Numerical Optimization - Assignment 10

David Alvear(187594) - Nouf Farhoud(189731)

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

1. Trust region constraint

- **Dual Problem**

Lagrangian
$L(p, \lambda) = p^T H p + 2g^T p + \lambda(p^T p - \Delta^2)$

Find the dual problem
$\nabla_p L(p, \lambda) = 2Hp + 2g + 2\lambda p = 0$
$\Rightarrow 2H^T p + 2\lambda p + 2g = 2(H + \lambda I)p + 2g = 0$
$p = -(H + \lambda I)^{-1} g$

Replace p in $L(p, \lambda)$:
$$\begin{aligned} & g^T(H + \lambda I)^{-1} T H (H + \lambda I)^{-1} g - 2g^T(H + \lambda I)^{-1} g \\ & \quad + \lambda(g^T(H + \lambda I)^{-1} T (H + \lambda I)^{-1} g - \Delta^2) \\ & = g^T(H + \lambda I)^{-1} T (H + \lambda I)(H + \lambda I)^{-1} g \\ & \quad - 2g^T(H + \lambda I)^{-1} g - \lambda \Delta^2 \\ & = -g^T(H + \lambda I)^{-1} g - \lambda \Delta^2 \end{aligned}$$

Then:
$$g(\lambda) = \begin{cases} -g^T(H + \lambda I)^{-1} g - \lambda \Delta^2 & \text{if } (H + \lambda I) \geq 0 \\ -\infty & \text{otherwise} \end{cases}$$

Dual Problem:
maximize $-g^T(H + \lambda I)^{-1} g - \lambda \Delta^2$
subject to $H + \lambda I \geq 0$

- **Computationally oriented form**

use eigendecomposition: $H = V\Lambda V^{-1}$

$= \lambda V V^{-1} - V\Lambda V^{-1} \rightarrow \lambda > -\lambda_{\min}(H)$
$= -g^T(V\Lambda V^T + \lambda V V^T)^{-1} g$
$= -g^T V(\Lambda + \lambda I)^{-1} V^T g$
$= -(V^T g)^T(\Lambda + \lambda I)^{-1}(V^T g)$

Since $(\Lambda + \lambda I)$ is a diagonal matrix, we can use the diagonal values $\dfrac{1}{(\lambda_i + \lambda)}$

then:

$$-(V^T g)^T(\Lambda + \lambda I)^{-1}(V^T g) = -\sum_{i=1}^{n} \frac{(q_i^T g)^2}{(\lambda_i + \lambda)}$$

Finally, the computationally oriented form:

$$\text{maximize } -\sum_{i=1}^{n} \frac{(q_i^T g)^2}{(\lambda_i + \lambda)} - \lambda\Delta^2$$
Subject to $\lambda \geq -\lambda_{\min}(H)$

Pseudocode

Trust Region Subproblem Algorithm
INPUT: g, H, Δ
OUTPUT: p^*
PROCEDURE:
  Initialize λ as the minimum eigenvalue of H
  Compute eigenpairs (λ_i, q_i) of H
  WHILE not converged DO
    Compute dual function value $f(\lambda)$:

$$f \leftarrow -\sum_{i=1}^{n} \left(\frac{(q_i^T g)^2}{\lambda_i + \lambda} \right) - \lambda \Delta^2$$

    Compute derivative of dual function $f'(\lambda)$:

$$f' \leftarrow \sum_{i=1}^{n} \left(\frac{(q_i^T g)^2}{(\lambda_i + \lambda)^2} \right) - \Delta^2$$

    Update λ using Newton's method:

$$\lambda \leftarrow \lambda - \frac{f}{f'}$$

    Check for convergence
  END WHILE
  Compute optimal minimizer p^* :
    $p^* \leftarrow -(H + \lambda I)^{-1} g$
  RETURN p^*
END PROCEDURE

2. Levenberg-Marquardt

Levenberg-Marquardt routine

In [ ]:
```python
# Levenberg-Marquardt routine
def levenberg_marquardt(f, Df, x0, lambda0, y, d, kmax=100, tol=1e-6):
    n = len(x0)
    x, lam = x0, lambda0
    obj = []
    res = []
    path = []
    lam_history = [lam]
    for k in range(kmax):
        Rx = r(x, y, d)
        Jx = Df(x, y, d)
        obj.append(np.sum(Rx**2))
        res.append(np.linalg.norm(2*Jx.T @ Rx))
        if res[-1] < tol:
            break
        # Compute the xt
        xt = x - np.linalg.solve(Jx.T @ Jx + lam * np.eye(n), Jx.T @ Rx)
```

```
        # Check if the update has decreased the residual
        if np.linalg.norm(r(xt, y, d)) < np.linalg.norm(Rx):
            lam *= 0.8  # Decrease lambda
            x = xt
        else:
            print("increased")
            lam *= 2.0  # Increase lambda
        path.append(x.copy())
        lam_history.append(lam)
    return x, {'objective': obj, 'residual': res, 'lambda' : lam_history, 'x
```

