TOML: Programming Exercises

Arnau Abella
Universitat Politècnica de Catalunya

March 25, 2021

Exercise 1

(a) We can check convexity by checking the second-order conditions.

$$f(x_1, x_2) = e_1^x(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1)$$
$$(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}) = (e^{x_1}(4x_1^2 + 4x_1(x_2+2)2x_2^2 + 6x_2 + 1,\ e_1^x(4x_1 + 4x_2 + 2))$$
$$H_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} e_1^x(4x_1^2 + 4x_1(x_2+2) + 2x_2^2 + 10x_2 + 9) & e_1^x(4x_1 + 4x_2 + 6) \\ 2e_1^x(2x_1 + 2x_2 + 3) & 4e_1^x \end{bmatrix}$$

The Hessian matrix it is not positive nor negative semidefinite. Therefore, $f(x_1, x_2)$ is not convex nor concave.

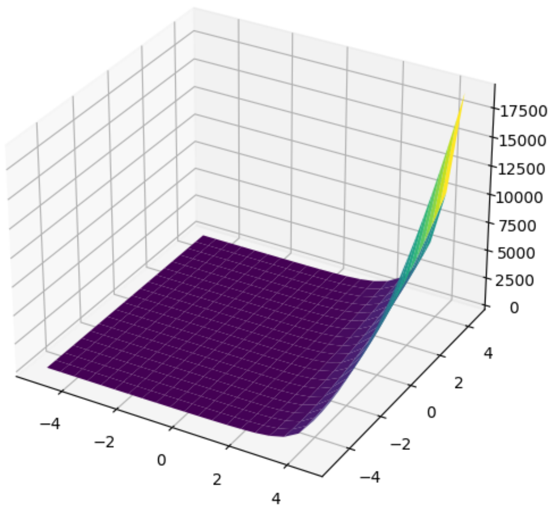

Figure 1: 3D plot of $f(x_1, x_2) = e_1^x(4x_1^2 + 2x_2^2 + 4x_1x_2 + 2x_2 + 1)$

(b) Depending on the starting point we get different optimal points since the objective function is not convex (see plot 1).

x_0	$x*$	$p*$	Iter.
(0, 0)	(-24.9, 45.4)	$3.29e^{-8}$	6
(10, 20)	(-0.0047, 8.95)	0.0	2
(-10, 1)	(-28.1, 21.4)	$1.03e^{-9}$	28
(-30, -30)	(-28.5, 10.46)	$9.5e^{-10}$	34

(c) The method has speed up

x_0	Execution time	Execution time (with jacobi)
(0, 0)	0.0032”	0.00185”
(10, 20)	0.0014”	0.0011”
(-10, 1)	0.0094”	0.0065”
(-30, -30)	0.011”	0.0077”

```python
#!/usr/bin/env python

import numpy as np
import matplotlib.pyplot as plt
from matplotlib import cm
from mpl_toolkits.mplot3d import Axes3D

def objFun(x1,x2):
    return np.exp(x1)*(4*x1**2 + 2*x2**2 + 4*x1*x2 + 2*x2 + 1)

x1 = np.arange(-5, 5, 0.5)
x2 = np.arange(-5, 5, 0.5)
x1, x2 = np.meshgrid(x1, x2)
z = objFun(x1,x2)
fig = plt.figure()
ax = Axes3D(fig)
ax.plot_surface(x1, x2, z, rstride=1, cstride=1, cmap=cm.viridis)
plt.savefig('./report/figs/exercise1_plot.png')
```

