Module 6 Activity

Assigned at the start of Module 6

Due at the end of Module 6

Weekly Discussion Forum Participation

Each week, you are required to participate in the module's discussion forum. The discussion forum consists of the week's Module Activity, which is released at the beginning of the module. You must complete/attempt the activity before you can post about the activity and anything that relates to the topic.

Grading of the Discussion

1. Initial Post:

Create your thread by **Day 5 (Saturday night at midnight, PST).**

2. Responses:

Respond to at least two other posts by **Day 7 (Monday night at midnight, PST).**

Grading Criteria:

Your participation will be graded as follows:

Full Credit (100 points):

- Submit your initial post by **Day 5.**
- Respond to at least two other posts by **Day 7.**

Half Credit (50 points):

- If your initial post is late but you respond to two other posts.
- If your initial post is on time but you fail to respond to at least two other posts.

No Credit (0 points):

- If both your initial post and responses are late.
- If you fail to submit an initial post and do not respond to any others.

Additional Notes:

- **Late Initial Posts:** Late posts will automatically receive half credit if two responses are completed on time.
- **Substance Matters:** Responses must be thoughtful and constructive. Comments like "Great post!" or "I agree!" without further explanation will not earn credit.
- **Balance Participation:** Aim to engage with threads that have fewer or no responses to ensure a balanced discussion.

Avoid:

- A number of posts within a very short time-frame, especially immediately prior to the posting deadline.
- Posts that complement another post, and then consist of a summary of that.

Problem 1: Linear Programming

Objective:

Solve a linear programming (LP) problem using Python, analyze the results, and interpret the solution in a real-world context.

Problem Statement:

A factory produces two products: Product (A) and Product (B).

- The profit from each unit of Product (A) is $3, and from each unit of Product (B) is $5.
- The factory operates under the following constraints:

1. **Time Constraint:**
   - Producing one unit of Product (A) requires 1 hour, and producing one unit of Product (B) requires 2 hours.
   - The total time available for production is 8 hours.
2. **Resource Constraint:**
   - Producing one unit of Product (A) uses 3 units of a raw material, and producing one unit of Product (B) uses 2 units of the same material.
   - The factory has 12 units of raw material available.
3. **Non-Negativity Constraint:**
   - The quantities of Product (A) and Product (B) cannot be negative.

The goal is to maximize the profit, what combination of Product (A) and Product (B) accomplishes this?

Example Code:

```python from scipy.optimize import linprog

## Coefficients of the objective function (negative for maximization)

c = [-1, -1]

## Coefficients for the constraints

A = [[1, 1], [1, 1]] b = [1, 1]

## Bounds for variables

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

## Solve the linear programming problem

result = linprog(c, A_ub=A, b_ub=b, bounds=x_bounds, method='simplex')

## Output results

print("Optimal Value:", -result.fun) # Flip the sign for maximization print("Optimal Solution:", result.x)

```python
from scipy.optimize import linprog

# --- The Craft Fair Problem ---
# x = number of product A
# y = number of product B
# Profit from x = $3
# Profit from y = $5

# Coefficients of the objective function (negative for maximization)
# Maximize: Z = 3x + 5y
c = [-3, -5]
```
```

```
# Constraint coefficients (Ax <= b)
# Time constraint:    x + 2y <= 8
# Resource constraint:  3x + 2y <= 12
#   x >= 0
#   y >= 0

A = [[1, 2], [3, 2]]
b = [8, 12]

# Bounds for each variable (x, y >= 0)
x_bounds = [(0, None), (0, None)]

# Solve the LP using the HiGHS solver (simplex is deprecated)
result = linprog(c, A_ub=A, b_ub=b, bounds=x_bounds, method='highs')

# Output results
if result.success:
    print(f"Product A (x): {result.x[0]:.2f}")
    print(f"Product B (y): {result.x[1]:.2f}")
    print(f"Maximum Profit: ${-result.fun:.2f}") # Flip the sign for
maximization
else:
    print("Optimization failed.")
    print("Message:", result.message)
```

```
Product A (x): 2.00
Product B (y): 3.00
Maximum Profit: $21.00
```

Questions:

Q1. Update the code to reflect the problem statement and solve the linear programming problem. What changes did you make to the code?

Ans: Above is updated code to solve it using linear programming. I have made following changes

1. Defined the Linear Programming Mathematical Formulation as below **Decision Variables:**

- `x` = number of product A
- `y` = number of product B
- Profit from `x = $3`
- Profit from `y = $5`

Objective Function:

```
Maximize: Z = 3x + 5y
```

Constraints:

```
x + 2y <= 8 # Time constraint:
3x + 2y <= 12 # Resource constraint:
x >= 0
y >= 0
```

1. Updated `A`, `b` and `c` as per above mathematical formulation
2. Switch to `highs` method as was getting deprecation warning for `simplex`
3. Updated print format for each product A and B

Q2. What do the optimal values for Product (A) and Product (B) represent?

