Numerical Optimization - Assignment 6

David Alvear(187594) - Nouf Farhoud(189731)

In [ ]:
```
import numpy as np
```

1. **Nonnegativity of the Lagrange Multipliers.** Explain with the aid of an appropriate diagram why it is not possible for the Lagrange multiplier of an inequality constraint to be negative at an optimal point.

The Lagrangian function $L(x, \lambda)$ is defined as:
$L(x, \lambda) = f(x) + \lambda \cdot g(x)$

At an optimal point, the stationary condition for the Lagrangian is satisfied:

$\nabla L(x, \lambda) = 0$

$\nabla f(x) + \lambda \cdot \nabla g(x) = 0 \quad \text{(Stationarity condition)}$

$g(x) \leq 0 \quad \text{(Primal feasibility)}$

$\lambda \geq 0 \quad \text{(Dual feasibility)}$

$\lambda \cdot g(x) = 0 \quad \text{(Complementary slackness)}$

The Lagrange multiplier associated with an inequality constraint must be non-nega
If λ were negative, the second term in the Lagrangian $(\lambda \cdot g(x))$ would become unb
as $g(x)$ is always less than or equal to zero. This would lead to an unbounded increa
making it impossible for the optimization problem to have a finite optimum.

2. **KKT conditions.** Consider again the healthy snack exercise from the previous assignment.
   - Write out the Lagrangian for this problem and the optimality conditions.
   - Verify that the solution you got satisfies the optimality conditions.

In [ ]:
```
from scipy.optimize import linprog
import numpy as np

# Define the coefficients of the objectivefunction
c = np.array([0.50, 0.80])

# Define the coefficients of the constraints
A = np.array([[-3, 0],  # Chocolate
              [-2, -4],  # Sugar
              [-2, -5]]) # Cream Cheese
```

```
b = np.array([-6, -10, -8]) # At least ...

# Bounds
x_bounds = [
    (0, None),
    (0, None)
]

# Solve the linear programming problem
result = linprog(c, A_ub=A, b_ub=b, bounds=x_bounds, method='highs')

# Print the result
print(result)
```

```
       message: Optimization terminated successfully. (HiGHS Status 7: Opti
mal)
       success: True
        status: 0
           fun: 2.2
             x: [ 2.000e+00  1.500e+00]
           nit: 1
         lower:  residual: [ 2.000e+00  1.500e+00]
                marginals: [ 0.000e+00  0.000e+00]
         upper:  residual: [       inf        inf]
                marginals: [ 0.000e+00  0.000e+00]
         eqlin:  residual: []
                marginals: []
       ineqlin:  residual: [ 0.000e+00  0.000e+00  3.500e+00]
                marginals: [-3.333e-02 -2.000e-01 -0.000e+00]
 mip_node_count: 0
 mip_dual_bound: 0.0
       mip_gap: 0.0
```

In [ ]:
```
# Check the multipliers
print(result.ineqlin)
```

```
 residual: [ 0.000e+00  0.000e+00  3.500e+00]
marginals: [-3.333e-02 -2.000e-01 -0.000e+00]
```

In [ ]:
```
# Check the lagrange optimal conditions
x1 = 2.0
x2 = 1.5
nu1 = 1/30
nu2 = 0.2000
nu3 = 0.0


print("Values:")
print("Gradient of the lagrange respect to x1:", 0.50 + nu1*(-3) + nu2*(-2)
print("Gradient of the lagrange respect to x2:", 0.80 + nu2*(-4) + nu3*(-5))
print("Gradient of the lagrange respect to nu1:", -3*x1 + 6)
print("Gradient of the lagrange respect to nu2:", -2*x1 - 4*x2 + 10)
print("Gradient of the lagrange respect to nu3:", -2*x1 - 5*x2 + 8)
```

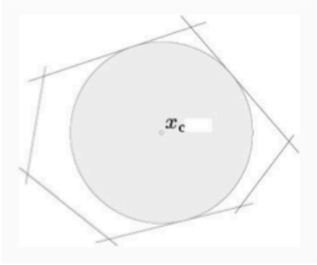
```
Values:
Gradient of the lagrange respect to x1: 0.0
Gradient of the lagrange respect to x2: 0.0
Gradient of the lagrange respect to nu1: 0.0
Gradient of the lagrange respect to nu2: 0.0
Gradient of the lagrange respect to nu3: -3.5
```

The results all show that the solutions satisfy the optimality conditions.