Listing 1: Code used for ploting exercise 1.

```python
#!/usr/bin/env python

import math
import numpy as np
from scipy.optimize import minimize
from timeit import timeit
from numdifftools import Jacobian, Hessian

def obj_fun(x):
    return math.exp(x[0])*(4*x[0]**2 + 2*x[1]**2 + 4*x[0]*x[1] + 2*x[1] +
    ↪  1)

def fun_jac(x):
    dx = math.exp(x[0])*(4*x[0]**2 + 4*x[0]*(x[1] + 2) + 2*x[1]**2 +
    ↪  6*x[1] + 1)
    dy = math.exp(x[0])*(4*x[0] + 4*x[1] + 2)
    return np.array([dx, dy])

# Slow, it seems it is computed each time..
fun_jac_true = Jacobian(obj_fun)

def fun_hess(x, a):
    dxx = math.exp(x[0])*(4*x[0]**2 + 4*x[0]*(x[1] + 2)*2*x[1]**2 +
    ↪  10*x[1] + 9)
    dxy = math.exp(x[0])*(4*x[0] + 4*x[1] + 6)
    dyx = 2*math.exp(x[0])*(2*x[0] + 2*x[1] + 3)
    dyy = 4*math.exp(x[0])
    return np.array([[dxx, dxy],[dyx, dyy]])

ineq_cons = {'type': 'ineq',
             'fun' : lambda x: np.array( -x[0]*x[1]+x[0]+x[1]-1.5
                                        , x[0]*x[1]+10)
            }

x0s = np.array([[.0, .0],[10.0,20.0],[-10.0, 1.0],[-30.0,-30.0]])

for x0 in x0s:
    print('=== Regular ===')
    res = timeit(minimize, obj_fun, x0, method='SLSQP',
    ↪  constraints=[ineq_cons], options={'ftol': 1e-9, 'disp': True})
    print(f'[x1*, x2*] = {res.x}\n')

    print('=== Optimized ===')
    res = timeit(minimize, obj_fun, x0, method='SLSQP', jac=fun_jac,
    ↪  constraints=[ineq_cons], options={'ftol': 1e-9, 'disp': True})
    print(f'[x1*, x2*] = {res.x}\n')
```

Listing 2: `exercise1.py`

Exercise 2

(a) We can check convexity by checking the second-order conditions.

$$
\begin{aligned}
&f(x_1, x_2) = x_1^2 + x_2^2 \\
&\left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}\right) = (2x, 2y) \\
&H_f = \begin{bmatrix} \frac{\partial^2 f}{\partial x_1^2} & \frac{\partial^2 f}{\partial x_1 \partial x_2} \\ \frac{\partial^2 f}{\partial x_2 \partial x_1} & \frac{\partial^2 f}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 2 & 0 \\ 0 & 2 \end{bmatrix}
\end{aligned}
$$

Alternatively, you can check if $P \geq 0$ in the quadratic form $\frac{1}{2}x^T Px + q^T x + r$.

$$
f(x_1, x_2) = x_1^2 + x_2^2 = \begin{bmatrix} x_1 & x_2 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \end{bmatrix}
$$

where $P = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \geq 0$

The Hessian matrix is positive and all inequalities are convex. Therefore, $f(x_1, x_2)$ is positive semi-definite.

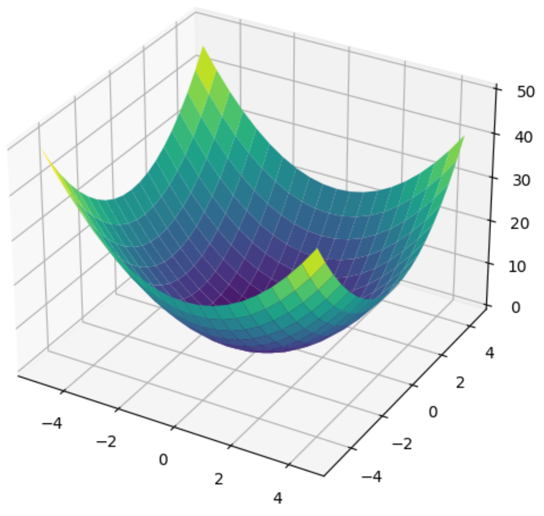

Figure 2: 3D plot of $f(x_1, x_2) = x_1^2 + x_2^2$

(b) The result for the feasible initial point is $x* = (1.0, 1.0)$ and $p* \approx 2$. For the unfeasible initial point, the algorithm exits giving the initial point. The number of steps untill convergence is 13. The code for this exercise can be found at `exercise2.py` and `exercise2_plot.py`.

