Numerical Optimization - Assignment 9

David Alvear(187594) - Nouf Farhoud(189731)

In [ ]:
```
import numpy as np
import scipy.optimize
import cvxpy as cp
import matplotlib.cm as cm
import matplotlib.pyplot as plt
from scipy.optimize import linprog
import scipy.optimize
```

1. Portfolio Optimization

Classic Portfolio Optimization

Problem Formulation

$$\text{minimize } V = \sum_i \sum_j X_i X_j \sigma_i \sigma_j \rho_{ij}$$

$$\text{subject to}$$

$$\sum_i X_i \mu_i \geq r_{\min}$$

$$\sum_i X_i = 1$$

$$X_i \geq 0 \quad i = 1 \dots n$$

In [ ]:
```
# Problem variables
rho = np.array([[1,0.3,0.4,0],[0.3,1,0,0],[0.4,0,1,0],[0,0,0,1]]) # This is
std = np.diag([0.2,0.1,0.05,0])
mean_return = np.array([0.12,0.1,0.07,0.03])
# sigma = std @ rho @ std.T
sigma = std @ rho @ std.T

# Define Variables and parameters
x = cp.Variable(4)
r_min = cp.Parameter(nonneg=True)

# Define objective
objective = cp.Minimize(cp.quad_form(x, sigma))

# Constraints
constraints = [
```

```
    cp.sum(x) == 1,
    x >= 0,
    x @ mean_return >= r_min
]

# Define problem and solve
problem = cp.Problem(objective, constraints)
r_min.value = 0.1
problem.solve()

# Solution
# get the asset allocations
print("Considering a minimum acceptance return of R=10%, the solution is the
print("The minimum risk is: ",problem.value)
print("The % of each of the 4 assets is divided as follows: ", (x.value))
```

```
Considering a minimum acceptance return of R=10%, the solution is the follow
ing:
The minimum risk is:  0.007901639344262298
The % of each of the 4 assets is divided as follows:  [1.96721311e-01 6.7213
1148e-01 1.31147541e-01 5.04380808e-22]
```