Ans. Following is optimal values

```
Product A (x): 2.00
Product B (y): 3.00
Maximum Profit: $21.00
```

Q3. How would the solution change if the profit for Product (B) increased to $6? **Ans.** It changed as below.

```
Product A (x): 0.00
Product B (y): 4.00
Maximum Profit: $24.00
```

Q4. How does the feasible region influence the optimal solution?

Ans. The feasible region is the set of all points that satisfy all constraints of the LP problem — both inequality and equality constraints. Following are ways feasible region influence optimal solution

1. The optimal solution lies on the boundary. We evaluate objective function at each point to get the optimal solution.
2. If no feasible region then there is no optimal solution.
3. For unbounded feasible region it may have no maximum therefore no finite optimal solution
4. If the objective function is parallel to an edge of the feasible region, you may get infinitely many optimal solutions.

Problem 2: Quadratic Programming

Objective:

Solve a quadratic programming (QP) problem by coding the objective function and constraints into CVXOPT and analyzing the solution.

Problem Statement:

A company wants to optimize the allocation of resources between two projects to minimize costs. The objective function and constraints are as follows:

- **Objective Function**: Minimize $2x_1^2 - x_1x_2 + 4x_2^2 - 3x_1 - 2x_2$.
- **Constraints**:
  - The total allocation of resources is limited: $x_1 + x_2 = 1$.
  - No resources can be allocated as negative values: $x_1, x_2 \geq 0$.

Task: Write the objective function and constraints in CVXOPT format, run the code, and find the optimal solution.

Example Code:

Use the following code template to solve the problem:

```` ```python from cvxopt import matrix, solvers ````

Define Q (quadratic term) and c (linear term)

Q = matrix([[1.0, -1.0], [-1.0, 1.0]]) # Quadratic coefficients c = matrix([-1.0, -1.0]) # Linear coefficients

Inequality constraints Gx <= h

G = matrix([[-1.0, 0.0], [0.0, -1.0]]) # Negative identity for x >= 0 h = matrix([0.0, 0.0]) # Bounds for x

Equality constraints Ax = b

A = matrix([1.0, 1.0], (1, 2)) # Total allocation constraint b = matrix([1.0]) # Total value

Solve the QP problem

sol = solvers.qp(Q, c, G, h, A, b)
````

Print the optimal solution

print("Optimal Solution:", sol['x']) print("Optimal Value:", sol['primal objective'])

Quadratic programming Objective function

Let's rewrite into standard form

Objective to minimize: \begin{align} f(x) = 2x_1^2 - x_1x_2 + 4x_2^2 - 3x_1 - 2x_2 \end{align} \begin{align}
f(x) =\frac{1}{2}\begin{bmatrix} x_1 \ x_2 \end{bmatrix}^T \begin{bmatrix} 4 & -2 \ 0 & 8 \end{bmatrix}\begin{bmatrix} x_1 \ x_2 \end{bmatrix}+ \begin{bmatrix} -3 \ -2 \end{bmatrix}^T \begin{bmatrix} x_1 \ x_2 \end{bmatrix} \end{align}

Where \begin{align} x = \begin{bmatrix} x_1 \ x_2 \end{bmatrix} \end{align}

\begin{align} Q = \begin{bmatrix} 4 & -2 \ 0 & 8 \end{bmatrix} \end{align} \begin{align} c = \begin{bmatrix} -3 \ -2 \end{bmatrix} \end{align}

Inequality Constraints: \begin{align} \begin{bmatrix} -1 & 0 \ 0 & -1 \end{bmatrix}
\begin{bmatrix} x_1 \ x_2 \end{bmatrix} \le
\begin{bmatrix} 0 \ 0 \end{bmatrix} \end{align}

Where

\begin{align} G = \begin{bmatrix} -1 & 0 \ 0 & -1 \end{bmatrix}
\end{align}

\begin{align} h = \begin{bmatrix} 0 \ 0 \end{bmatrix}
\end{align}

Equality Constraints: \begin{align} \begin{bmatrix} 1 & 1 \end{bmatrix}
\begin{bmatrix} x_1 \ x_2 \end{bmatrix} =
\begin{bmatrix} 1 \end{bmatrix} \end{align}

Where

\begin{align} A = \begin{bmatrix} 1 & 1 \end{bmatrix}
\end{align}

\begin{align} b = \begin{bmatrix} 1 \end{bmatrix}
\end{align}

Goal

Find the specific values for x_1 and x_2 that make the function $f(x)=2x_1^2 - x_1x_2 + 4x_2^2 - 3x_1 - 2x_2$ as small as possible, while still obeying a set of given rules (constraints).