(c) The number of steps with giving the Jacobian as an input is 14 but the number of functions is 14, way smaller than before (67). My hypothesis is that the giving initial point is near the optimal solution and so the number of steps is small. Probably, the approximation of the Jacobian is also good.

Exercise 3

We can check convexity by checking the second-order conditions of the objective function and the inequalities. We have already checked the convexity of $f(x_1, x_2) = x_1^2 + x_2^2$ and its domain. Then, we only need to check the convexity of the inequalities. The second inequality is convex since it is linear and $x_1^2 + x_1 x_+ x_2^2 \leq 3$ is a quadratic equation with $P \geq 0$, so it is convex.

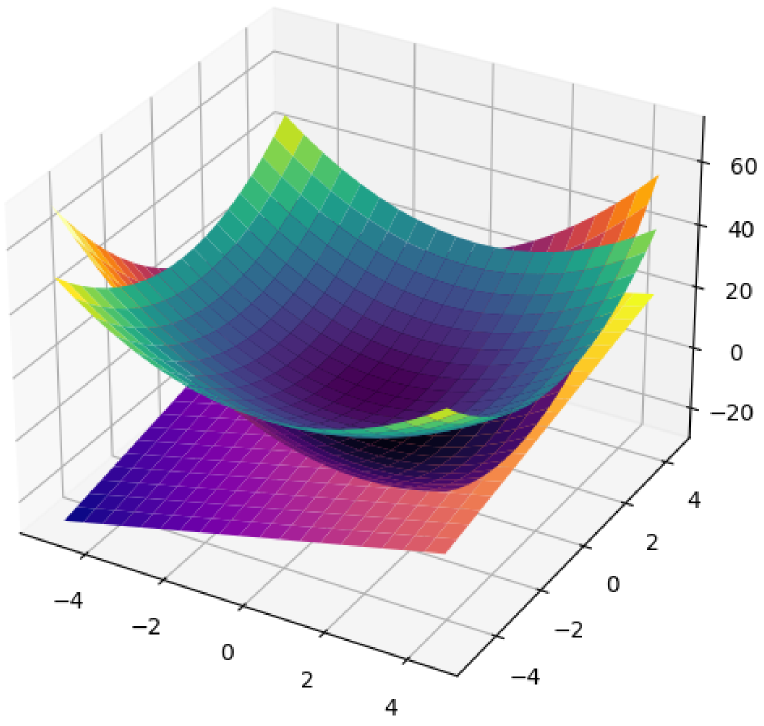

Figure 3: 3D plot of the COP

The optimal point is $x* = (0.69230769, 0.46153846)$ with optimal value $p* = 0.6923076923076928$. The convergence is after 3 iterations since $x_0 = (0, 0)$.

Exercise 4

```
#!/usr/bin/env python

import numpy as np
from cvxopt import matrix, solvers, spdiag

# matrices in cvxopt are weirdly indexed.

# >>> m = matrix([1,2,3,4], (2,2)))
# >>> print(m)
# [1 3]
# [2 4]

# m[0] = 1
# m[1] = 2
# m[2] = 3
# print(m[0,:]) = [1 3]

# >>> print(matrix(np.array([[1,2],[3,4]])))
# [1 2]
# [3 4]


m = 1 # #inequalities
n = 1 # #variables

# return None if x is not in the domain of f_0
def F(x=None, z=None):
    if x is None:
        return m, matrix(3.0, (n, 1))
    # x is always in the domain
    f_0 = x[0]**2 + 1
    f_1 = (x[0] - 2)*(x[0] - 4)
    f = matrix([f_0, f_1], (m+1, 1))
    Df_0 = 2*x[0]
    Df_1 = 2*(x[0] - 3)
    Df = matrix([Df_0, Df_1]) # (m+1, 1)
    if z is None:
        return f, Df
    else:
        H = spdiag([z[0]*2 + z[1]*2])
        return f, Df, H

res = solvers.cp(F)
print(res['status'])
print(res['x'])
```