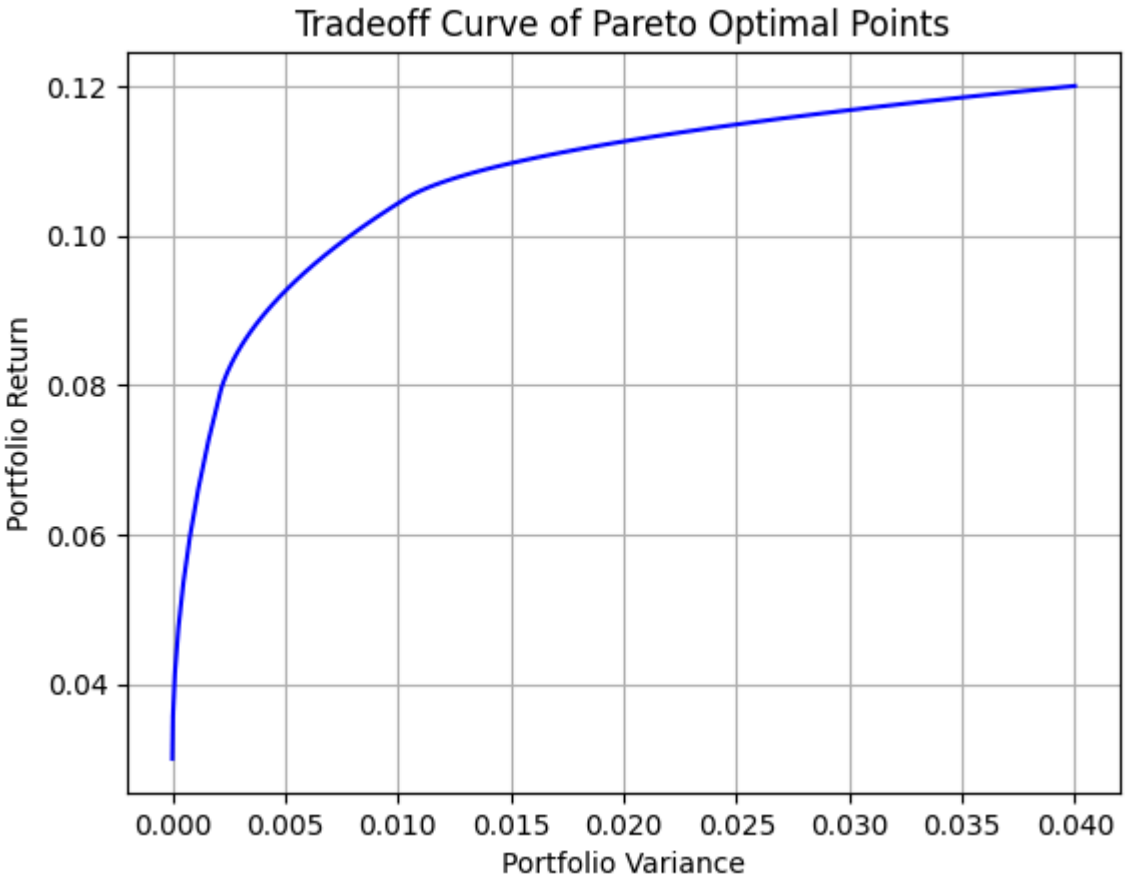
Bi-Objective Portfolio Optimization

Problem Formulation

$$\begin{aligned}
&\text{minimize } KV - (1-K)R \\
&\text{subject to} \\
&\sum_i X_i = 1 \\
&X_i \geq 0 \quad i = 1 \ldots n
\end{aligned}$$

In [ ]:
```
# Problem variables
rho = np.array([[1,0.3,0.4,0],[0.3,1,0,0],[0.4,0,1,0],[0,0,0,1]]) # This is
std = np.diag([0.2,0.1,0.05,0])
mean_return = np.array([0.12,0.1,0.07,0.03])
# sigma = std @ rho @ std.T
sigma = std @ rho @ std.T

# Define Variables and parameters
x = cp.Variable(4)
risk_tol = cp.Parameter(nonneg=True)

# Define objective
objective = cp.Minimize(risk_tol*cp.quad_form(x, sigma) - (1 - risk_tol) * x

# Constraints
constraints = [
    cp.sum(x) == 1,
    x >= 0,
]
```

```python
# Define problem and solve
problem = cp.Problem(objective, constraints)

# Loop to solve the problem for different values of risk tolerance
tradeoff_points = []
for val in np.linspace(0.01, 1, 100):
    risk_tol.value = val
    problem.solve()
    risk =  (x.value @ sigma @ x.value)
    returns =  (x.value @ mean_return)
    tradeoff_points.append((x.value, problem.value, val, risk, returns))

# Extracting the results
variances = [point[3] for point in tradeoff_points]
returns = [point[4] for point in tradeoff_points]

# Plot the Pareto curve
plt.figure('Tradeoff Curve')
plt.plot(variances, returns, 'b-')
plt.xlabel('Portfolio Variance')
plt.ylabel('Portfolio Return')
plt.title('Tradeoff Curve of Pareto Optimal Points')
plt.grid(True)
plt.show()
```



We can see in the curve the behaviour of the bi-objective optimization problem. If we want higher returns we expect to have a higher risk. This tradeoff between returns and

variance can be used to decide the strategy used for the investment. A conservative approach would lie in the region of small return values with a small risk. On the contrary, an aggresive investmen will expect a higher return value with a higher risk.

Short Position Optimization Problem

Problem Definition

$$\text{Define } X = X_{\text{long}} - X_{\text{short}}$$

$$\text{minimize } V = X^T \Sigma X$$

$$\begin{aligned}
&\text{subject to} \\
&\sum_i \mu_i X_i \geq r_{\min} \\
&\sum_i X_i = \sum_i (X_{\text{long}_i} - X_{\text{short}_i}) = 1 \\
&\sum_i X_{\text{short}_i} \leq P \sum_i X_{\text{long}_i} \\
&X_{\text{short}_i} \geq 0 \quad i = 1 \dots n \\
&X_{\text{long}_i} \geq 0 \quad i = 1 \dots n
\end{aligned}$$

2. Central Path

2. Central path. The goal of this exercise is to show that the barrier method generates a sequence of iterates along the central path. Consider the convex problem:

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & Ax = b \\ & f_i(x) \leq 0, \quad i = 1, \cdots, m \end{array} \tag{1}$$

- Write the KKT conditions for its log barrier problem

$$\begin{array}{ll} \text{minimize} & f_0(x) - \tau \sum_{i=1}^{m} \log(-f_i(x)) \\ \text{subject to} & Ax = b \end{array} \tag{2}$$

- Show that the KKT conditions for a point on the central path of (1) are equivalent to problem (2).