Following is code to implement the Quardratic programming using

```
from cvxopt import matrix, solvers
```

```
# Define Q (quadratic term) and c (linear term)
Q = matrix([[4.0, -2.0],   # column 0
            [0.0, 8.0]],   # column 1
            (2, 2))        # shape: 2x2
c = matrix([-3.0, -2.0])    # Linear term

# Inequality constraints Gx <= h  (x1 ≥ 0, x2 ≥ 0 → -x1 ≤ 0, -x2 ≤ 0)
G = matrix([[-1.0, 0.0],
            [ 0.0, -1.0]])
h = matrix([0.0, 0.0])

# Equality constraint: x1 + x2 = 1
A = matrix([1.0, 1.0], (1, 2))  # 1 row, 2 columns
b = matrix([1.0])

# Solve the QP problem
sol = solvers.qp(Q, c, G=G, h=h, A=A, b=b)

# Print the optimal solution
print("Optimal Solution:")
for i, val in enumerate(sol['x']):
    # Format each value to 5 decimal places
    print(f"  x[{i}]: {val:.5f}")
print(f"\nOptimal Value: {sol['primal objective']:.5f}")

     pcost       dcost       gap    pres   dres
 0: -1.7778e+00 -2.8889e+00  1e+00  1e-16  7e-01
 1: -1.7812e+00 -1.8001e+00  2e-02  1e-16  1e-02
 2: -1.7812e+00 -1.7814e+00  2e-04  2e-16  1e-04
 3: -1.7812e+00 -1.7813e+00  2e-06  2e-16  1e-06
 4: -1.7812e+00 -1.7813e+00  2e-08  1e-16  1e-08
Optimal solution found.
Optimal Solution:
  x[0]: 0.68750
  x[1]: 0.31250

Optimal Value: -1.78125
```

Questions:

Q1. What are the optimal values for x_1 and x_2, and what do they represent in the context of the problem?

Ans: As per above code execution, the optimal values for x_1 is `0.68750` and x_2 is `0.31250`. This gives the location of the minimum which is `-1.78125`.

Q2. How does the constraint $x_1 + x_2 = 1$ impact the solution space?

Ans: Without this constraints x_1 and x_2 could be any real numbers, it means the solution space is the entire infinite 2D plane. The solver would have to search everywhere.

With $x_1 + x_2 = 1$ constraints, the solution space has now collapsed from an infinite 2D plane down to just the infinite 1D line defined by this equation. The solver is no longer allowed to look anywhere else.

Q3. If the constraint $x_1 + x_2 = 1$ were replaced with $x_1 + x_2 \leq 1$, how would the solution change?

Ans. $x_1 + x_2 = 1$ forced all possible solutions to lie on a 1D line segment. On the other hand, $x_1 + x_2 \leq 1$ defines a 2D triangular region. The line $x_1 + x_2 = 1$ now acts as the boundary. The "less than" part (<) allows the solution to exist anywhere inside this boundary as well.

Problem 3: Analyzing the Simplex Method

Objective:

Analyze the provided Simplex algorithm implementation by stepping through the code, interpreting its operations, and understanding its runtime complexity.

Problem Statement:

The Simplex algorithm is used to solve the following linear programming problem:

- **Objective Function:** Maximize $3x_1 + 5x_2$
- **Constraints:**
  - $x_1 + 2x_2 \leq 8$
  - $3x_1 + 2x_2 \leq 12$
  - $x_1, x_2 \geq 0$

Task: Run the provided Simplex method implementation to solve the problem. Then:

1. Comment on the function's operations **line by line**, analyzing the **runtime complexity** of each line or block.
2. Calculate the **runtime complexity** of the Simplex algorithm in terms of the number of constraints (m) and variables (n).

Code:

Below is the Simplex method implementation. Run the code and analyze its steps:

```python
import numpy as np
import pandas as pd

def simplex_algorithm(c, A, b):
    """
    Solve the Linear Programming problem using the Simplex Method.

    Maximize: c^T * x
    Subject to: A * x <= b, x >= 0

    Parameters:
    - c: Coefficients of the objective function (1D numpy array).
    - A: Constraint coefficients (2D numpy array).
    - b: Constraint bounds (1D numpy array).

    Returns:
    - Optimal value and solution.
    """
    m, n = A.shape # m = number of constrains, n = number of varaibles

    # Add slack variables to convert inequalities to equalities
    slack_vars = np.eye(m)
    tableau = np.hstack([A, slack_vars, b.reshape(-1, 1)])

    # Append the objective row
    obj_row = np.hstack([-c, np.zeros(m + 1)])
    tableau = np.vstack([tableau, obj_row])

    # Variable tracking
    basic_vars = [n + i for i in range(m)]
    non_basic_vars = list(range(n))

    step = 0

    while True:
        print(f"Step {step}: Tableau")
        # df is of size (m+1)*(n+m+1)
        df = pd.DataFrame(
            tableau,
            columns=[f"x{i + 1}" for i in range(n + m)] + ["RHS"],
            index=[f"Constraint {i + 1}" for i in range(m)] +
["Objective"]
        )
        print(df)
        print("\n")

        # Check if the current solution is optimal (no negative
coefficients in the objective row)
        if np.all(tableau[-1, :-1] >= 0): # Run time: scan all columns
of objective row i.e. O(m+n)
            print("Optimal solution found!\n")
            break

        # Choose entering variable (most negative coefficient in the
objective row)
```

```
        entering = np.argmin(tableau[-1, :-1]) # Run time: Scan all 
columns i.e. O(m+n)

        # Choose leaving variable (minimum positive ratio of RHS to 
pivot column)
        ratios = []
        for i in range(m): # Run time: m times
            if tableau[i, entering] > 0:
                ratios.append(tableau[i, -1] / tableau[i, entering])
            else:
                ratios.append(np.inf)
        leaving = np.argmin(ratios)

        if ratios[leaving] == np.inf:
            raise ValueError("Problem is unbounded!")

        # Pivot operation
        pivot = tableau[leaving, entering]
        tableau[leaving, :] /= pivot # Run time O(m+n)

        for i in range(m + 1): # Run time: m+1 times
            if i != leaving:
                tableau[i, :] -= tableau[i, entering] * 
tableau[leaving, :] # Run time: This is a vector operation on two 
rows, each of length n+m+1

        # Update basic and non-basic variables
        basic_vars[leaving] = entering

        step += 1

    # Extract solution
    solution = np.zeros(n + m)
    for i, var in enumerate(basic_vars): # Run time : n times
        if var < n:
            solution[var] = tableau[i, -1]

    optimal_value = tableau[-1, -1]

    print("Optimal Value:", optimal_value)
    print("Solution:", solution[:n])

    return optimal_value, solution[:n]


c = np.array([3, 5])  # Coefficients of the objective function
A = np.array([[1, 2], [3, 2]])  # Coefficients of the constraints
b = np.array([8, 12])  # RHS of the constraints

simplex_algorithm(c, A, b)
```

```
Step 0: Tableau
               x1   x2   x3   x4   RHS
Constraint 1  1.0  2.0  1.0  0.0   8.0
Constraint 2  3.0  2.0  0.0  1.0  12.0
Objective    -3.0 -5.0  0.0  0.0   0.0


Step 1: Tableau
               x1   x2   x3   x4   RHS
Constraint 1  0.5  1.0  0.5  0.0   4.0
Constraint 2  2.0  0.0 -1.0  1.0   4.0
Objective    -0.5  0.0  2.5  0.0  20.0


Step 2: Tableau
               x1   x2    x3    x4   RHS
Constraint 1  0.0  1.0  0.75 -0.25   3.0
Constraint 2  1.0  0.0 -0.50  0.50   2.0
Objective     0.0  0.0  2.25  0.25  21.0


Optimal solution found!

Optimal Value: 21.0
Solution: [2. 3.]

(np.float64(21.0), array([2., 3.]))
```

Simplex Algorithm Time complexity

Total Complexity = (Number of Iterations) × (Cost per Iteration)

- m = number of constraints (rows in A)
- n = number of original variables (columns in A)

Part 1: Cost Per Iteration (Cost of a single pivot)

- The size of the tableau is (m+1) x (n+m+1).
- Check for Optimality: `np.all(tableau[-1, :-1] >= 0)` - `O(m+n)`
- Choose Entering Variable: `entering = np.argmin(tableau[-1, :-1])` - `O(m+n)`
- Choose Leaving Variable: `O(m)`
- Pivot Operation (The Most Expensive Step): `O(m * (n+m))`

I.e. time complexity of each iteration is `O(m * (n+m))`

Part 2: Number of Iterations (Number of pivots)

Each iteration of the Simplex algorithm moves from one vertex of the feasible polytope to an adjacent vertex with a better or equal objective value.

- The number of iterations is the number of vertices visited.
- The total number of vertices can be at most `C(n+m, n)` (the number of ways to choose n basic variables from n+m total variables).

Overall time complexity

```
C(n+m, n) * (m * (n+m))
```

I.e. it is exponential.