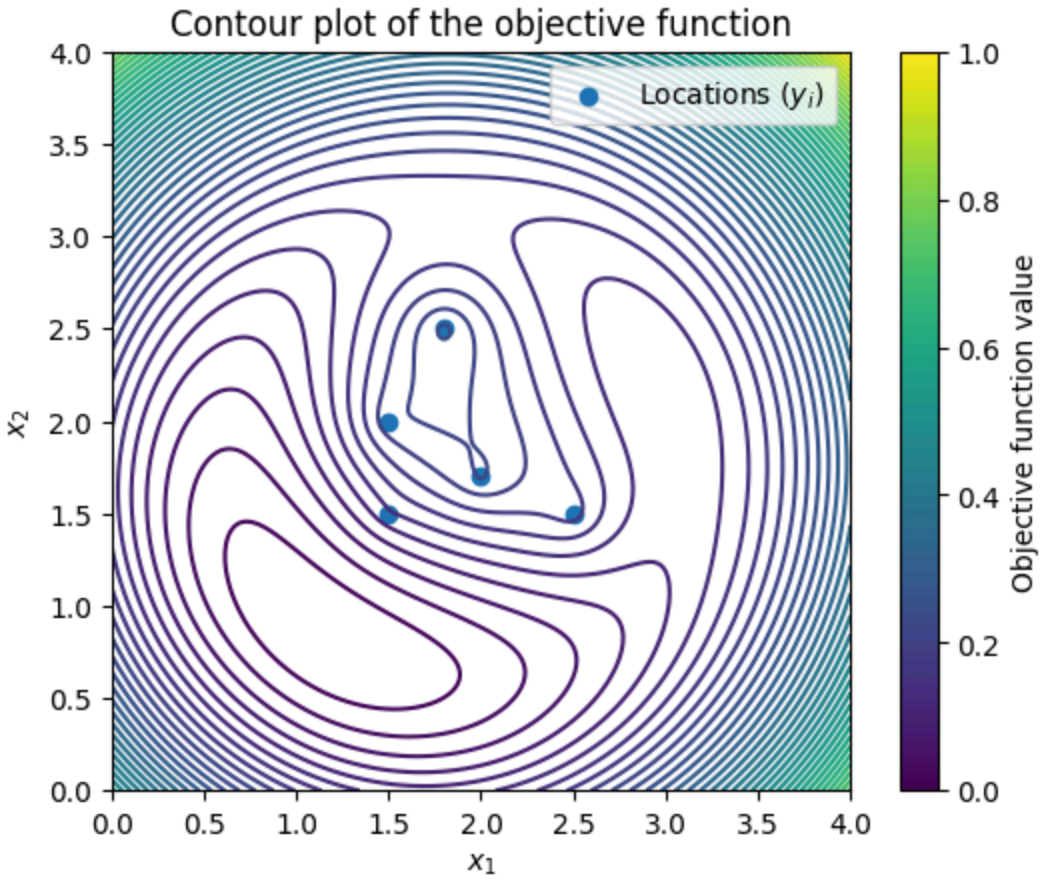
Estrategy of reducing lambda

Lambda controls the influence of the algorithm to behave between gradient descent and Gauss-newton method. When it is large the algorithm behaves more like as gradient descent. When it is small it behaves as a gauss-newton. The algorithm adjusts lambda based on the new estimate of the solution x_t. If the norm of the residuals decreases it means that the step was successful and the algorithm will decrease the lambda value under the assumption to be closer to the minimum. Conversely, if the norm of the residual does not decrease indicates that the step might be not optimal. Then, the lambda value will be increased (doubled).

