Electrical Engineering Department

Sharif University of Technology

Convex optimization 1

AMIRHOSSEIN NAGHDI - 400102169

2024

Q1.DCP

part 1

Solution:

(a)

Original:

$$\|(x, y, z)\|_2^2 \leq 1$$

Convexity Analysis: The squared Euclidean norm is a convex function. Its level set $\|(x, y, z)\|_2^2 \leq 1$ is convex because the sublevel sets of a convex function are convex.

DCP Expression:

$$x^2 + y^2 + z^2 \leq 1$$

Explanation: The squared norm expands to $x^2 + y^2 + z^2$, which is a valid DCP-compatible form because x^2, y^2, and z^2 are convex, and summing convex functions preserves convexity.

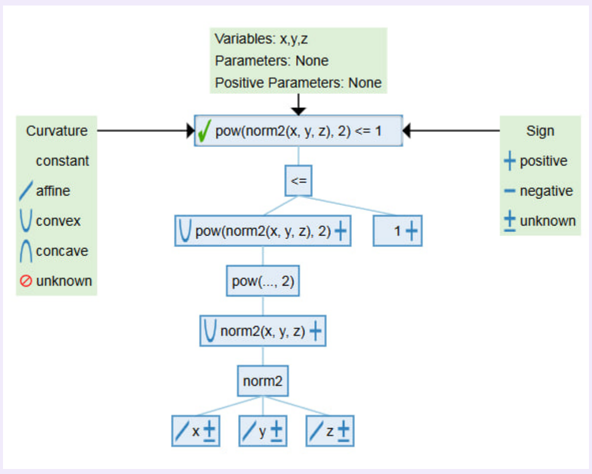

Figure 1: $||x^2 + y^2 + z^2||_2^2 \leq 1$

(b)

Original:

$$\sqrt{x^2 + 1} \leq 3x + y$$

Convexity Analysis: The function $\sqrt{x^2 + 1}$ is convex, and $3x + y$ is affine (hence convex). The inequality describes a convex set because it bounds a convex function by another convex function.

DCP Expression:

$$\sqrt{x^2 + 1} - 3x - y \leq 0$$

Explanation: Rewriting preserves equivalence. The DCP rules allow subtraction of affine functions from convex ones, as the resulting function remains convex.

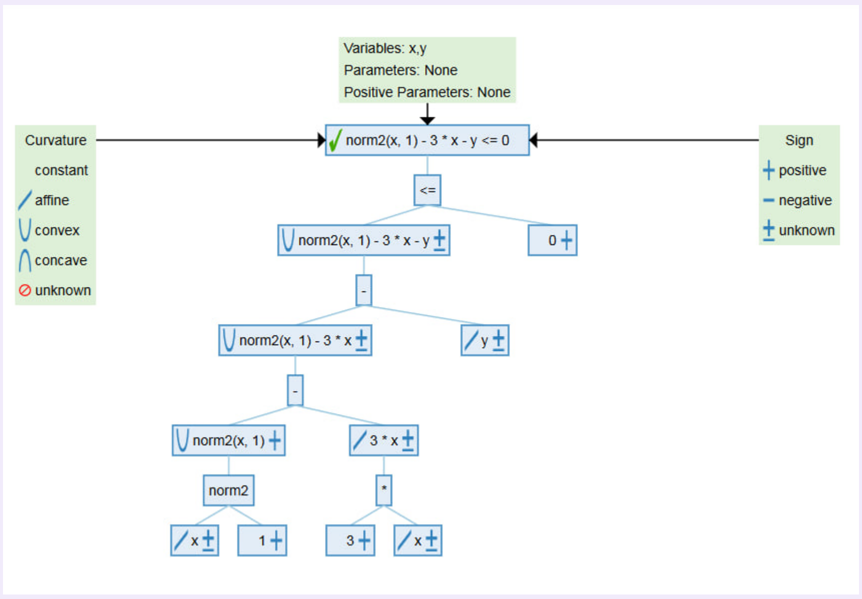

Figure 2: $\sqrt{x^2+1} \leq 3x+y$

(c)

Original:

$$\frac{1}{x} + \frac{2}{y} \leq 5, \quad x > 0, \quad y > 0$$

Convexity Analysis: Both $\frac{1}{x}$ and $\frac{2}{y}$ are convex for $x > 0, y > 0$. The sum of convex functions is convex, making the left-hand side convex.

DCP Expression:

$$\frac{1}{x} + \frac{2}{y} \leq 5$$

Explanation: This expression is already DCP-compliant. Both $\frac{1}{x}$ and $\frac{2}{y}$ are valid DCP functions under the positivity constraint.