3. Center of Polyhedron. Consider a convex polygon P with m sides defined as $P = \{x \,|\, a_i^T x \leq b_i,\ i = 1 \cdots m\}$. We seek to find the center and radius of the largest inscribable ball in the polygon.

- Formulate the problem as a linear programming problem. Hint. A point x is at a distance R from the line i if $a_i^T x - b_i = -R\|a_i\|$.
- Use the data sample below and solve the problem.
- What is the significance of the Lagrange multipliers in your solution.
- Write out the Lagrangian for this problem as well as the optimality conditions. Verify that the solution satisfies the optimality conditions.

```
A = np.array( [[0, -1], [2, -1], [1, 1], [-1/3, 1], [-1, 0], [-1, -1]] )
b = np.array( [0, 8, 7, 3, 0, -1] )
```



Objective: Maximizing R (to find the largest inscribable ball) Contraint: For each side i of the convex polygon

In [ ]:
```python
from scipy.optimize import linprog
import numpy as np

dimension = 2

A = np.array([[0, -1], [2, -1], [1, 1], [-1/3, 1], [-1, 0], [-1, -1]])
b = np.array([0, 8, 7, 3, 0, -1])

m = A.shape[0]
# print(m)

c = np.zeros(dimension + 1)
c[-1] = -1

constraint_matrix = np.hstack((A, np.linalg.norm(A, axis=1).reshape(-1, 1)))

x_bounds = [(None, None)] * dimension + [(0, None)]

res = linprog(c, A_ub=constraint_matrix, b_ub=b, bounds=x_bounds, method='hi

if res.success:
    center = res.x[:-1]
    radius = res.x[-1]
```

```
    print(f"Center of the largest inscribable ball: {center}")
    print(f"Radius of the largest inscribable ball: {radius}")

    primal_objective_value = res.fun
    dual_variables = res.slack[:m]

    for i in range(m):
        slackness = np.dot(A[i], center) - b[i]
        print(f"Complementary slackness for constraint {i + 1}: {slackness *
else:
    print("Optimization failed.")
```

```
Center of the largest inscribable ball: [2.4961282  1.86556478]
Radius of the largest inscribable ball: 1.8655647845761698
Complementary slackness for constraint 1: -0.0
Complementary slackness for constraint 2: -3.4199841199175114
Complementary slackness for constraint 3: -0.0
Complementary slackness for constraint 4: -0.0
Complementary slackness for constraint 5: -1.5739671093408072
Complementary slackness for constraint 6: -2.4318015048042256
```

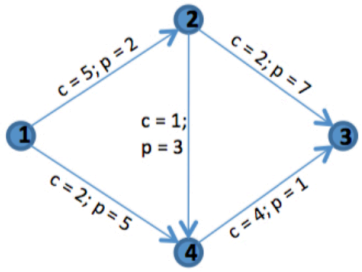
The Lagrange multiplier is significant in this problem, because it helps in finding the optimal values for decision variables while satisfying a set of constraints.

Also the solutions as seen satisfy the optimality conditions.

$$L(x_c, R, \lambda) = -R + \sum_{i=1}^{m} \lambda_i \left(a_i^T x_c - b_i + R\|a_i\| \right)$$

4. Minimum Cost Flow. Many graph problems (shortest-path, multi-commodity flow, minimum-cost flow, etc.) can be formulated as linear programs. In this exercise, we explore the problem of the minimum-cost shipment of an amount of goods from a source node to a destination node in a transportation network, consisting of n nodes and m arcs connecting these nodes. Associated with each arc is a maximum capacity c_i (the maximum amount that can be sent along the arc) and a unit cost p_i (the cost of shipping one unit along the arc). Formulate the problem as an LP which minimizes an appropriate cost function subject to capacity constraints, as well as constraints that the sum of all goods arriving and departing from a node is zero, except for the source and destination (where the sum is q and $-q$ respectively, q being the amount of goods shipped). The variables are the amounts sent along the arcs of the network.

- Solve the problem for the graph below where we wish to send 4 units from node 1 to node 3.
- What is the significance of the multipliers of the capacity constraints? What is the significance of the multipliers of the node constraints?