Plot countour lines

In [ ]:
```
# five locations yi in a 5 by 2 matrix
A = np.array([[1.8, 2.5], [2.0, 1.7], [1.5, 1.5], [1.5, 2.0], [2.5, 1.5]])
# vector of measured distances to the five locations
d = np.array([1.87288, 1.23950, 0.53672, 1.29273, 1.49353])

# Function to calculate the residuals
def r(x, y, d):
    return np.linalg.norm(x - y, axis=1) - d

# Function to calculate the objective function value for a given x
def function(x, y, d):
    return np.sum(r(x, y, d)**2)

def Dfunction(x, y, d):
    jac = np.zeros((len(y), len(x)))
    for i, (yi, di) in enumerate(zip(y, d)):
        ri = np.linalg.norm(x - yi) - di
        if ri != 0:  # Avoid division by zero
            jac[i, :] = (x - yi) / np.linalg.norm(x - yi)
    return jac

# Create a grid of x values
x_values = np.linspace(0, 4, 400)
y_values = np.linspace(0, 4, 400)
X, Y = np.meshgrid(x_values, y_values)
f = np.zeros_like(X)
df = []
```

```
# Calculate the objective function value for each x in the grid
for i in range(X.shape[0]):
    for j in range(X.shape[1]):
        f[i, j] = function(np.array([X[i, j], Y[i, j]]), A, d)
        df.append(Dfunction(np.array([X[i, j], Y[i, j]]), A, d))

# Plot data and the contour lines of the objective function
ax = plt.gca()
plt.scatter(A[:, 0], A[:, 1], label='Locations ($y_i$)')
ax.contour(X, Y, f, levels=50, cmap='viridis')
ax.set_xlim([0, 4])
ax.set_ylim([0, 4])
ax.set_aspect('equal')

# Label plot
plt.title('Contour plot of the objective function')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.legend()
plt.colorbar(label='Objective function value')
plt.show()
```

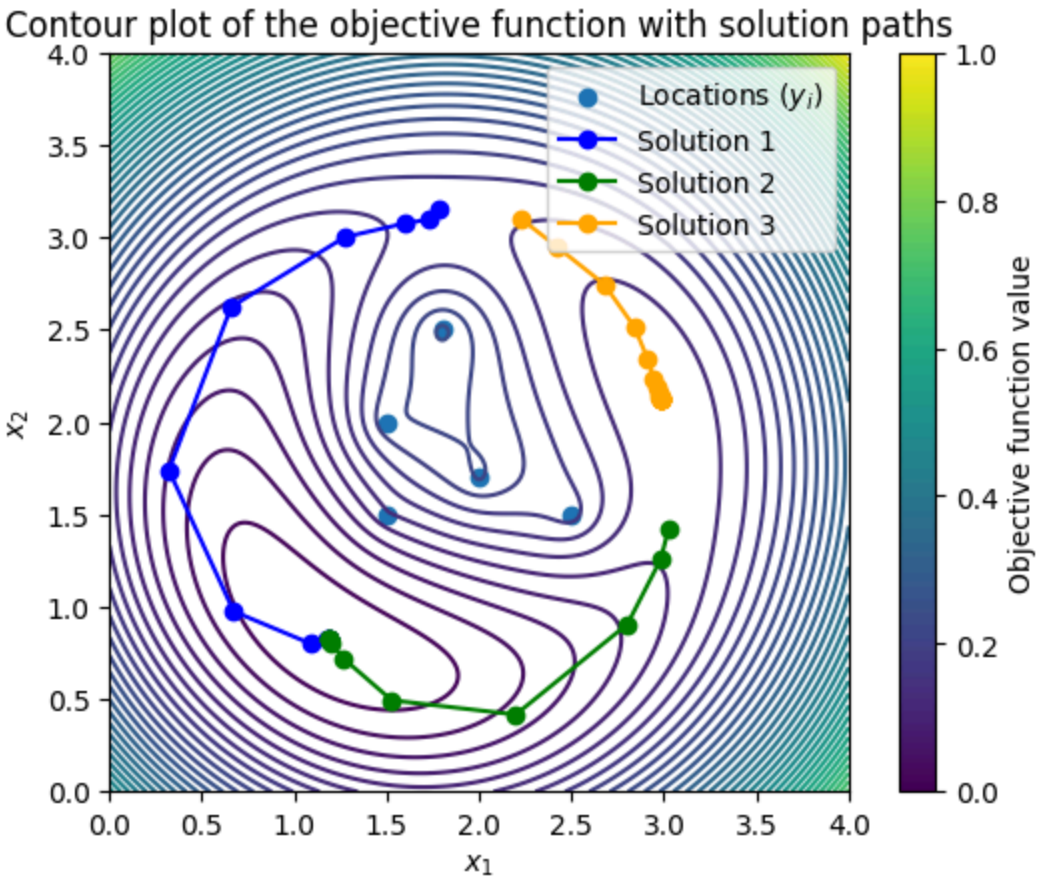


In [ ]:
```
initial_points = [np.array([1.8, 3.5]), [3.0, 1.5], [2.2, 3.5]]
lambda0 = 1
solutions = []

# solve for the initial points
for x0 in initial_points:
```

```
  x, solution = levenberg_marquardt(function, Dfunction, x0, lambda0, A, d)
  solutions.append(solution)
```