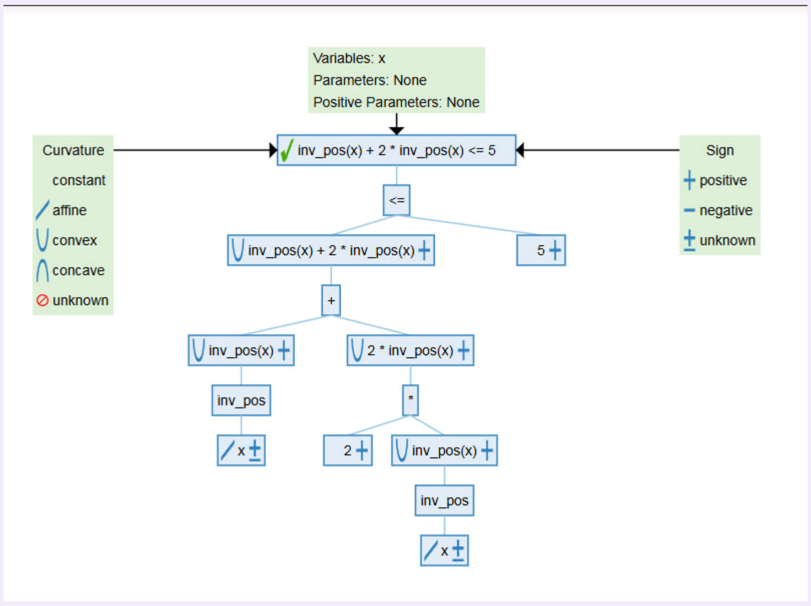

Figure 3: $\frac{1}{x} + \frac{2}{y} \leq 5$

(d)

Original:

$$(x+z)y \geq 1, \quad x+z \geq 0, \quad y \geq 0$$

Convexity Analysis: The function $(x+z)y$ is bilinear, and the inequality $(x+z)y \geq 1$ implies a non-convex region.

DCP Expression:

$$\frac{1}{y} \leq x+z, \quad x+z \geq 0, \quad y > 0$$

Explanation: Rewriting $(x+z)y \geq 1$ as $\frac{1}{y} \leq x+z$ is equivalent. This form is DCP-compatible because $\frac{1}{y}$ is convex when $y > 0$, and adding affine terms preserves compatibility.

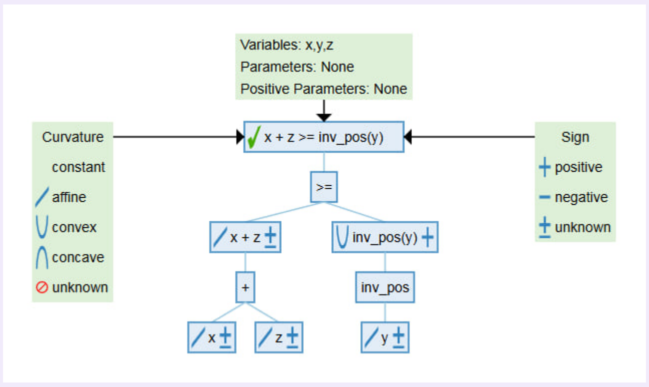

Figure 4: $(z+x)y \geq 1$

(e)

Original:

$$x\sqrt{y} \geq 1, \quad x \geq 0, \quad y \geq 0$$

Convexity Analysis: The product $x\sqrt{y}$ is not convex. Reformulation is required to analyze its convexity.

DCP Expression:

$$\frac{1}{x} \leq \sqrt{y}, \quad x > 0, \quad y \geq 0$$

Explanation: Rewriting $x\sqrt{y} \geq 1$ as $\frac{1}{x} \leq \sqrt{y}$ ensures DCP compatibility. The square root is concave, and the reciprocal is convex for $x > 0$.

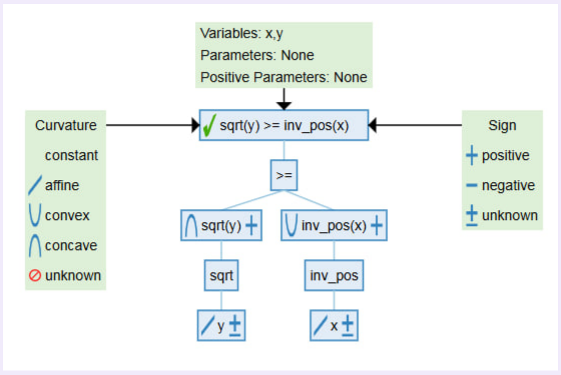

Figure 5: $x\sqrt{y} \geq 1$

(f)

Original:

$$\log\left(e^{y-1} + e^{x/2}\right) \leq -e^{x}$$

Convexity Analysis: The left-hand side, $\log\left(e^{y-1} + e^{x/2}\right)$, is convex. The right-hand side, $-e^{x}$, is concave. Convex functions bounded above by concave functions form a convex set.

DCP Expression:

$$\log\left(e^{y-1} + e^{x/2}\right) + e^{x} \leq 0$$

Explanation: Rewriting preserves equivalence. This form is DCP-compatible as log, exponentials, and affine operations adhere to DCP rules.

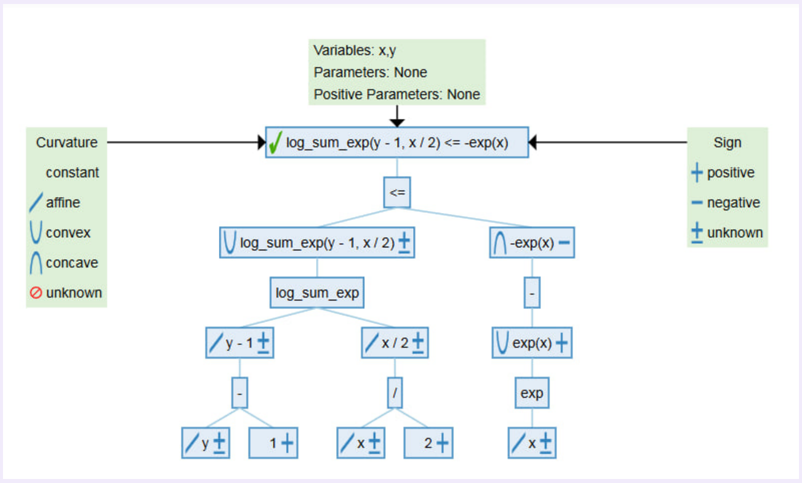

Figure 6: $log(e^{y-1} + e^{\frac{x}{2}} \leq -e^{x})$

(g)

Original:

$$\frac{y^2}{x+z} \leq 1$$

Convexity Analysis: The function $\frac{y^2}{x+z}$ is convex for $x + z > 0$, as the numerator y^2 is convex and the denominator $x + z$ is affine.