Convex Problem:

$$\text{minimize } f_0(x)$$
$$\text{s.t } \begin{cases} Ax = b \\ f_i(x) \leq 0 \quad i = 1, \dots, m \end{cases}$$
$$L(x, \nu, \lambda) = f_0(x) + \nu^T(Ax - b) + \sum_{i=1}^{m} \lambda_i f_i(x)$$

Now, Recall the Path following method to solve the convex problem
Approximate KKT Conditions

$$\nabla L(x, \nu, \lambda) = \nabla f_0(x) + \nu^T A + \sum_{i=1}^{m} \lambda_i \nabla f_i(x)$$
$$Ax = b$$
$$-\lambda_i f_i(x) = \tau : i = 1, \dots, m$$
$$f(x) \leq 0 \quad \lambda \geq 0$$

Log-Barrier Problem:

$$\text{minimize } f_0(x) - \tau \sum_{i=1}^{m} \log(-f_i(x))$$
$$\text{s.t } Ax = b$$
$$L(x, \nu) = f_0(x) - \tau \sum_{i=1}^{m} \log(-f_i(x)) + \nu^T(Ax - b)$$

KKT Conditions for the log-barrier Problem

$$\nabla L(x, \nu) = \nabla f_0(x) - \tau \sum_{i=1}^{m} \frac{1}{f_i(x)} \nabla f_i(x) + \nu^T A$$
$$\nabla L(x, \nu) = \nabla f_0(x) - \tau \operatorname{diag}\left(\frac{1}{f(x)}\right) D(f(x)) + \nu^T A = 0$$
$$Ax = b$$

If the KKT Conditions of a point in the Central Path are equivalent to the KKT conditions in the log barrier:

$$\nabla f_0(x) + \nu^T A + \sum_{i=1}^{m} \lambda_i f_i(x) = 0 = \nabla f_0(x) + \nu^T A - \tau \sum_{i=1}^{m} \frac{1}{f_i(x)} \nabla f_i(x)$$

$$\sum_{i=1}^{m} \lambda_i f_i(x) = -\tau \sum_{i=1}^{m} \frac{1}{f_i(x)} \nabla f_i(x)$$

From this part we can see that for $i = 1$ to m :

$$-\lambda_i f_i(x) = \tau \frac{1}{f_i(x)} \nabla f_i(x)$$

The Central Path KKT Condition $-\lambda_i f_i(x) = \tau$ for $i = 1$ to m, define a series of iterations to find the solution of the original problem reducing τ.
The last expression relates the equivalence with the barrier method and we can see that for a different value of τ we are solving the problem for a point x in the Central Path.

Additionally, we can see that selecting $\lambda_i = \dfrac{\tau}{f_i(x)}$ in the ∇L of Log-barrier Problem is equivalent to the Central Path Primal Problem

3. Interior Point Method for LP

3. Interior Point Method for LP. An inequality constrained LP

$$\begin{array}{ll} \text{minimize} & p^T x \\ \text{subject to} & Ax = b \\ & x \geq 0 \end{array} \tag{3}$$

may be solved by following a "central path" in the interior of the feasible region. The central path is defined as the set of points where the complementarity conditions are equally violated (all $\lambda_i x_i = \tau$).