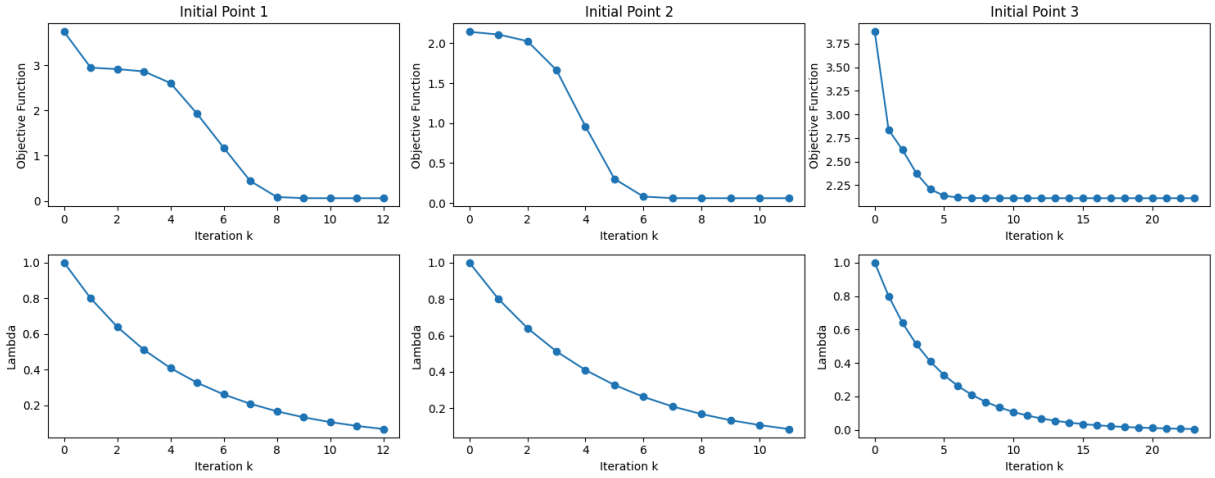
In [ ]:

```
colors = ['blue', 'green', 'orange']
labels = ['Solution 1', 'Solution 2', 'Solution 3']

# Plot data and the contour lines of the objective function
ax = plt.gca()
plt.scatter(A[:, 0], A[:, 1], label='Locations ($y_i$)')
ax.contour(X, Y, f, levels=50, cmap='viridis')
ax.set_xlim([0, 4])
ax.set_ylim([0, 4])
ax.set_aspect('equal')

# Label plot
plt.title('Contour plot of the objective function')
plt.xlabel('$x_1$')
plt.ylabel('$x_2$')
plt.legend()
plt.colorbar(label='Objective function value')

# Solve and plot paths
for x0, color, label, sol in zip(initial_points, colors, labels, solutions):
    path = sol['x_h']
    ax.plot(path[:, 0], path[:, 1], marker='o', linestyle='-', color=color,

ax.set_title('Contour plot of the objective function with solution paths')
ax.set_xlabel('$x_1$')
ax.set_ylabel('$x_2$')
ax.legend()
plt.show()
```



We can see that we solved the problem using three different initial points. It is evident that the algorithm behavour depends on the initial condition. For solutions 1 and 2 we have the same result and minimum value, but the algorithm with the 3 initial point found a different local minimum and returned the solution for that case.

In [ ]:
```
# Plotting the solutions and the convergence plots
fig, axs = plt.subplots(2, len(solutions), figsize=(15, 6))

for i, (solution, ax_obj, ax_lam) in enumerate(zip(solutions, axs[0], axs[1]
    info = solution
    k = np.arange(len(info['objective']))
    # Plot objective function vs. k
    ax_obj.plot(k, info['objective'], '-o')
    ax_obj.set_title(f'Initial Point {i+1}')
    ax_obj.set_xlabel('Iteration k')
    ax_obj.set_ylabel('Objective Function')

    # Plot lambda_k vs. k
    ax_lam.plot(k, info['lambda'], '-o')
    ax_lam.set_xlabel('Iteration k')
    ax_lam.set_ylabel('Lambda')

plt.tight_layout()
plt.show()
```



We found that solutions 1 and 2 have the same results, and it is seen in the graphs that the lambda value was decreased gradually and the objective function decreased around 6 and 8 iterations to the minimum. In solution 1 we see a slow convergence at the begining. The solution 3 found a different local minimum and drastically reduced the objective function in the first 3 iterations, while the other two initial conditions the solution started to converge after the 6 iteration.