Listing 3: `exercise4.py`

Exercise 5

```python
#!/usr/bin/env python

import numpy as np
from cvxopt import matrix, solvers, spdiag

solvers.options['maxiters'] = 1000

m = 2 # inequalities
n = 2 # variables

def F(x=None, z=None):
    if x is None:
        return m, matrix([2.0, 1.0], (n, 1))
    f_0 = x[0]**2 + x[1]**2
    f_1 = (x[0] - 1)**2 + (x[1] - 1)**2 - 1
    f_2 = (x[0] - 1)**2 + (x[1] + 1)**2 - 1
    f = matrix([f_0, f_1, f_2], (m+1, 1))
    Df_0 = [2*x[0], 2*x[1]]
    Df_1 = [2*(x[0] - 1), 2*(x[1] - 1)]
    Df_2 = [2*(x[0] - 1), 2*(x[1] + 1)]
    Df = matrix(np.array([Df_0, Df_1, Df_2]))
    if z is None:
        return f, Df
    else:
        # H = spdiag([H_0, H_1, H_2])
        H_0 = z[0] * matrix([2, 0, 0, 2], (n, n))
        H_1 = z[1] * matrix([2, 0, 0, 2], (n, n))
        H_2 = z[2] * matrix([2, 0, 0, 2], (n, n))
        H = H_0 + H_1 + H_2
        return f, Df, H

res = solvers.cp(F)
print(res['status'])
print(res['x'])
```

Listing 4: exercise5.py

Exercise 6

The results of Gradient Descent Methdod are shown in . The results from the Newton's Method are similar except for the number of iterations which is larger. This is contradictory which may be caused by a bug in the code.

	x_0	$\hat{x}*$	$\hat{p}*$	η_{final}	iterations	execution time (s)
$2x^2 - 0.5$	(3.0)	$9.64e^{-6}$	-0.4999	$3.86e^{-5}$	10	0.00058
$2x^4 - 4x^2 + x - 0.5$	(-2.0)	-1.0574	-3.529	$5.816e^{-5}$	13	0.415
	(-0.5)	-1.0575	-3.529	$7.96e^{-5}$	13	0.398
	(0.5)	-1.057	-3.529	$7.339e^{-5}$	14	0.3916
	(2.0)	0.930	-1.533	$5.873e^{-5}$	11	0.3148

Table 1: Gradient Descent Method results

```python
import math
import numpy as np
from scipy.optimize import minimize
from numdifftools import Jacobian, Hessian
from backtracking import backtrackingLineSearch
import numpy.linalg as LA

def gdm(objFun, x0, jacobian=None, accuracy=1e-4):
    """Gradient Descent Method.
    """
    if jacobian is None:
        jacobian = Jacobian(objFun)
    def getStep(x): return -jacobian(x)
    def getEta(d): return LA.norm(d, ord=2)
    steps = 0 # number of iterations
    x = x0
    d = getStep(x)
    eta = getEta(d)
    while(eta > accuracy):
        t = backtrackingLineSearch(objFun, x, jacobian=jacobian)
        x = x + t*d
        d = getStep(x)
        eta = getEta(d)
        steps += 1
    return (x, objFun(x), steps, eta)
```

Listing 5: Gradient Descent Method

```python
import math
import numpy as np
from scipy.optimize import minimize
from numdifftools import Jacobian, Hessian
from backtracking import backtrackingLineSearch
from numpy.linalg import inv

def newtons(objFun, x0, jacobian=None, hessian=None, accuracy=1e-4):
    """Newton's Method"""
    # For some reason, Jacobian returns the jacobian in matrix form i.e.
    ↪  [[]]
    if jacobian is None: jacobian = Jacobian(objFun)
    if hessian is None: hessian = Hessian(objFun)
    def getLambdaSquare(x):
        j = jacobian(x).reshape(-1)
        hinv = inv(hessian(x))
        return j.T @ hinv @ j
    def getStep(x):
        j = jacobian(x).reshape(-1)
        hinv = inv(hessian(x))
        return - hinv @ j
    steps = 0
    x = x0
    d = getStep(x)
    l = getLambdaSquare(x)
    # TODO l may be negative...
    while(l/2 > accuracy):
        t = backtrackingLineSearch(objFun, x, jacobian=jacobian)
        x = x + t*d
        d = getStep(x)
        l = getLambdaSquare(x)
        steps += 1
    return (x, objFun(x), steps, l)
```