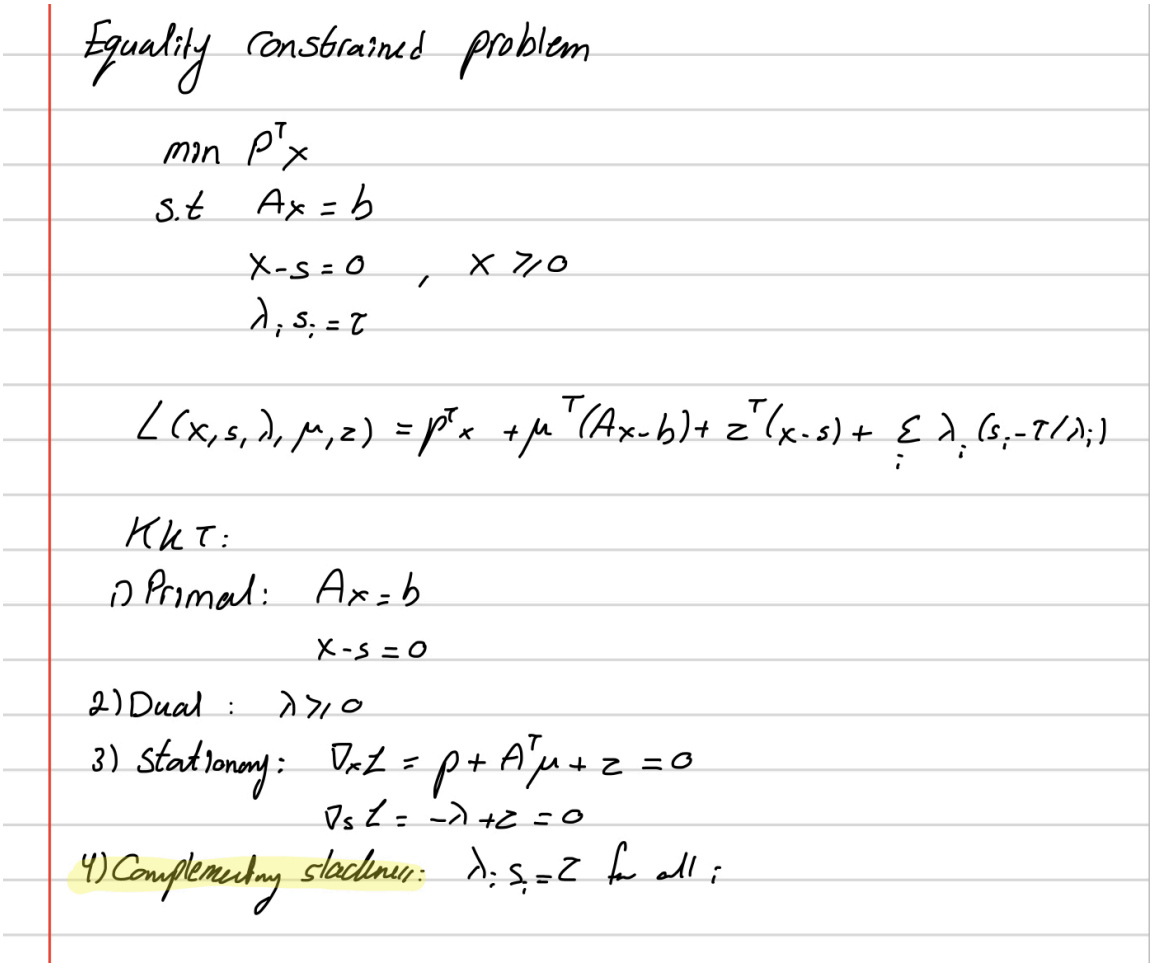
- Checking whether or not the problem has a feasible solution may be done by solving the equality-constrained problem of the previous assignment. Explain how. When that solution exists, it may be used as the initial point for solving (3).
- Every step along the central path may be computed by solving an equality constrained problem. Write the form of the equality constrained problem and show that its KKT conditions are the same conditions defining the central path.
- The path-following method may then be viewed as a sequence of equality-constrained problems, each with a decreasing value of the parameter τ ($\tau^{l+1} = \tau^l/\mu$). Implement such a strategy:

  ```
  def interiorLP(p, A, b, x0, tol):
  ```

  to solve an LP to a tolerance `tol` on the duality gap. The function should return the values of the primal and dual variables at optimality, and a history of the iterates.
- Test your implementation on the given data (with $A_{100\times150}$). Starting from the same initial feasible point, how many total (inner) Newton iterations are needed to decrease the duality gap to 10^{-6} with $\mu = 2$ and with $\mu = 10$. Comment.

The initial phase of applying an interior point method involves determining if the problem has a feasible solution. This can be accomplished through the following steps:

- **Solve the Equality-Constrained Subproblem:** Ax=b This subproblem determines if there exists any x that satisfies the equality constraints.
- **Feasibility Check:** If this subproblem has a solution, then the problem Ax=b,x≥0 has a feasible solution. If the subproblem has no solution, then the original LP problem is infeasible.
- **Using the solution as an initial Point:** if the equality constrained subproblem has a solution, where the solution is >0, then it can be used as an initial point for the interior point method.



The complementary slackness directly corresponds to the definition of the central path. where each pair of primal and dual variables associated with the non-negativity constraint satisfies the complementarity slackness condition uniformly with parameter τ.

The equality x−s=0 ensures that x remains non-negative as s must be non-negative.

We solve the equality constrained problem at each step and this continues until τ approaches zero, ideally converging to the optimal solution of the original LP problem.