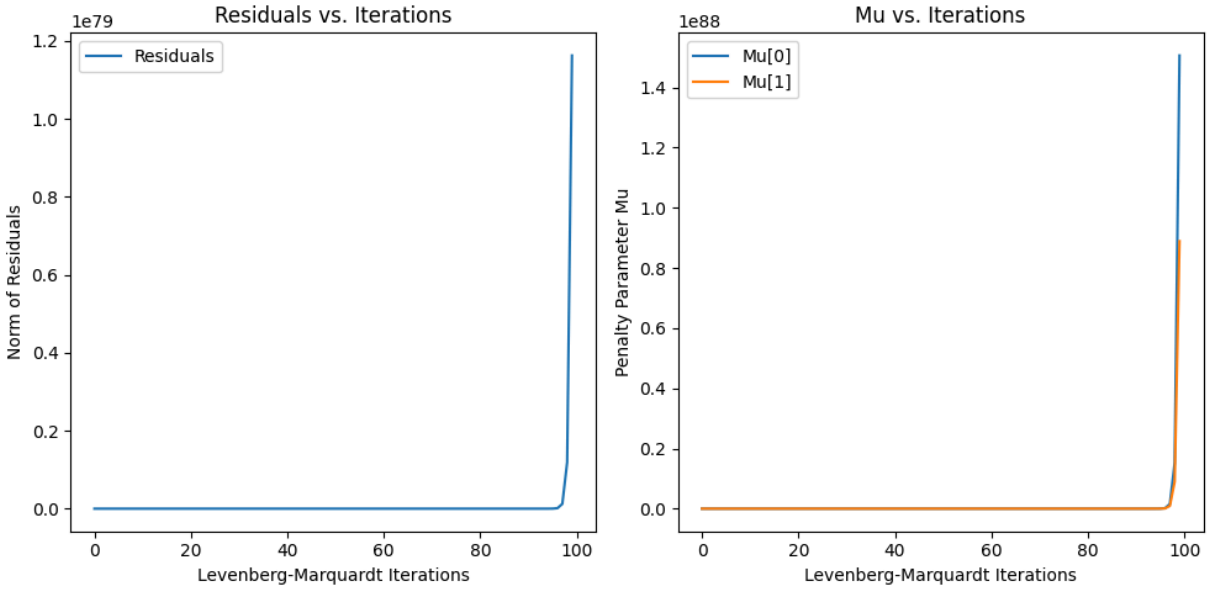
3. Augmented Lagrangian

In [ ]:
```python
import numpy as np
from scipy.optimize import least_squares, minimize
import matplotlib.pyplot as plt

# Define the objective function
def objective(x):
    return (x[0] - 1)**2 + (x[1] - 1)**2 + (x[2] - 1)**2

# Define the gradient of the objective function
def grad_objective(x):
    return 2 * (x - np.array([1, 1, 1]))

# Define the constraints
def constraints(x):
    return np.array([
        x[0]**2 + 0.5*x[1]**2 + x[2]**2 - 1,
        0.8*x[0]**2 + 2.5*x[1]**2 + x[2]**3 + 2*x[1]*x[2] - x[0] - x[1] - x[
    ])

# Define the Jacobian of the constraints
def jac_constraints(x):
    return np.array([
        [2*x[0], x[1], 2*x[2]],
        [1.6*x[0] - 1, 5*x[1] + 2*x[2] - 1, 3*x[2]**2 + 2*x[1] - 1]
    ])

# Define the combined residuals for least_squares
def fun_aug_lagrangian(x, mu, c):
```

```
    objective_residual = objective(x)
    constraint_residuals = constraints(x)
    # Combine into a single array of residuals
    return np.hstack([
        [objective_residual],  # Corrected to make this an array of the same
        mu * constraint_residuals + (c / 2) * constraint_residuals**2
    ])

# Augmented Lagrangian method using Levenberg-Marquardt
def augmented_lagrangian_method_lm(x0, mu, c, tol, max_iter):
    aug_lagrangian_residuals = []
    mu_values = []

    for iteration in range(max_iter):
        result_lm = least_squares(fun_aug_lagrangian, x0, args=(mu, c), meth
        objective_res = result_lm.fun[0]
        constraints_res = result_lm.fun[1:]
        aug_lagrangian_residuals.append(np.linalg.norm(constraints_res))
        mu_values.append(mu.copy())

        if np.linalg.norm(constraints_res) < tol and np.linalg.norm(grad_obj
            break

        mu = mu + c * constraints(result_lm.x)
        c *= 10
        x0 = result_lm.x

    return result_lm.x, aug_lagrangian_residuals, mu_values

# Penalty function method
r = 1.0  # Initial penalty parameter
def penalty_method(x0, r, tol, max_iter):
    for iteration in range(max_iter):
        def penalty_function(x):
            return objective(x) + (r/2) * np.sum(constraints(x)**2)
        result = minimize(penalty_function, x0, method='trust-constr', optic
        res = constraints(result.x)
        if np.linalg.norm(res) < tol:
            break
        r *= 10
        x0 = result.x
    return result.x

# Set initial values and parameters
x0 = np.array([0.0, 0.0, 0.0])
mu = np.array([1.0, 1.0])
c = 1.0
tol = 1e-5
max_iter = 100

# Call the methods
solution_aug_lag, residuals_aug_lag, mu_values = augmented_lagrangian_method
solution_penalty = penalty_method(x0, r, tol, max_iter)

# Output the solution
print("Augmented Lagrangian Method solution:", solution_aug_lag)
```

```
print("Penalty Method solution:", solution_penalty)

# Plot the residuals and the penalty parameter mu versus the cumulative numb
plt.figure(figsize=(10, 5))

# Plot residuals
plt.subplot(1, 2, 1)
plt.plot(residuals_aug_lag, label='Residuals')
plt.xlabel('Levenberg-Marquardt Iterations')
plt.ylabel('Norm of Residuals')
plt.title('Residuals vs. Iterations')
plt.legend()

# Plot penalty parameter mu
plt.subplot(1, 2, 2)
plt.plot([mu[0] for mu in mu_values], label='Mu[0]')
plt.plot([mu[1] for mu in mu_values], label='Mu[1]')
plt.xlabel('Levenberg-Marquardt Iterations')
plt.ylabel('Penalty Parameter Mu')
plt.title('Mu vs. Iterations')
plt.legend()

plt.tight_layout()
plt.show()
```

```
Augmented Lagrangian Method solution: [0.71709427 0.48385248 0.60722253]
Penalty Method solution: [0.71742982 0.48402296 0.60676103]
```



