exercise is quite different from the rest, we are going to test how it works a descent gradient algorithm. Starting from a random point, we are going to move forward in the negative direction of the gradient to reach the local/global minima.

$$x_{(k+1)} = x_k + t\Delta x$$

where t = step size and Δ = negative direction of the gradient of the function.

In order to test this algorithm we wrote a python program that implements the previously described Gradient Descent Method using the Backtracking Line Search.

```
beta = 0.8 # between 0, 1 (0.1 very crude search / 0.8 less crude)
f = lambda x: (2 * x ** 2 - 0.5)
fdx = lambda x: (4 * x)
t = 1
x0 = 3
n = 1
stop_criterion = 10 ** -4
#Backtrack Line Search
while (f(x0 + t * (-fdx(x0))) > stop_criterion):
    t = beta * t
    n += 1
res = x0 + t * (-fdx(x0))
```

First we are going to test the algorithm with the following input:

- $f(x) = 2x^2 - 0.5$
- initial point $x_0 = 3$
- accuracy $n = 10^-4$

Obtaining the following output:

```
A)
N -> 7
Final result -> -0.14572800000000097
Final accuracy -> -0.45752670003199947
```

Now, we are going to change the parameters and test the program with several initial points.

- $f(x) = 2x^4 - 4x^2 + x - 0.5$
- initial points: $x_0 = [-2, -0.5, 0.5, 2]$
- accuracy $n = 10^-4$

Obtaining the following output:

```
B)
x0 -> -2
N -> 12
Final result -> 1.2298154065920017
Final accuracy -> -0.7449829222651432
--------------------------
x0 -> -0.5
N -> 6
Final result -> -1.5485760000000004
Final accuracy -> -0.13927797011934806
--------------------------
x0 -> 0.5
N -> 4
Final result -> 1.3192000000000004
Final accuracy -> -0.08474545897389785
```

```
-------------------------
x0 -> 2
N -> 12
Final result -> -1.3672543600640021
Final accuracy -> -2.355595594150049
-------------------------
```

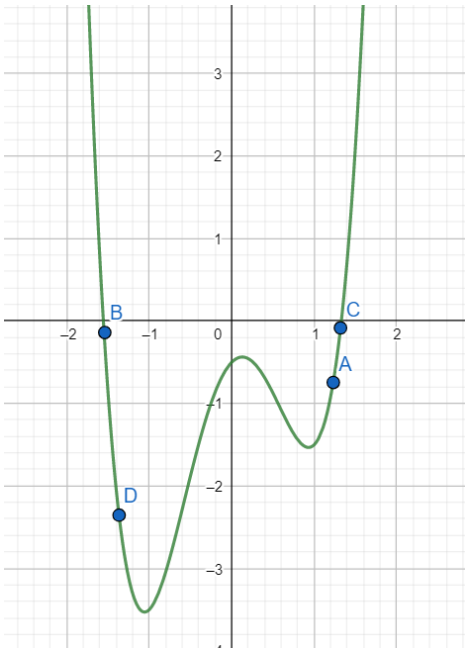

Figure 3: Graphical representation of the objective function + optimal points found (A = $x_0(-2)$, B = $x_0(-0.5)$, C = $x_0(0.5)$, D = $x_0(2)$

It seems that the β of 0.8 is maybe too greedy, for example, for $x_0(-2)$ it clearly passes to the positive x dimension and approximates a local minima, when the global minima is around x = -1.

Now, we are going to implement the Newton's Method

```
f = lambda x: (2 * x ** 4 - 4 * x ** 2 + x - 0.5)
fdx = lambda x: (8 * x ** 3 - 8 * x + 1)
f2dx = lambda x: (24 * x ** 2 - 8)
lambda_sq_fn = lambda x: (fdx(x) * f2dx(x)**-1 * fdx(x))
new_step_fn = lambda x: (-f2dx(x)**-1 * fdx(x))
stop_criterion = 10 ** -4
for x0 in [-2, -0.5, 0.5, 2]:
    print("x0 -> " + str(x0))
    x = x0
    n = 0
    while lambda_sq_fn(x) > stop_criterion:
        new_step = new_step_fn(x)
        t = backtrack_t(x, f, fdx)
        x = x + t*new_step
```

```
        n += 1
    res = x
    print("N -> " + str(n))
    print("Final result -> " + str(res))
    print("Final accuracy -> " + str(f(res)))
    print("--------------------------")
```

Next, we are going to test it with the previous input data, in order to provide a final comparison.

```
C)
x0 -> -2
N -> 19
Final result -> -1.0590435754424643
Final accuracy -> -3.529483443480867
--------------------------
x0 -> -0.5
N -> 0
Final result -> -0.5
Final accuracy -> -1.875
--------------------------
x0 -> 0.5
N -> 0
Final result -> 0.5
Final accuracy -> -0.875
--------------------------
x0 -> 2
N -> 19
Final result -> 0.9314629636958623
Final accuracy -> -1.5334898298723156
--------------------------
```

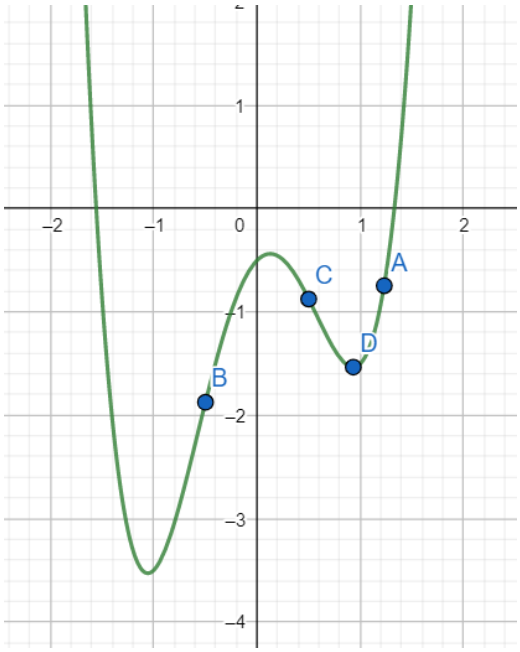

Figure 4: Graphical representation of the objective function + optimal points found (A = $x_0(-2)$, B = $x_0(-0.5)$, C = $x_0(0.5)$, D = $x_0(2)$

Notice that now, all points found are nearer to the local/global minima. Furthermore, the number of iterations to values near 0 (0.5 and -0.5) is being reduced to 0. Although, for -2 and 2 is being increased from 12 to 19.

2.7 Ex 7

In this exercise we are going back to cvxpy library in order to solve the proposed problem.

maximize $log(x_1) + log(x_2) + log(x_3)$
st $x_1 + x_3 <= 1$
$x_1 + x_2 <= 2$
$x_3 <= 1$
$x_1, x_2, x_3 >= 0$

We modeled the previously described problem, obtaining the following output:

```
solve 0.9547712365062518
status: optimal
optimal value p* =  0.9547712365062518
optimal var: x1 =  0.4226489442893967
optimal var: x2 =  1.577351049782123
optimal var: 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
```

2.8 Ex 8

In this final exercise we will use CVXPy to solve the proposed Resource Allocation problem with dual variables. Obtaining the following output:

```
solve 3.9889840093737394
status: optimal
optimal value p* =  3.9889840093737394
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
```

3 Appendix - Code

All code mentioned in this document is included in the delivery zip file