DCP Expression:

$$y^2 \leq x + z$$

Explanation: This form is equivalent and DCP-compatible. Multiplication by an affine positive function preserves convexity.

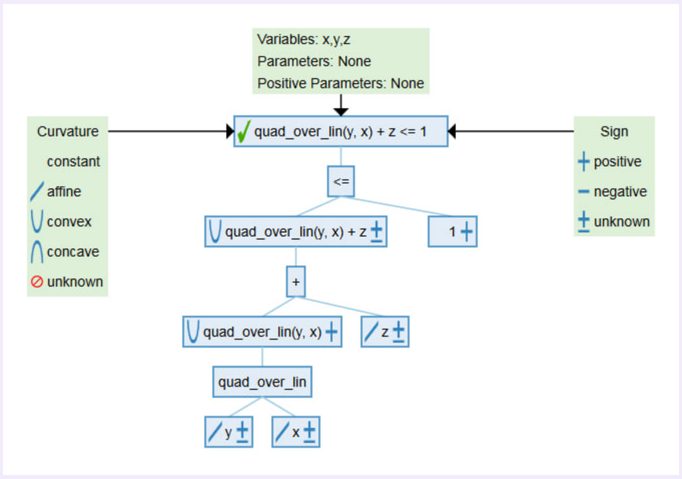

Figure 7: $\frac{y^2}{x+z} \leq 1$

(h)

Original:

$$x \log\left(\frac{x}{y}\right) \leq x - y, \quad x > 0, \quad y > 0$$

Convexity Analysis: The term $x\log(x/y)$ is convex for $x, y > 0$, and the right-hand side $x - y$ is affine. The inequality defines a convex set.

DCP Expression:

$$x\log(x) - x\log(y) \leq x - y$$

Explanation: Expanding $\log(x/y)$ into $\log(x) - \log(y)$ and distributing x preserves equivalence. This expression is DCP-compatible.

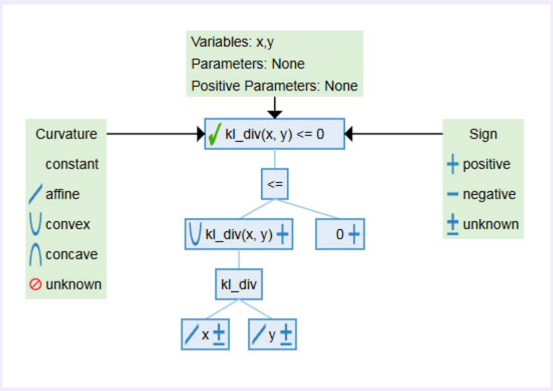

Figure 8: $xlog(\frac{x}{y}) - x + y \leq 0$

part 2

Solution:

Q2. Lasso

```python
import csv
import numpy as np
import cvxpy as cp
from PIL import Image
import matplotlib.pyplot as plt

# Function to read the CSV image file
def read_image(filename):
    with open(filename, newline='') as f:
        reader = csv.reader(f)
        image = np.array([[float(pixel) for pixel in row] for row in reader])
    return image

# Function to perform LASSO regularization using convex optimization
```

```python
def lasso(Y, lam=1.0, p=1):
    n, m = Y.shape
     = cp.Variable((n, m))

    # Objective function
    obj = 0.5 * cp.sum_squares(cp.vec(Y - ))
    for i in range(n):
        for j in range(m):
            vec0 = cp.abs([i, j] - [i, j+1]) if j < m - 1 else 0
            vec1 = cp.abs([i, j] - [i+1, j]) if i < n - 1 else 0
            obj += lam * cp.norm(cp.vstack([vec0, vec1]), p)

    # Solve the problem
    prob = cp.Problem(cp.Minimize(obj))
    prob.solve(solver=cp.SCS, verbose=False)
    return prob.value, .value

# Display function for grayscale images
def display_grayscale_image(img_array):
    img = Image.fromarray((img_array * 255).astype(np.uint8))
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    plt.show()

toy = read_image("toy.csv")
display_grayscale_image(toy)

# LASSO with p=1
opt1, img1 = lasso(toy, 1, 1)
print("Optimal value for p=1:", opt1)
display_grayscale_image(img1)

# LASSO with p=2
opt2, img2 = lasso(toy, 1, 2)
print("Optimal value for p=2:", opt2)
display_grayscale_image(img2)


# Running LASSO on the "baboon" image with multiple lambda and p values
baboon = read_image("baboon.csv")
display_grayscale_image(baboon)

results = []
for p in [1, 2]:
    for lam_exp in range(9):
        lam = 10 ** (-lam_exp / 4)
        opt, img = lasso(baboon, lam, p)
        results.append((opt, img))
        print(f"For =10^(-{lam_exp}/4), p={p}, the optimal value is {opt}")
        display_grayscale_image(img)
```

Listing 1: Python example

0.0.1 part1 and part2

Solution:

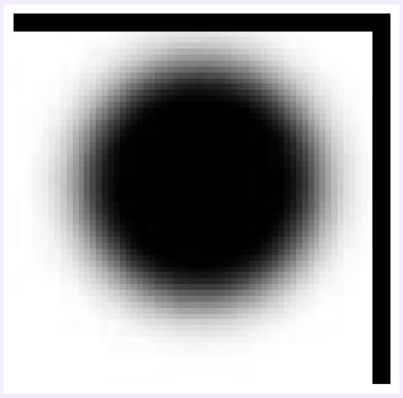

Figure 9: original image

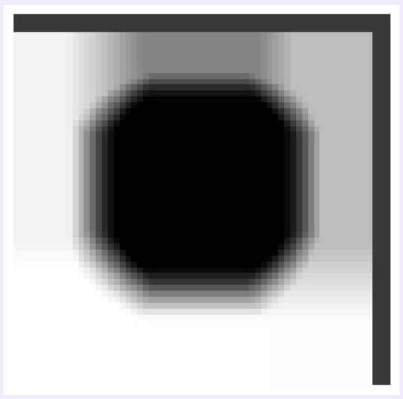

Figure 10: Lasso 1-p

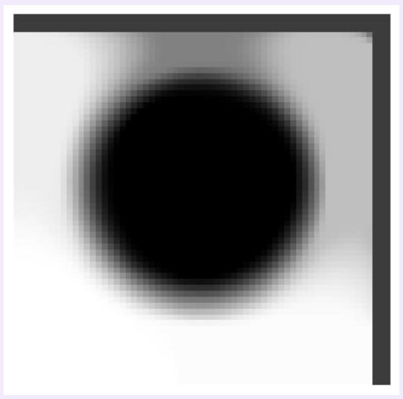

Figure 11: Lasso 2-p

Optimal values

Optimal value for p=1: 199.77117641436223

Optimal value for p=2: 182.20674924329853

1. **Explanation of Shape Change**
   The shape of the solution changes due to the total variation penalty, which encourages neighboring pixel values to be similar. This regularization reduces noise by smoothing out variations, creating a cleaner but often slightly distorted image where fine details may be lost. This is because the penalty specifically discourages sharp changes between neighboring pixel values, thereby smoothing the overall shape.

2. **Problem Category**
   This problem is a Quadratic Programming (QP) problem with an additional regularization term. The least-squares term is quadratic, while the total variation term is a sum of absolute values, making it a convex optimization problem. CVX and CVXPY can handle this QP because the sum of a quadratic function and an absolute value (convex) term results in a convex optimization problem that can be efficiently solved.

1. **Anisotropic vs. Isotropic Penalty**:

   - The **anisotropic** penalty (using the ℓ_1-norm) encourages sparse differences, leading to sharper edges and piecewise-constant regions, which often preserves edges but results in blocky artifacts.
   - The **isotropic** penalty (using the ℓ_2-norm) encourages smooth transitions between pixels. This results in a smoother output, especially across edges, but with less blockiness. This penalty tends to blur edges more compared to the anisotropic form.

2. **Problem Type**:

   - This is a **Quadratic Program (QP)** because it involves a quadratic objective (squared error term) with an additional regularization term based on linear expressions. The use of ℓ_2-norms makes it more complex than a standard linear program (LP).

0.0.2 part 3

Solution:

The results are in ipynb file
1. Change in Histograms with Varying λ:

- As λ increases, the regularization term becomes more dominant, which encourages stronger smoothing.
- For both isotropic and anisotropic penalties, higher λ values lead to histograms with less variation, indicating a smoothing effect across the pixel intensity range.
- With an isotropic penalty (using the ℓ_2-norm), the histogram typically shows a broader distribution of intensities, reflecting a smoother, more natural gradient between pixels.
- With an anisotropic penalty (using the ℓ_1-norm), the histogram may display sharper peaks, indicating piecewise-constant regions where pixel values cluster around certain intensities.

2. Isotropic vs. Anisotropic Differences:

- The isotropic penalty smooths transitions more naturally, often leading to a more blurred appearance but a more gradual change in pixel intensities.
- The anisotropic penalty preserves edges and encourages a piecewise-constant solution, resulting in sharper transitions and blocky effects where neighboring pixels have similar intensities. This can be seen in the histogram as distinct peaks.

Q3. Group Testing

part 1

Solution:

```python
import csv
import numpy as np
import cvxpy as cp
from PIL import Image
import matplotlib.pyplot as plt

# Function to read the CSV image file
def read_image(filename):
    with open(filename, newline='') as f:
        reader = csv.reader(f)
        image = np.array([[float(pixel) for pixel in row] for row in reader])
    return image

# Function to perform LASSO regularization using convex optimization
def lasso(Y, lam=1.0, p=1):
    n, m = Y.shape
     = cp.Variable((n, m))

    # Objective function
    obj = 0.5 * cp.sum_squares(cp.vec(Y - ))
    for i in range(n):
        for j in range(m):
            vec0 = cp.abs([i, j] - [i, j+1]) if j < m - 1 else 0
            vec1 = cp.abs([i, j] - [i+1, j]) if i < n - 1 else 0
            obj += lam * cp.norm(cp.vstack([vec0, vec1]), p)

    # Solve the problem
    prob = cp.Problem(cp.Minimize(obj))
    prob.solve(solver=cp.SCS, verbose=False)
    return prob.value, .value

# Display function for grayscale images
def display_grayscale_image(img_array):
    img = Image.fromarray((img_array * 255).astype(np.uint8))
    plt.imshow(img, cmap='gray')
    plt.axis('off')
    plt.show()
```

```python

toy = read_image("toy.csv")
display_grayscale_image(toy)

# LASSO with p=1
opt1, img1 = lasso(toy, 1, 1)
print("Optimal value for p=1:", opt1)
display_grayscale_image(img1)

# LASSO with p=2
opt2, img2 = lasso(toy, 1, 2)
print("Optimal value for p=2:", opt2)
display_grayscale_image(img2)


# Running LASSO on the "baboon" image with multiple lambda and p values
baboon = read_image("baboon.csv")
display_grayscale_image(baboon)

results = []
for p in [1, 2]:
    for lam_exp in range(9):
        lam = 10 ** (-lam_exp / 4)
        opt, img = lasso(baboon, lam, p)
        results.append((opt, img))
        print(f"For =10^(-{lam_exp}/4), p={p}, the optimal value is {opt}")
        display_grayscale_image(img)
```