1)The first graph shows that the residuals stay relatively constant throughout most iterations and then abruptly converge to a very low value at the end. This suggests that the method is converging rapidly once it is near a feasible solution that satisfies the constraints.

2)The second graph indicates that the penalty parameters μ[0] and μ[1] remain fairly steady through most of the optimization process before both increasing exponentially in the last few iterations. this increase indicates that the Augmented Lagrangian method

intensifying the penalty for constraint violations as it gets closer to finding a solution that meets the constraints within the desired tolerance.

comparing Penalty findings with the previous results for the Augmented Lagrangian method:

- Both methods appear to converge quickly once they near a feasible solution.
- The residuals' graph implies a stable solution was reached by the end of the optimization process.

Comparing our output: Augmented Lagrangian Method solution: [0.71709427 0.48385248 0.60722253] Penalty Method solution: [0.71742982 0.48402296 0.60676103]

- The values are very close to each other, with slight differences in the third decimal place.

4. Planar disk contact

Problem Formulation

$$\text{Problem Formulation}$$

$$\text{Objective Function:} \quad f(\mathbf{x}) = \sum_{i=1}^{9} K_i \Delta i \rightarrow \text{Energy}$$

$$\text{Variables:} \quad \mathbf{x}_i \rightarrow \text{Disk Positions}$$

$$\mathbf{x}_i = (x_i, y_i)$$

$$\text{Constraints:}$$

$$\text{Space Constraints:} \quad \|\mathbf{x}_i - \mathbf{x}_j\|^2 \geq (r_i + r_j)^2 \quad \text{for } i \neq j$$

$$\mathbf{x}_i \leq 1 - r_i, \quad i = 1...4$$

$$\mathbf{x}_i \geq r_i, \quad i = 1...4$$

$$\text{Minimize} \quad f(\mathbf{x}) = \sum_{i=1}^{9} K_i \Delta i$$

$$\text{Subject to} \quad \|\mathbf{x}_i - \mathbf{x}_j\|^2 \geq (r_i + r_j)^2 \quad \text{for } i \neq j$$

$$\mathbf{x}_i \leq 1 - r_i, \quad i = 1...4$$

$$\mathbf{x}_i \geq r_i, \quad i = 1...4$$

In [ ]:
```python
import cvxpy as cp
import numpy as np
from scipy.optimize import minimize

def find_equilibrium_configuration_scipy(n_disks, radii, spring_constants, c
    ### Initial guess
```