Problem Formulation

Variables

x_1 : node 1 to node 2
x_2 : node 1 to node 4
x_3 : node 2 to node 3
x_4 : node 4 to node 3
x_5 : node 2 to node 4

Define the vector of variables

$$x = [x_1, x_2, x_3, x_4, x_5]^T$$

Minimize the total cost

$$\text{Minimize } P^T x = \begin{bmatrix} 2 & 5 & 7 & 1 & 3 \end{bmatrix} \begin{bmatrix} x \\ x \\ x \\ x \\ x \end{bmatrix}$$

Subject to the flow conservation constraints

$$x_3 + x_5 - x_1 = 0$$
$$x_4 - x_2 - x_5 = 0$$
$$x_1 + x_2 - q = 0$$
$$q - x_3 - x_4 = 0$$

Subject to the capacity constraints

$$0 \leq x_1 \leq 5$$
$$0 \leq x_2 \leq 2$$
$$0 \leq x_3 \leq 2$$
$$0 \leq x_4 \leq 4$$
$$0 \leq x_5 \leq 1$$

- **Solve the probem for the graph below where we wish to send 4 units from node 1 to node 3**

In [ ]:
```
import numpy as np
import cvxpy as cp

# setup variables
n = 5 # % edges
q = 4 # transport quantity
p = np.array([2,5,7,1,3])
# Capacity
c = np.array([5, 2, 2, 4, 1])
# Connectivity
A = np.array([
```

```
    [-1,0,1,0,1],
    [0,-1,0,1,-1],
    [1,1,0,0,0],
    [0,0,-1,-1,0]
])
b = np.array([0,0,q,-q])

# Each variable is an edge in the graph
x = cp.Variable(n)

# objective function
objective = cp.Minimize(p @ x)

# Define the constraints
# Connectivity constraints and inequality constraints
constraints = [A @ x == b, x <= c]

# Create problem
problem = cp.Problem(objective, constraints)

# Solve the problem
problem.solve()
```

Out[ ]:
```
26.99999999981136
```

In [ ]:
```
print(f'The optimum solution is the vector x_* = {x.value}')
```

```
The optimum solution is the vector x_* = [2. 2. 1. 3. 1.]
```

The problem has optimal solution $x^* = [2, 2, 1, 3, 1]$ minimizing the cost of moving the quantities along the edges of the graph.

- **What is the significance of the multipliers of the capacity constraints?**

In [ ]:
```
# The node constraints have the lagrange multipliers
print(f"lagrange multipliers for the node constraints: {np.array(problem.cor
```

```
lagrange multipliers for the node constraints: [2.47272369e-10 3.00000000e+0
0 4.03201462e-10 1.97013615e-10
 3.00000000e+00]
```

The lagrange multipliers for the capacity constraints gives us the information of how the constraints are affecting the optimal solution and the sensitivity of the change of the optimal value respect to the change in the constraint.

For the capacity constraints we can see that we have a tight bound for $x_2 = 3.00$ and $x_5 = 3.00$ which implies that the constraint is higly active and an increment in the capacity will decrease the value of the cost function. Additionally, the multiplier could give us how much the cost function will decrease if we modify the bound of the capacity. The for the arcs **x_2** and **x_5** we will probably have a disminution of 3.00 per unit transported along the edge.

- **What is the significance of the multipliers of the node constraints?**

In [ ]:
```
# The capacity constraints have the lagrange multipliers
print(f"lagrange multipliers for the capacity constraints: {problem.constrai
```

```
lagrange multipliers for the capacity constraints: [-2.75  3.25 -4.75  4.25]
```

The multipliers of the node constrains give insight of the relation of each node in the graph respect to the cost function. Node's 1 and 3 have a negatie multiplier which represents the supply and demand of the graph. It implies that if I increase the supply and the demand I will have a decresease in the cost function per good transported. Moreover, the nodes 2, and 4 serve as transfer points to meet the demand in the graph. This means that the cost will fe affected if there are changes in the nodes.