Listing 2: Python example

so the result is:
Optimization status: optimal
Identified infected individuals: [74 174 183 302 427 446 735 805 860 985]
True infected individuals: [74 174 183 302 427 446 735 805 860 985]
Does the algorithm identify all infected individuals correctly? True

part 2

Solution:

```python
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt

N = 1000
S = 10
M = 100

ind0 = np.random.choice(N, S, replace=False)
x0 = np.zeros(N)
x0[ind0] = np.random.rand(S)

success_rates = []
K_values = range(2, 21)

for K in K_values:

    A = np.zeros((M, N))
    for i in np.arange(N):
        ind = np.random.choice(M, K, replace=False)
        A[ind, i] = 1

    y = A @ x0

    x = cp.Variable(N, nonneg=True)
    objective = cp.Minimize(cp.norm1(x))
    constraints = [A @ x == y]
    problem = cp.Problem(objective, constraints)
    problem.solve()

    identified = np.where(x.value > 1e-4)[0]
    success = np.array_equal(np.sort(identified), np.sort(ind0))
    success_rates.append(success)

plt.figure(figsize=(10, 6))
plt.plot(K_values, success_rates, marker='o')
plt.xlabel('Number of Splits per Sample (K)')
plt.ylabel('Success Rate')
plt.title('Effect of K on Group Testing Success Rate')
plt.grid(True)
plt.show()
```

Listing 3: Python example

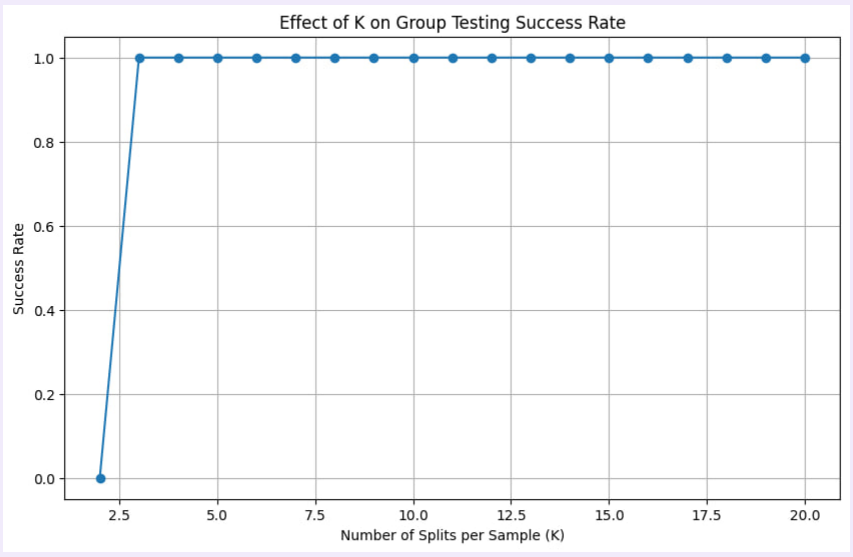

Figure 12: result of part 2 Q3

As we see in 13 For $K \geq 3$ we have valid result and for $K < 3$ not.