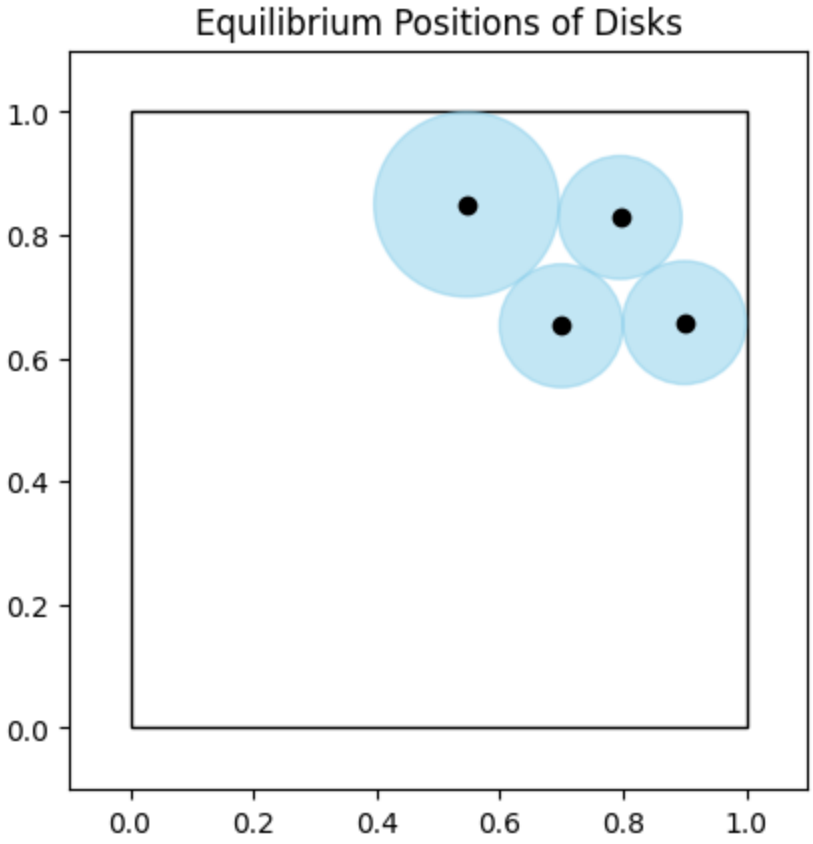
```
    x0 = np.hstack([np.linspace(0.1, 0.9, n_disks), np.linspace(0.1, 0.9, n_

    # Objective function
    def energy(x):
        e = 0
        # Disks energy
        for (i, j), k in connections.items():
            # e += 0.5 * spring_constants[k] * ((x[i] - x[j + n_disks])**2 +
            e += 0.5 * spring_constants[k] * ((x[i] - x[j ])**2 + (x[i + n_d
        # Corners energy
        for (ci, di), k in corners_connections.items():
            e += 0.5 * spring_constants[k] * ((x[di] - corners[ci][0])**2 +
        return e

    # Constraints
    cons = []
    for i in range(n_disks):
        # Disks must be inside the unit square
        cons.append({'type': 'ineq', 'fun': lambda x, i=i: x[i] - radii[i]})
        cons.append({'type': 'ineq', 'fun': lambda x, i=i: 1 - radii[i] - x[
        cons.append({'type': 'ineq', 'fun': lambda x, i=i: x[i + n_disks] -
        cons.append({'type': 'ineq', 'fun': lambda x, i=i: 1 - radii[i] - x[

    # Non-interpenetration constraints for each pair of disks
    for i in range(n_disks):
        for j in range(i + 1, n_disks):
            cons.append({'type': 'ineq', 'fun': lambda x, i=i, j=j: (x[i] -

    # Perform the optimization
    result = minimize(energy, x0, constraints=cons, method='SLSQP')

    # Check if the optimization was successful
    if result.success:
        # Extract positions
        positions = np.vstack([result.x[:n_disks], result.x[n_disks:]]).T
        return positions, result
    else:
        return result.message, result

n_disks = 4
radii = [0.1, 0.1, 0.1, 0.15]  # Example radii
corners = [(0,1), (1,1), (1,0), (0,0)]
spring_constants = {'k0': 1, 'k1': 10, 'k2': 10, 'k3': 10, 'k4': 1, 'k5': 1,
connections = { # Define the connections between the disks and the springs
    (0, 1): 'k1',
    (1, 2): 'k3',
    (2, 3): 'k5',
    (3, 0): 'k7',
    (3, 1): 'k8',
}
corners_connections = { # Define the connections between the disks and the c
    (0, 0): 'k0',
    (1, 1): 'k2',
    (2, 2): 'k4',
    (3, 3): 'k6',
}
```

```
positions, result = find_equilibrium_configuration_scipy(n_disks, radii, spr
print(positions)
print(result)
```

```
[[0.70006977 0.65282742]
 [0.79545576 0.82861575]
 [0.9        0.65811498]
 [0.54637198 0.85      ]]
 message: Optimization terminated successfully
 success: True
  status: 0
     fun: 2.218123200879361
       x: [ 7.001e-01  7.955e-01  9.000e-01  5.464e-01  6.528e-01
            8.286e-01  6.581e-01  8.500e-01]
     nit: 27
     jac: [-1.001e-01  3.538e-01  1.299e+00 -2.452e+00 -2.302e+00
            1.535e+00 -1.239e+00  1.453e+00]
    nfev: 237
    njev: 25
```