In [ ]:

```python
import numpy as np
import scipy.optimize

def interiorLP(p, A, b, x0, tol):
    n = len(x0)  # Number of variables
    m = A.shape[0]  # Number of equality constraints
    tau = 1.0
    mu = 10  # Reduction factor for tau
    max_iter = 100
    history = []

    # Initialize x, s (slack variables), and lambda_ (dual variables for equ
    x = np.maximum(x0, 1e-2)  # Ensure x is strictly positive
    s = np.ones(n)            # Slack variables initially set equal
    lambda_ = np.zeros(m)     # Dual variables for equality constraints

    for iteration in range(max_iter):
        # Form the system of KKT conditions
        X = np.diag(x)
        S = np.diag(s)

        # Calculate the residuals
        r_pri = A @ x - b
        r_dual = p - A.T @ lambda_ - s
        r_cent = x * s - tau * np.ones(n)

        # Construct KKT Matrix
        M = np.block([
            [np.zeros((n, n)), A.T, np.eye(n)],
            [A, np.zeros((m, m)), np.zeros((m, n))],
            [S, np.zeros((n, m)), X]
        ])

        # Solve for directions
        r = np.hstack([-r_dual, -r_pri, -r_cent])
        delta = np.linalg.solve(M, r)

        dx = delta[:n]
        dlambda = delta[n:n+m]
        ds = delta[n+m:]

        # Line search for maximum step size that maintains non-negativity
        alpha_x = min(1, *[-xi/dxi for xi, dxi in zip(x, dx) if dxi < 0])
        alpha_s = min(1, *[-si/dsi for si, dsi in zip(s, ds) if dsi < 0])
        alpha = min(alpha_x, alpha_s, 0.99)

        # Update x, lambda, and s
        x += alpha * dx
        lambda_ += alpha * dlambda
        s += alpha * ds
        tau /= mu

        # Store history
        history.append((x.copy(), lambda_.copy(), s.copy()))
```

```
        # Check convergence (duality gap)
        gap = np.dot(x, s)
        if gap < tol:
            break

    return x, lambda_, history

# Example usage
seed = 9727
rng = np.random.default_rng(seed)
n = 150
p = 100
A = np.hstack((rng.standard_normal((p, n-p)), np.eye(p)))
b = A @ rng.random(n)
pobj = np.concatenate((rng.standard_normal(n-p), np.zeros(p)))

# Initial feasible point (make sure it is feasible)
x0 = np.maximum(0, rng.random(n))

# Solve using the custom interior point method
x_opt, lambda_opt, history = interiorLP(pobj, A, b, x0, tol=1e-6)

# Print solution and history
print("Optimal x:", x_opt)
print("Optimal lambda:", lambda_opt)
# print("History of iterates:", history)
```

```
Optimal x: [8.32873829e-02 3.74313092e-01 1.02973301e-01 5.76638683e-01
 1.17725180e-01 2.23832976e-01 2.44409805e-01 8.20719626e-01
 2.32959510e-24 3.24322965e-02 3.33133024e-01 6.97884065e-02
 8.14350962e-01 6.28891325e-01 8.29909789e-01 5.86743524e-01
 3.11067845e-01 8.21763774e-01 8.67721030e-01 1.56486935e-01
 5.16715162e-01 8.33073167e-01 1.54574769e-01 9.52905013e-01
 2.34610167e-01 5.89529457e-01 3.86376940e-01 6.74609325e-01
 3.62150445e-01 6.22041377e-01 4.86841105e-01 1.91357417e-01
 9.54728767e-02 3.53238119e-02 9.29884440e-02 1.25639129e-01
 8.79461790e-01 9.70670086e-01 2.12750704e-01 3.52773544e-02
 4.42495401e-01 7.11108895e-01 8.28228654e-01 1.99587282e-01
 8.19254139e-01 8.63796189e-01 8.64247104e-02 9.49130909e-01
 1.83231591e-01 7.99889084e-01 1.21385483e+00 2.21735598e-21
 2.26862317e-01 3.57666638e-01 7.56432586e-22 4.41678573e-20
 7.20763969e-20 2.82811129e-02 4.47937777e-19 1.11391518e-01
 7.66431750e-01 9.77683771e-22 5.17932708e-01 7.85810913e-23
 2.32169319e-20 6.89303496e-01 2.81557524e-21 3.00921562e-01
 1.20977447e+00 1.70678043e-18 3.51991564e-24 1.85270672e-24
 1.69006343e+00 5.27506388e-01 4.17378978e-01 3.12885107e-01
 1.24872100e-21 3.22801517e-22 4.46788154e-01 9.25978260e-01
 6.39000179e-25 1.51166092e-22 5.09724764e-01 7.83438833e-01
 8.31512205e-01 4.04829837e-01 4.50428547e-24 6.40250479e-17
 4.59466793e-01 1.48932224e-18 3.38960222e-01 7.49462179e-01
 8.86947430e-01 7.15763644e-25 6.52829608e-21 2.48518430e-24
 4.01551464e-20 5.91561619e-19 8.68298856e-02 2.09558227e-20
 1.01414389e+00 6.79285620e-19 7.94601201e-01 2.26283770e-01
 4.42084307e-23 9.54379487e-24 3.50412864e-01 2.97203871e-18
 4.51054306e-02 3.73154965e-19 7.15695028e-01 8.43510681e-01
 6.49975115e-01 2.45619266e-02 3.45568677e-01 9.14985392e-02
 2.37243534e-24 2.85396940e-01 4.04117778e-01 8.10472673e-01
 5.17005912e-01 5.68633913e-01 1.17753265e-24 5.81137986e-21
 1.92066303e-06 1.18072743e+00 6.79956765e-02 5.13292853e-01
 5.09472495e-18 7.85725780e-25 2.19506380e+00 3.44818782e-19
 3.00575215e-01 1.11960889e-24 1.78820865e-20 9.41440324e-20
 3.87181476e-01 1.94005344e-23 3.76772267e-01 3.78932966e-16
 9.96931416e-20 2.05346202e-24 6.32765291e-01 3.73225542e-20
 2.69456744e-22 1.75623549e-18 3.20795662e-26 5.85872947e-01
 5.80089975e-02 3.28080245e-18]
Optimal lambda: [  680.2763612   -120.57789904   680.27638108   680.27616027
 -1539.0406558   -416.35130616  -190.36072015   680.27528568
   379.70519027   680.27632661   680.27629378 -1316.62095484
   680.27629269    59.12283116  -409.22886921   680.27628993
  -677.65588186   680.27610927   680.27631789   435.30177414
   501.26465951   343.12881589   680.2763643    680.27622696
   680.27625142   680.27614119   247.26055575  -666.41839561
   680.276274     680.27632134  -341.3556252    368.18565717
   680.27634041   680.27628719   680.27632388   680.27623003
   539.6129237    389.14328458   680.27622884    93.64499447
   680.27623714   680.27631418   680.27633351 -1083.77686182
  -718.70423455   424.80875564   431.5201494    158.34052565
   680.27591362  -128.72936789   680.27635034   219.24594987
   680.27635934   680.2762855  -1781.15263309  -204.67813911
   680.27620179   291.7804457    680.27461677   337.97647758
   680.27632078   680.27632376   680.27636858   680.27156171
   680.27631446   680.27638087   412.02930682   680.27619669
   680.27633988   680.27627438   680.27633692   680.27637103
```

```
 140.16549718  -240.64414499   677.85627655   680.27634358
 680.27516263   680.27628954   466.23609694  -443.61092756
 680.27636407    62.55117458   680.27626091   109.62648413
-322.66495059  -130.09614569   680.2761913    648.35969456
 680.27613288   579.50278028  -168.55212136   373.64253857
 680.27631097  -534.47635878 -1585.22787562    43.34279529
 212.86093002   680.27627499   680.27437859   497.71196703]
```