part 3

Solution:

```python
import numpy as np
import cvxpy as cp
import matplotlib.pyplot as plt

N = 1000
M = 100
tolerance = 1e-4

prevalence_levels = np.arange(0.01, 0.20, 0.01)

success_rates = []
optimal_K = []

for prevalence in prevalence_levels:
    S = int(N * prevalence)
    ind0 = np.random.choice(N, S, replace=False)
    x0 = np.zeros(N)
    x0[ind0] = np.random.rand(S)

    best_K = None
    success_for_prevalence = False

    for K in range(1, M + 1):
        A = np.zeros((M, N))
        for i in np.arange(N):
            ind = np.random.choice(M, K, replace=False)
            A[ind, i] = 1

        y = A @ x0

        x = cp.Variable(N, nonneg=True)
        objective = cp.Minimize(cp.norm1(x))
        constraints = [A @ x == y]
        problem = cp.Problem(objective, constraints)
        problem.solve()

        identified = np.where(x.value > tolerance)[0]
        success = np.array_equal(np.sort(identified), np.sort(ind0))

        if success:
            best_K = K
            success_for_prevalence = True
            break

    optimal_K.append(best_K if success_for_prevalence else None)
    success_rates.append(success_for_prevalence)


plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
plt.plot(prevalence_levels * 100, success_rates, marker='o', color='blue')
plt.xlabel('Prevalence (%)')
plt.ylabel('Success Rate')
plt.title('Success Rate vs. Prevalence')
plt.grid(True)

plt.subplot(1, 2, 2)
plt.plot(prevalence_levels * 100, optimal_K, marker='o', color='green')
plt.xlabel('Prevalence (%)')
plt.ylabel('Optimal K')
plt.title('Optimal K vs. Prevalence')
plt.grid(True)

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

Listing 4: Python example

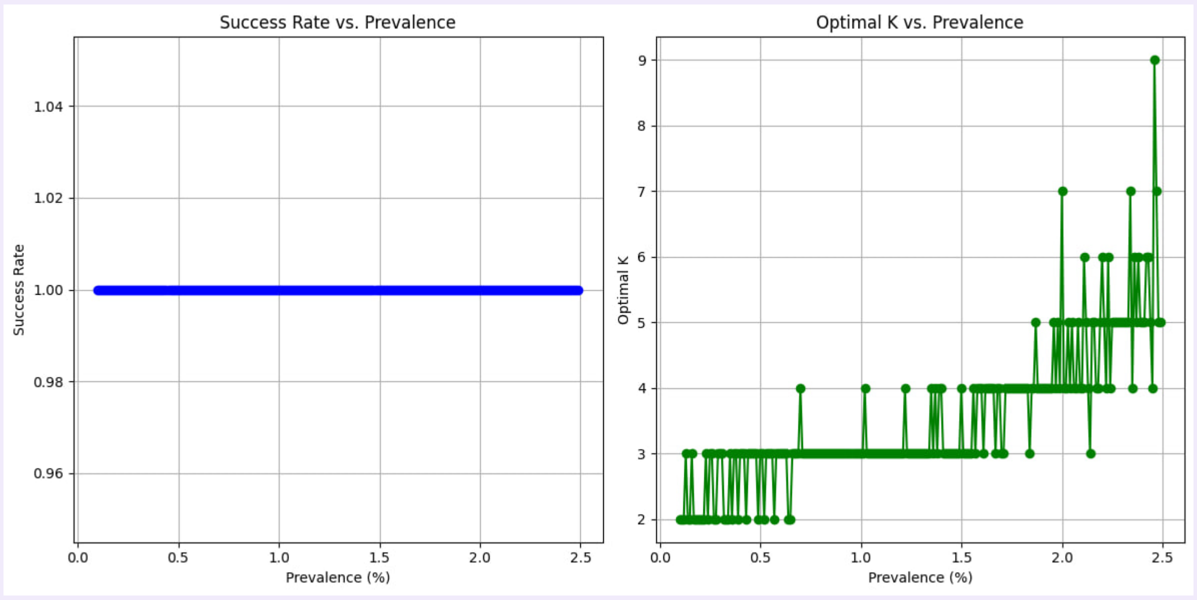

Figure 13: result of part 3 Q3

Q4. Learning Quadratic Metrics from Distance

part 1

Solution:

We aim to determine a matrix $P \in S_n^+$ (the cone of symmetric positive semidefinite matrices) such that the quadratic pseudo-metric

$$d(x, y) = \sqrt{(x-y)^\top P(x-y)}$$

best approximates a set of given distances d_i. The optimization problem is:

$$\min_{P \in S_n^+} \quad \frac{1}{N} \sum_{i=1}^{N} \left(d_i - \sqrt{(x_i - y_i)^\top P(x_i - y_i)} \right)^2 .$$

1. Challenges and Non-Convexity The problem is non-convex due to the presence of the square root and the quadratic term within the objective function. However, there are techniques to address such problems approximately or transform them into forms amenable to convex optimization.

2. Reformulation via Variable Substitution Introduce auxiliary variables $z_i = \sqrt{(x_i - y_i)^\top P(x_i - y_i)}$. The problem can be rewritten as:

$$\min_{P \in S_n^+, z_i} \quad \frac{1}{N} \sum_{i=1}^{N} (d_i - z_i)^2,$$

subject to

$$z_i^2 = (x_i - y_i)^\top P(x_i - y_i), \quad i = 1, \dots, N.$$

3. Relaxation into a Convex Problem To make the optimization convex, we relax the equality constraint $z_i^2 = (x_i - y_i)^\top P(x_i - y_i)$ into a second-order cone constraint:

$$z_i \geq \sqrt{(x_i - y_i)^\top P(x_i - y_i)}, \quad i = 1, \dots, N.$$

This leads to the convex optimization problem:

$$\min_{P \in S_n^+, z_i} \quad \frac{1}{N} \sum_{i=1}^{N} (d_i - z_i)^2,$$

subject to:

$$z_i^2 \geq (x_i - y_i)^\top P(x_i - y_i), \quad i = 1, \dots, N.$$

This problem can be solved using convex optimization solvers like CVXPY or MOSEK, which handle semidefinite programming (SDP) and second-order cone programming (SOCP).

4. Alternative Linearization Instead of minimizing the squared residuals directly in terms of z_i, another approach linearizes the square root as follows: Consider minimizing:

$$\min_{P \in S_n^+} \quad \sum_{i=1}^{N} \left(d_i^2 - 2d_i \sqrt{(x_i - y_i)^\top P(x_i - y_i)} + (x_i - y_i)^\top P(x_i - y_i) \right).$$

This can be approached by iterative methods, alternating between fixing P and updating the z_i variables or using approximation methods such as gradient descent over P with appropriate regularization.

5. Convex Form Approximation In practice, relaxations that drop the square root constraint often yield:

$$\min_{P \in S_n^+} \sum_{i=1}^{N} \left(d_i^2 - 2d_i \|x_i - y_i\|_P + \|x_i - y_i\|_P^2 \right),$$

where $\|x\|_P = \sqrt{x^\top P x}$, which is convex if $P \in S_n^+$.

part2

Solution:

```python
data = np.load('LQM_data.npz')
X_train = data['X_train']
Y_train = data['Y_train']
d_train = data['d_train']


n = X_train.shape[0]
P = cp.Variable((n, n), symmetric=True)

objective_terms = []
objective_sum = 0
for i in range(len(d_train)):
    xi = X_train[:, i]
    yi = Y_train[:, i]
    di = d_train[i]

    objective_sum = objective_sum + cp.power(cp.quad_form(xi - yi, P)- cp.power(
    di, 2), 2)

objective = cp.Minimize(objective_sum)
constraints = [P >> 0]
problem = cp.Problem(objective, constraints)
problem.solve()

P_opt = P.value


X_test = data['X_test']
Y_test = data['Y_test']
d_test = data['d_test']

len(d_test)
MSE = 0
for i in range(len(d_test)):
    xi = X_test[:, i]
    yi = Y_test[:, i]
    di = d_test[i]
    diff = xi - yi
    predicted_distance = (diff.T @ P_opt @ diff)**(1/2)
    MSE += (predicted_distance - di)**2/len(d_test)

print("Optimal P: ",P_opt)
print(f"Mean Squared Error on Test Data: {MSE}")
```

Listing 5: Python example

Q5. Fitting a sphere to data

part 1

Solution:

We can solve this problem using convex optimization. The objective is to minimize the error function

$$\sum_{i=1}^{m} \left(\|u_i - x_c\|_2^2 - r^2 \right)^2$$

where $x_c \in \mathbb{R}^n$ is the center of the sphere, and $r \in \mathbb{R}$ is its radius.

Step 1: Reformulate the Error Function

Lets expand the squared error term:

$$\left(\|u_i - x_c\|_2^2 - r^2 \right)^2$$

Define:

$$d_i = \|u_i - x_c\|_2^2$$

Thus, the error function becomes:

$$\sum_{i=1}^{m} (d_i - r^2)^2$$

Step 2: Introduce New Variables

Introduce new variables t_i as:

$$t_i = d_i - r^2 = \|u_i - x_c\|_2^2 - r^2, \quad \forall i = 1, \ldots, m$$

Thus, the objective function becomes:

$$\sum_{i=1}^{m} t_i^2$$

Step 3: Convex Relaxation (SOCP Formulation)

To make the problem more tractable, we introduce auxiliary variables $s \geq 0$ representing the squared error:

$$\text{minimize} \quad s$$

subject to:

$$t_i^2 \leq s, \quad \forall i = 1, \ldots, m$$

$$t_i = \|u_i - x_c\|_2^2 - r^2$$

This formulation minimizes the maximum squared error, a relaxation of the sum-of-squares form. The constraints are now second-order cone (SOC) constraints, making this an **SOCP** (Second-Order Cone Program), which is convex.

Step 4: Final Formulation

The optimization problem is:

$$\text{minimize} \quad s$$

subject to:

$$\|u_i - x_c\|_2^2 - r^2 \leq \sqrt{s}, \quad \|u_i - x_c\|_2^2 - r^2 \geq -\sqrt{s}, \quad \forall i$$

Conclusion

This SOCP formulation is convex and can be efficiently solved using standard convex optimization solvers. By minimizing s, we minimize the maximum squared deviation from the desired sphere fit.

part 2

Solution:

```python
import numpy as np
import cvxpy as cvx

file_path = 'Q5.npz'
data = np.load(file_path)
x_coords, y_coords = data['U'][0], data['U'][1]

print("X Coordinates:", x_coords)
print("Y Coordinates:", y_coords)

cx = cvx.Variable()
cy = cvx.Variable()

obj = cvx.Minimize(cvx.sum(cvx.norm(cvx.vstack((x_coords - cx, y_coords - cy)),
    axis=0)))

prob = cvx.Problem(obj)
prob.solve()

xc = np.array([cx.value, cy.value])
print(f"Optimal center: {xc}")

distances = np.sqrt((x_coords - xc[0])**2 + (y_coords - xc[1])**2)
rms_distance = np.sqrt(np.mean(distances**2))
print(f"Root Mean Squared Distance (RMSD): {rms_distance}")
```

Listing 6: Python example

```
X Coordinates: [-3.8355737 -3.2269177 -1.6572955 -2.8202585 -1.7831869 -2.1605783
-2.0960803 -1.3866295 -3.2077849 -2.0095986 -2.0965432 -2.8128775
-3.6501826 -2.1638414 -1.727471 -1.574323 -1.3761806 -1.3602495
-1.5257654 -1.9231176 -2.9296195 -3.282827 -2.9078414 -3.5423007
-3.1388035 -1.7957226 -2.6267585 -3.6652627 -3.7394118 -3.7898021
-3.6200108 -3.0386294 -2.0320023 -2.9577808 -2.9146706 -3.2243786
-2.1781976 -2.254515 -1.2559218 -1.8875105 -3.6122685 -2.6552417
-1.412756 -3.7475311 -2.1367633 -3.9263527 -2.3118969 -1.4249518
-2.0196394 -1.4021445]
```