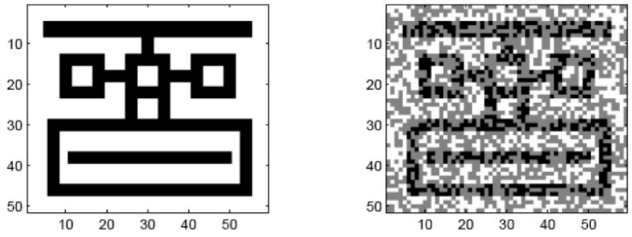
5. Image Completion. A grayscale image is represented as an $M \times N$ matrix of intensities U. Suppose we are given the values of U_{ij}^0 for a subset of the pixels of such an image and seek to complete the image by guessing the missing values. The reconstructed image consists of the complete matrix $U \in R^{M \times N}$ where U satisfies the interpolation conditions $U_{ij} = U_{ij}^0$. The reconstruction is found by minimizing an appropriate measure of roughness of the image satisfying these conditions.

- Write a formulation of the image interpolation problem using an L_2 roughness measure

$$\sum_{i=1}^{M}\sum_{j=1}^{N}(U_{ij} - U_{i-1,j})^2 + (U_{ij} - U_{i,j-1})^2$$

  Hint: The simplest formulation will result in a quadratic objective and linear constraints.

- To test your solution, consider the image below. Use the data from the obscured image on the right to see how well you can recover the original image on the left (data posted on Blackboard).



In [ ]:
```
import numpy as np
import matplotlib
import cvxpy as cp
import matplotlib.pyplot as plt

# Read a sample (``original'') image
U0 = plt.imread('bwicon.png')        # values are between 0.0 (black) and 1.
m, n = U0.shape

# Create 50% mask of known pixels and use it to obscure the original
np.random.seed(211)          # seed the random number generator (for repeata
unknown = np.random.rand(m,n) < 0.5
U1 = U0*(1-unknown) + 0.5*unknown

# Display images
plt.figure(1)
plt.subplot(1, 2, 1)
```

```
plt.imshow(U0, cmap='gray')
plt.title('Original image')
plt.subplot(1, 2, 2)
plt.imshow(U1, cmap='gray')
plt.title('Obscured image')
plt.show()
```

Problem Formulation

$$\text{minimize}_{U_{ij}} \sum_{i=1}^{M} \sum_{j=1}^{N} \left((U_{ij} - U_{i,j-1})^2 + (U_{ij} - U_{i-1,j})^2 \right)$$
$$\text{subject to} \quad U_{ij} - U_{ij} = 0$$

$$\text{Define} \quad X \in \mathbb{R}^{MN}, \quad K = Mj + i$$

$$\text{minimize}_{X} \sum_{k=0}^{MN} \left((X_k - X_{k-1})^2 + (X_k - X_{k-M})^2 \right)$$
$$\text{subject to} \quad X_k - X_k^0 = 0$$

Then:

$$\text{minimize}_{X} \quad X^T C X$$
$$\text{subject to} \quad AX = b$$

In [ ]:
```
set(U1.flatten())
```

Out[ ]:
```
{0.0, 0.5, 1.0}
```

In [ ]:
```
m , n = U0.shape
# Build the matrices
C1 = np.zeros((m*n, m*n))
C2 = np.zeros((m*n, m*n))

# Create the adjacency matrices
for i in range (1, m*n):
  C1[i, i-1] = -1
  C1[i, i] = 1
  C2[i, i-m] = -1
  C2[i, i] = 1

# Constraints
A = np.zeros((m*n, m*n))
b = U1.flatten()

for i in range(m*n):
  if b[i] == 0.5:
    A[i,i] = 0
  else:
    A[i,i] = 1

b = np.zeros([m*n, 1])
_A = np.zeros([m*n, 1])

for j in range(0,n):
    for i in range(0,m):
        k = m * j + i
        if unknown[i][j]==True:
            b[k]=0
        elif unknown[i][j]==False:
            b[k]=U0[i][j]
            _A[k]=1
A=np.diag(np.reshape(_A,[m*n,],order='F'))
```

In [ ]:
```
# Define variable
x = cp.Variable([m*n, 1])
# Define objective
objective = cp.Minimize(cp.norm(C1 @ x, 2) + cp.norm(C2 @ x, 2))
# Constraints
constraints = [A @ x == b]

# Create problem
problem = cp.Problem(objective, constraints)

# Solve the problem
problem.solve()
```

Out[ ]:
```
28.5142816266938
```

In [ ]:
```
image = np.array(x.value)
image = np.reshape(image, (m, n), order='F')
plt.imshow(image, cmap='bone')
```

Out[ ]: <matplotlib.image.AxesImage at 0x7cfc1f983370>