- When the function prints "Optimal x" and "Optimal lambda," it's displaying the entire vector solutions for the primal and dual problems, respectively. Where x is a vector of length n (150), and lambda is a vector or length m (100).
- Some entries in x are extremely close to zero. This suggests that the corresponding variables contribute minimally to the optimal solution, however, the Non-zero entries indicate variables that are actively contributing to the optimal solution.
- The Large positive values in lambda indicate constraints that are significantly affecting the objective value. A high positive value suggests a tight constraint where a slight relaxation could substantially increase the objective value.
- Each entry in this history provides snapshots of the values of the primal variables x, dual variables λ, and slack variables s at each step of the algorithm. It is very lengthy which is why i have commented it out.

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

def interiorLP(p, A, b, x0, tol, mu):
    n = len(x0)  # Number of variables
    m = A.shape[0]  # Number of equality constraints
    tau = 1.0
    max_iter = 100
    history = []

    # Initialize x, s (slack variables), and lambda_ (dual variables for equ
    x = np.maximum(x0, 1e-2)  # Ensure x is strictly positive
    s = np.ones(n)            # Slack variables initially set equal
    lambda_ = np.zeros(m)     # Dual variables for equality constraints

    iteration_count = 0
    for iteration in range(max_iter):
        iteration_count += 1
        # Form the system of KKT conditions
        X = np.diag(x)
        S = np.diag(s)

        # Calculate the residuals
        r_pri = A @ x - b
        r_dual = p - A.T @ lambda_ - s
        r_cent = x * s - tau * np.ones(n)

        # Construct KKT Matrix
```

```
        M = np.block([
            [np.zeros((n, n)), A.T, np.eye(n)],
            [A, np.zeros((m, m)), np.zeros((m, n))],
            [S, np.zeros((n, m)), X]
        ])

        # Solve for directions
        r = np.hstack([-r_dual, -r_pri, -r_cent])
        delta = np.linalg.solve(M, r)

        dx = delta[:n]
        dlambda = delta[n:n+m]
        ds = delta[n+m:]