```
Y Coordinates: [5.906125 7.5112709 7.470473 7.737812 5.4818448 7.723145 7.7072529
6.1452654 7.6023307 7.6382459 5.242151 5.1622157 7.25855 7.6899057
5.4564872 7.3510769 6.9730981 6.9056362 5.7518622 7.677503 7.7561481
5.4188036 5.1741322 5.5660735 7.7008514 5.4273243 7.7336173 7.2686635
6.0293335 5.9057623 5.7754097 5.3028798 5.2594588 5.3040353 7.7731243
5.4402982 7.7681141 5.2233652 6.2741755 5.4133273 7.2743342 7.7564498
6.0732284 7.2351834 7.6955709 6.2241593 7.7636052 7.1457752 5.3154475
5.9675466]
Optimal center: [-2.47364933 6.68612404]
Root Mean Squared Distance (RMSD): 1.3070721865919228
```

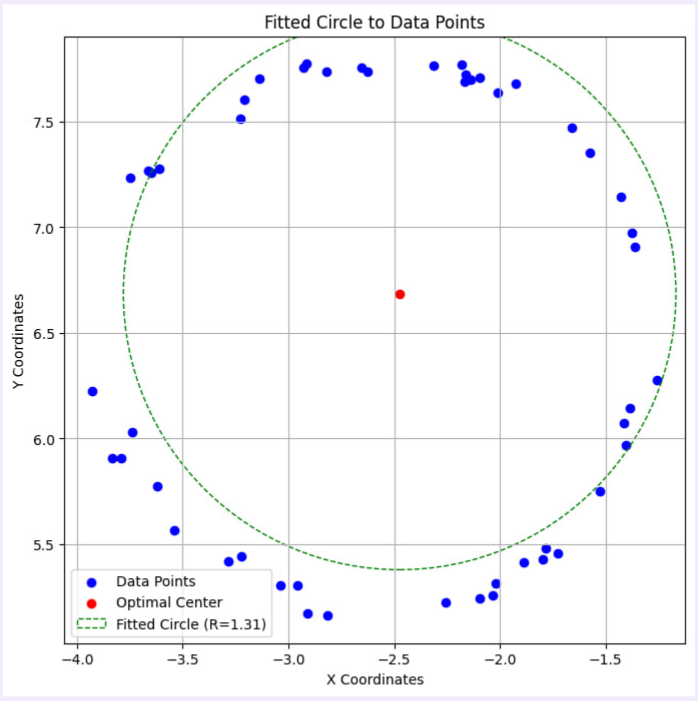

Figure 14: result of part 2 Q5

Q6. Minimax Rational Fit to the Exponential

Solution:

```python
import numpy as np
import cvxpy as cvx

k = 201
t = np.linspace(-3, 3, k)
y = np.exp(t)
t_pow = np.vstack([np.ones_like(t), t, t**2])

upper_bound = np.exp(3)
```

```python
lower_bound = 0
tolerance = 1e-3
max_iterations = 1000

a = cvx.Variable(3)
b = cvx.Variable(2)
gamma = cvx.Parameter()

def create_problem(gamma_value):
    gamma.value = gamma_value
    b_full = cvx.hstack([1, b])
    constraints = cvx.abs(a @ t_pow - cvx.multiply(y, b_full @ t_pow)) <= gamma
    * (b_full @ t_pow)
    objective = cvx.Minimize(gamma)
    return cvx.Problem(objective, [constraints])

def check_solution_bounds(upper_bound, lower_bound):
    create_problem(upper_bound).solve()
    if gamma.value is None or gamma.value <= 0:
        raise ValueError("Upper bound too small; no feasible solution.")

    create_problem(lower_bound).solve()
    if gamma.value is not None:
        raise ValueError("Lower bound too large; feasible solution exists.")

try:
    check_solution_bounds(upper_bound, lower_bound)
    u, l = upper_bound, lower_bound
    best_gamma, best_a, best_b = None, None, None

    for iteration in range(max_iterations):
        mid_gamma = (u + l) / 2
        prob = create_problem(mid_gamma)
        try:
            prob.solve()
        except cvx.SolverError:
            prob.solve(solver=cvx.SCS)

        print(f"Iteration: {iteration}, Status: {prob.status}, Upper: {u:.3f},
    Lower: {l:.3f}, Gamma: {mid_gamma:.3f}")

        if prob.status == 'optimal':
            u = mid_gamma
            best_a = a.value
            best_b = np.hstack([1, b.value])
            best_gamma = gamma.value[0]
        else:
            l = mid_gamma

        if u - l <= tolerance:
            print("Tolerance met. Converged.")
            break
    else:
        print("Maximum iterations reached. Solution may not have converged.")

    if best_gamma is not None:
        print("\nOptimal Solution Found:")
        print(f"Optimal Gamma: {best_gamma:.6f}")
        print(f"Optimal a: {best_a}")
        print(f"Optimal b: {best_b}")
    else:
        print("\nNo feasible solution found.")

except ValueError as e:
    print(e)
print("\nOptimal Solution Found:")
print(f"Optimal Gamma: {best_gamma:.6f}")
print(f"Optimal a0 , a1 , a2: {best_a}")
print(f"Optimal b0 = 1 , b1 , b2: {best_b}")
```

Listing 7: Python example

Lower bound too large; feasible solution exists.

```
Optimal Solution Found:
Optimal Gamma: 0.023293
Optimal a0 , a1 , a2: [1.00976773 0.61190828 0.11350632]
Optimal b0 = 1 , b1 , b2: [ 1. -0.41457934 0.04849808]
```