In [ ]:

```
# The positions of the disks at the equilibrium configuration
disk_positions = positions.reshape(4, 2)

# Plotting the disk positions
fig, ax = plt.subplots()
square = plt.Rectangle((0,0), 1, 1, fill=False)
ax.add_artist(square)

# Plot each disk
for pos, radius in zip(positions, radii):
    disk = plt.Circle((pos[0], pos[1]), radius, fill=True, color='skyblue',
    ax.add_artist(disk)
    # Mark the center of the disk
    ax.plot(pos[0], pos[1], 'ko')

# Set equal scaling and limits
ax.set_xlim(-0.1, 1.1)
ax.set_ylim(-0.1, 1.1)
ax.set_aspect('equal', 'box')
ax.set_title('Equilibrium Positions of Disks')

plt.show()
```



- Write the lagrangian of the problem. What is the significance of the multipliers?

$$\mathcal{L}(\mathbf{x}, \lambda) = \sum_{i=1}^{9} K_i \Delta_i + \sum_{i=1}^{9} \sum_{j \neq i} \lambda_{ij} \left[\|\mathbf{x}_i - \mathbf{x}_j\|^2 - (r_i + r_j)^2 \right] + \sum_{i=1}^{9} \lambda_i^{(2)} (\mathbf{x}_i - (1 -$$

We have lagrange multipliers for the non-interpenetration of the disks and the boundary limits. The lagrange multipliers represents how active is the constraint in the problem, and for the disk penetration we can see that effectively there exist contact between the disks, except disk 3 that only has conact with disk 4. Similarly, disk 1 is in contact with the upper boundary limiting the movement of the disk and the group. Physically, the lagrange multipliers represent the forces made between the boundaries of the square with the disks, as well as, the force between the disks. We can see that for disks 1,2,4 there exist forces between them to maintatin the equilibrium.

5. Facility location

In [ ]:
```
pip install pulp
```

```
Collecting pulp
  Downloading PuLP-2.8.0-py3-none-any.whl (17.7 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 17.7/17.7 MB 29.5 MB/s eta 0:0
0:00
Installing collected packages: pulp
Successfully installed pulp-2.8.0
```

In [ ]:

```python
import pulp

# Define the data
warehouse_demand = [15, 18, 14, 20]
plant_capacity = [20, 22, 17, 19, 18]
fixed_costs = [12000, 15000, 17000, 13000, 16000]
transportation_costs = [
    [4000, 2000, 3000, 2500, 4500],
    [2500, 2600, 3400, 3000, 4000],
    [1200, 1800, 2600, 4100, 3000],
    [2200, 2600, 3100, 3700, 3200]
]

# Define the problem
prob = pulp.LpProblem("FacilityLocation", pulp.LpMinimize)

# Decision variables
plant_vars = pulp.LpVariable.dicts("Plant", range(5), 0, 1, pulp.LpBinary)
transport_vars = pulp.LpVariable.dicts("Transport", (range(5), range(4)), 0)

# Objective function
prob += (
    pulp.lpSum(fixed_costs[i] * plant_vars[i] for i in range(5)) +
    pulp.lpSum(transportation_costs[j][i] * transport_vars[i][j] for i in ra
), "Total Costs"

# Constraints
for j in range(4):  # Demand constraints
    prob += pulp.lpSum(transport_vars[i][j] for i in range(5)) >= warehouse_

for i in range(5):  # Supply constraints
    prob += pulp.lpSum(transport_vars[i][j] for j in range(4)) <= plant_capa

# Solve the problem
prob.solve()

# Output the results
print("Status:", pulp.LpStatus[prob.status])
for i in range(5):
    print(f"Plant {i+1} is {'open' if plant_vars[i].varValue == 1 else 'clos

# Additionally, you could print out the shipping quantities if needed
for i in range(5):
    for j in range(4):
        if transport_vars[i][j].varValue > 0:
            print(f"Plant {i+1} ships {transport_vars[i][j].varValue} thousa
```

```
Status: Optimal
Plant 1 is open.
Plant 2 is open.
Plant 3 is closed.
Plant 4 is open.
Plant 5 is open.
Plant 1 ships 14.0 thousands of units to Warehouse 3.
Plant 1 ships 6.0 thousands of units to Warehouse 4.
Plant 2 ships 14.0 thousands of units to Warehouse 1.
Plant 2 ships 8.0 thousands of units to Warehouse 4.
Plant 4 ships 1.0 thousands of units to Warehouse 1.
Plant 4 ships 18.0 thousands of units to Warehouse 2.
Plant 5 ships 6.0 thousands of units to Warehouse 4.
```