        # Line search for maximum step size that maintains non-negativity
        alpha_x = min(1, *[-xi/dxi for xi, dxi in zip(x, dx) if dxi < 0])
        alpha_s = min(1, *[-si/dsi for si, dsi in zip(s, ds) if dsi < 0])
        alpha = min(alpha_x, alpha_s, 0.99)

        # Update x, lambda, and s
        x += alpha * dx
        lambda_ += alpha * dlambda
        s += alpha * ds
        tau /= mu

        # Store history
        history.append((x.copy(), lambda_.copy(), s.copy()))

        # Check convergence (duality gap)
        gap = np.dot(x, s)
        if gap < tol:
            break

    return iteration_count, x, lambda_, history

# Set up test data
seed = 9727
rng = np.random.default_rng(seed)
n = 150
p = 100
A = np.hstack((rng.standard_normal((p, n-p)), np.eye(p)))
b = A @ rng.random(n)
pobj = np.concatenate((rng.standard_normal(n-p), np.zeros(p)))
x0 = np.maximum(0, rng.random(n))  # Initial feasible point

# Test with mu = 2
iter_count_mu2, x_opt2, lambda_opt2, history2 = interiorLP(pobj, A, b, x0, 1
print(f"Total iterations with mu=2: {iter_count_mu2}")

# Test with mu = 10
iter_count_mu10, x_opt10, lambda_opt10, history10 = interiorLP(pobj, A, b, x
print(f"Total iterations with mu=10: {iter_count_mu10}")
```

```
Total iterations with mu=2: 29
Total iterations with mu=10: 100
```

- With μ=2, the method required 29 iterations to decrease the duality gap to 10^−6. This suggests a relatively efficient convergence.
- With μ=10, the method hit the maximum iteration limit of 100 without reaching the convergence criterion of reducing the duality gap to 10^−6. It is possible that the path taken by the algorithm jumps too quickly towards the boundary of the feasible region.

4. Interior point barrier method for a QP

4. Interior point barrier method for a QP. Implement a basic log-barrier interior point method for solving the convex quadratic program:

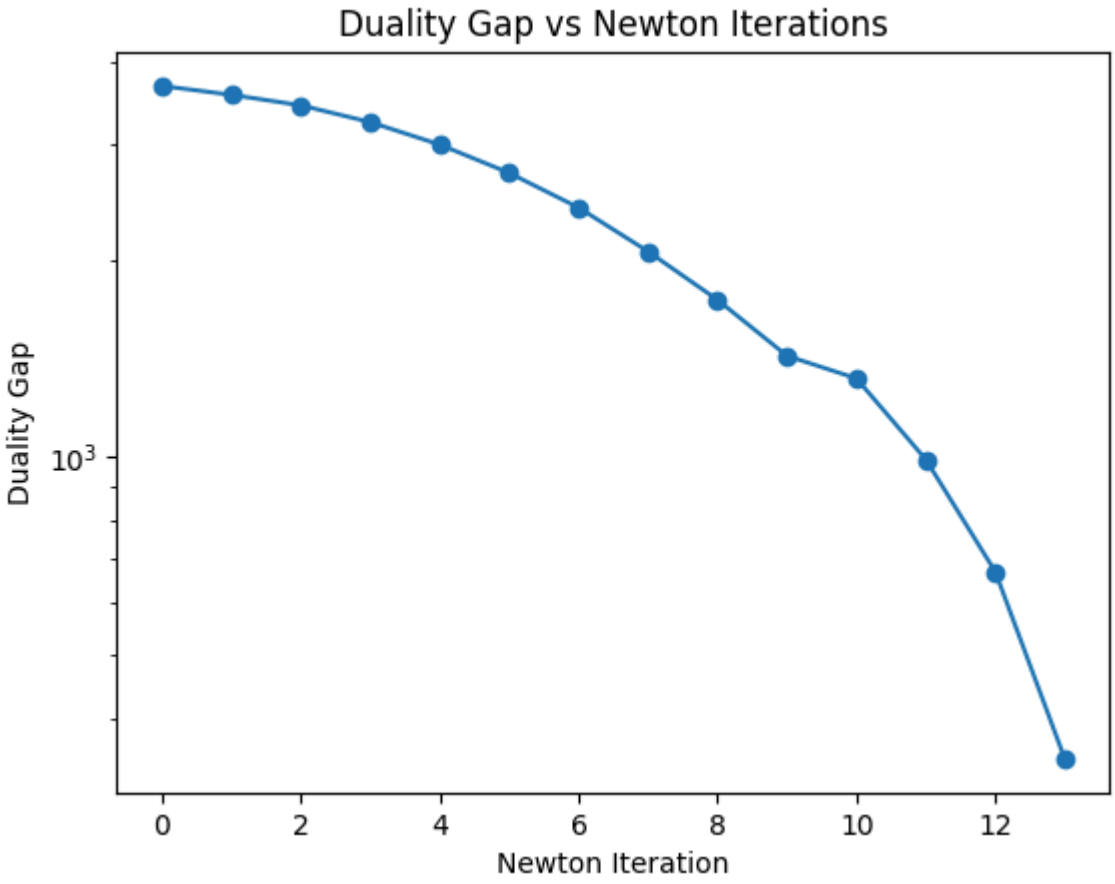
$$\begin{array}{ll} \text{minimize} & 1/2x^T Hx + g^T x \\ \text{subject to} & Ax - b \leq 0 \end{array}$$

- Write a routine to generate an initial feasible point, or determine that the problem is infeasible.
- Generate random data (symmetric positive definite H) and use them to test the performance of the method by plotting duality gap vs number of Newton iterations (use $n = 200$, $m = 100$).
- Is there a variation on the basic method that does not require nested iteration?

In [ ]:
```python
# Function to generate a random symmetric positive definite matrix
def generate_positive_definite_matrix(n):
    A = np.random.rand(n, n)
    return np.dot(A, A.transpose())