Listing 6: Newton's Method

```python
from numdifftools import Jacobian

def backtrackingLineSearch(objFun, x, alpha=0.25, beta=0.85,
↪   jacobian=None):
    """Backtracking Line Search"""
    t = 1
    if jacobian is None: jacobian = Jacobian(objFun)
    j = jacobian(x)
    # TODO do you also use jacobian for newton's ?
    while(objFun(x + t*(-j)) >= objFun(x) + alpha*t*j.T*(-j)):
        t *= beta
    return t
```

Listing 7: Backtracking Line Search

Exercise 7

- $p* = 0.9547$
- $x* = (0.42264894, 1.57735105, 0.57735105)$
- $\lambda* = (1.732, 0.633, 6.437e^{-9}, 6.544e^{-9}, 1.775e^{-9}, 4.779e^{-9})$

```
#!/usr/bin/env python

import math
from numpy import array
import cvxpy as cp

x = cp.Variable(3, name="x")
objective_fn = - cp.log(x[0]) - cp.log(x[1]) - cp.log(x[2])
constraints = [ x[0] + x[2] <= 1
              , x[0] + x[1] <= 2
              , x[2] <= 1
              , x[0] >= 0
              , x[1] >= 0
              , x[2] >= 0
              ]
assert objective_fn.is_dcp()
assert all(constraint.is_dcp() for constraint in constraints)
problem = cp.Problem(cp.Minimize(objective_fn), constraints)
problem.solve()
print("status: ", problem.status)
print(f'x*: {x.value}')
print("p*: ",  problem.value)
print("Dual values: ", [*map(lambda x: x.dual_value, constraints)])
```

Listing 8: `exercise_7.py`

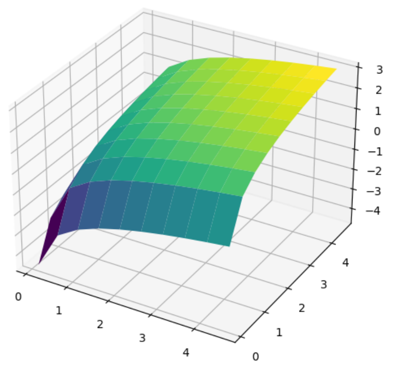

Figure 4: 3D plot of $\log x + \log y$

Exercise 8

- $p* = -3.99$
- $x* = (0.16666665, 0.33333335, 0.33333335)$
- $R* = (0.5, 0.16666665, 0.33333335)$
- $\lambda* = (2.999, 2.999, 2.999, 2.999)$

```python
#!/usr/bin/env python

import math
from numpy import array
import cvxpy as cp

x = cp.Variable(3, name="x")
R = cp.Variable(3, name="R") # R_12, R_23, R_32
objective_fn = cp.log(x[0]) + cp.log(x[1]) + cp.log(x[2])
constraints = [ x[0] + x[1] <= R[0]
              , x[0] <= R[1]
              , x[2] <= R[2]
              , R[0] + R[1] + R[2] <= 1
              ]
assert objective_fn.is_dcp()
assert all(constraint.is_dcp() for constraint in constraints)
problem = cp.Problem(cp.Maximize(objective_fn), constraints)
problem.solve()
print("status: ", problem.status)
print("p*: ",  problem.value)
print(f'x*: {x.value}')
print(f'R*: {R.value}')
print("Dual values: ", [*map(lambda x: x.dual_value, constraints)])
```

Listing 9: exercise_8.py