# Function to find an initial feasible point for the QP
def generate_feasible_start(A, b):
    res = linprog(np.ones(A.shape[1]), A_ub=A, b_ub=b, method='highs')
    if res.success:
        return res.x
    else:
        return None

# Interior point method implementation for a QP
def interior_point_method(H, g, A, b, x0, max_iter=200):
    m, n = A.shape
    mu = 10  # Barrier parameter
    tol = 1e-5  # Tolerance for stopping criterion
    duality_gaps = []

    x = x0
    for i in range(max_iter):
        # Define the barrier function and its derivatives
        f = lambda x: 0.5 * np.dot(x, H @ x) + np.dot(g, x) + mu * np.sum(-r
        grad_f = lambda x: H @ x + g - mu * np.sum(A.T / (b - A @ x), axis=1
        hess_f = lambda x: H + mu * A.T @ np.diag(1 / (b - A @ x)**2) @ A

        # Newton's method to find the search direction
        grad = grad_f(x)
        hess = hess_f(x)
```

```
        delta_x = np.linalg.solve(hess, -grad)

        # Line search to find the step size
        t = 1
        while np.any(A @ (x + t * delta_x) >= b):
            t *= 0.9

        # Update x
        x_new = x + t * delta_x

        # Check for convergence (i.e., if the step size is very small)
        if np.linalg.norm(x_new - x) < tol:
            x = x_new
            break

        x = x_new

        # Calculate the duality gap (estimation)
        primal_obj = 0.5 * np.dot(x, H @ x) + np.dot(g, x)
        dual_obj = -np.sum(mu * np.log(b - A @ x))  # This is an estimation
        duality_gap = primal_obj - dual_obj
        duality_gaps.append(duality_gap)

    return x, duality_gaps

# Generate random data
n = 200  # Dimension of x
m = 100  # Number of constraints
H = generate_positive_definite_matrix(n)
g = np.random.rand(n)
A = np.random.rand(m, n)
b = np.random.rand(m) + A @ np.random.rand(n)  # ensure that b is beyond a r

# Generate an initial feasible point
x0 = generate_feasible_start(A, b)
if x0 is not None:
    # Run the interior point method
    solution, duality_gaps = interior_point_method(H, g, A, b, x0)

    #print the initial feasible point
    print("The problem is feasible, an initial point was found")

    # Plot the duality gap vs Newton iterations
    plt.plot(duality_gaps, marker='o')
    plt.xlabel('Newton Iteration')
    plt.ylabel('Duality Gap')
    plt.title('Duality Gap vs Newton Iterations')
    plt.yscale('log')  # Log scale might make it easier to see convergence
    plt.show()

else:
    print("The problem is infeasible or an initial point couldn't be found."
```

```
The problem is feasible, an initial point was found
```



The duality gap decreases as newton iteration increses and gets closer to the optimal point, as expected.

Yes, there are variations of the interior point methods that don't require nested iteration. (Nested iteration: Which typically refers to the need to solve a series of barrier subproblems at progressively lower levels of the barrier parameter)

One variation is the **primal-dual interior point method**. This method updates both the primal and dual variables simultaneously within each iteration.