Math 564 - Project 03

Sajjad Uddin Mahmud

Nov 20, 2025

1 Task 1: Convexity, Coercivity, and Boundary Solutions

Part 1: Convexity

The objective function is

$$f(z) = \frac{\beta}{2} \sum_{k=0}^{4} \|z_{2k} - z_{2k+1}\|^2 + \frac{\alpha}{2} \sum_{k=1}^{4} \|z_{2k-1} - z_{2k}\|^2$$

To verify convexity, we compute the Hessian matrix. Each squared norm term $\|z_i - z_j\|^2$ can be written as:

$$\|z_i - z_j\|^2 = (z_i - z_j)^T (z_i - z_j) = z_i^T z_i - 2 z_i^T z_j + z_j^T z_j$$

This is a quadratic function in the decision variables $z_1, \ldots, z_8$. The Hessian of each term $\|z_i - z_j\|^2$ with respect to the vector of all variables is positive semidefinite, as it has the block structure corresponding to $(z_i - z_j)(z_i - z_j)^T$, which is a rank-one positive semidefinite matrix.

Since $\alpha > 0$ and $\beta > 0$, the objective function $f(z)$ is a positive weighted sum of convex quadratic functions. Therefore, $f(z)$ is convex.

Part 2: Coercivity

A function is coercive if $f(z) \to \infty$ as $\|z\| \to \infty$. Consider the objective function structure: it represents a chain of connected segments from the fixed point z_0 through $z_1, \ldots, z_8$ to the fixed point z_9.

If any variable z_k (for $k = 1, \ldots, 8$) moves to infinity, at least one of the squared distances involving z_k must also go to infinity. Specifically:

- If $z_1 \to \infty$, then $\|z_0 - z_1\|^2 \to \infty$ (since z_0 is fixed)
- If $z_k \to \infty$ for $k \in \{2, \ldots, 7\}$, then either $\|z_{k-1} - z_k\|^2 \to \infty$ or $\|z_k - z_{k+1}\|^2 \to \infty$
- If $z_8 \to \infty$, then $\|z_8 - z_9\|^2 \to \infty$ (since z_9 is fixed)

Since all coefficients $\alpha, \beta > 0$, we have $f(z) \to \infty$ as $\|z\| \to \infty$. Therefore, $f(z)$ is coercive.

Part 3: Optimal Points on Boundary

The optimal locations of $z_1, \ldots, z_8$ lie on the boundary of their respective polyhedra.

We are minimizing total squared distance. If a point z_k is inside its polygon (not touching any edge), it has room to move in any direction. We can always move it slightly closer to its two neighbors, reducing the total distance. The only reason a point would stop moving closer is if it hits the boundary of the polygon and cannot move any further. Therefore, the optimal solution has all points on the boundary.

2 Task 4: Optimal Solutions for Various α Values

We solved the transmission line optimization problem for various values of α while keeping $\beta = 1.0$ fixed. The fixed endpoints are $z_0 = (4, 5)$ and $z_9 = (26, 5)$.

Table 2.1 summarizes the objective function values and α/β ratios for all test cases.

Table 2.1: Optimization results for various α values with $\beta = 1.0$

α	β	α/β	Objective Value
1.0	1.0	1.0	29.701
1.5	1.0	1.5	37.520
2.0	1.0	2.0	44.853
3.0	1.0	3.0	59.195
5.0	1.0	5.0	87.543
10.0	1.0	10.0	157.371
20.0	1.0	20.0	295.839

The optimal coordinates for the transfer stations $z_1, \ldots, z_8$ are presented in Table 2.2.

Table 2.2: Optimal transfer station coordinates for different α values

α	z_1	z_2	z_3	z_4	z_5	z_6	z_7	z_8
1.0	(6.88, 5.10)	(10.17, 5.08)	(12.91, 5.16)	(15.66, 5.23)	(18.20, 5.24)	(21.66, 5.24)	(22.58, 5.27)	(24.29, 5.14)
1.5	(6.99, 5.47)	(10.24, 5.42)	(13.20, 5.69)	(15.50, 5.70)	(18.20, 5.59)	(21.67, 5.52)	(22.62, 5.47)	(23.97, 5.28)
2.0	(7.03, 5.62)	(10.24, 5.43)	(13.20, 5.70)	(15.50, 5.70)	(18.20, 5.60)	(21.67, 5.54)	(22.63, 5.53)	(23.79, 5.37)
3.0	(7.10, 5.85)	(10.25, 5.51)	(13.20, 5.70)	(15.50, 5.70)	(18.20, 5.62)	(21.67, 5.59)	(22.66, 5.64)	(23.70, 5.50)
5.0	(7.21, 6.23)	(10.30, 5.75)	(13.20, 5.70)	(15.50, 5.70)	(18.20, 5.57)	(21.67, 5.55)	(22.66, 5.66)	(23.70, 5.50)
10.0	(7.40, 6.89)	(10.40, 6.30)	(13.20, 5.70)	(15.50, 5.70)	(18.20, 5.46)	(21.67, 5.43)	(22.67, 5.68)	(23.70, 5.50)
20.0	(7.44, 7.02)	(10.40, 6.30)	(13.20, 5.70)	(15.50, 5.70)	(18.20, 5.21)	(21.66, 5.19)	(22.67, 5.69)	(23.70, 5.50)

Figures 1 to 6 show the optimal transmission line layouts for different values of α. As α increases, the inter-community transmission costs become more expensive relative to intra-community costs, causing the path to adjust accordingly.

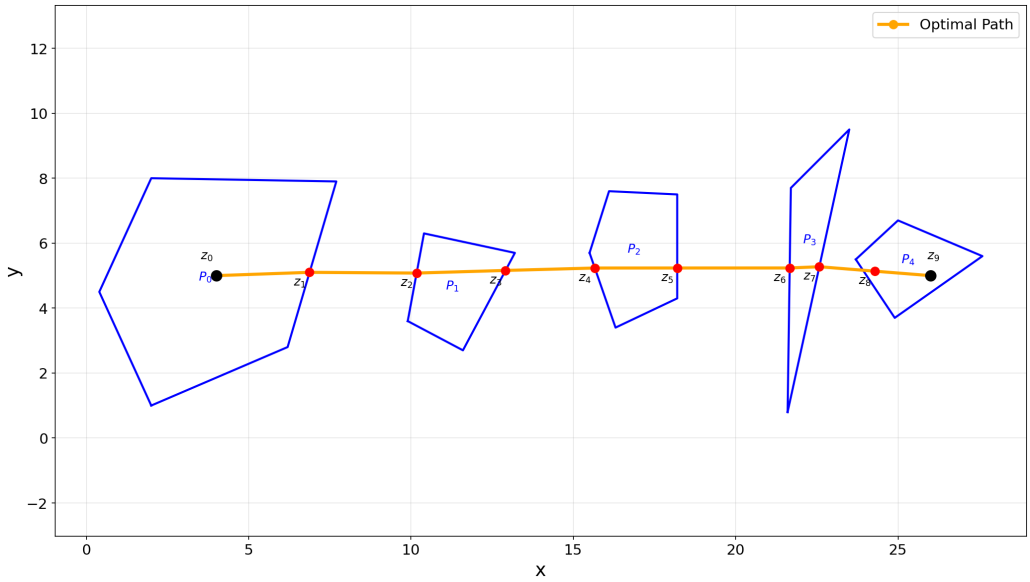

Figure 1: Optimal transmission line layout for $\alpha = 1.0$

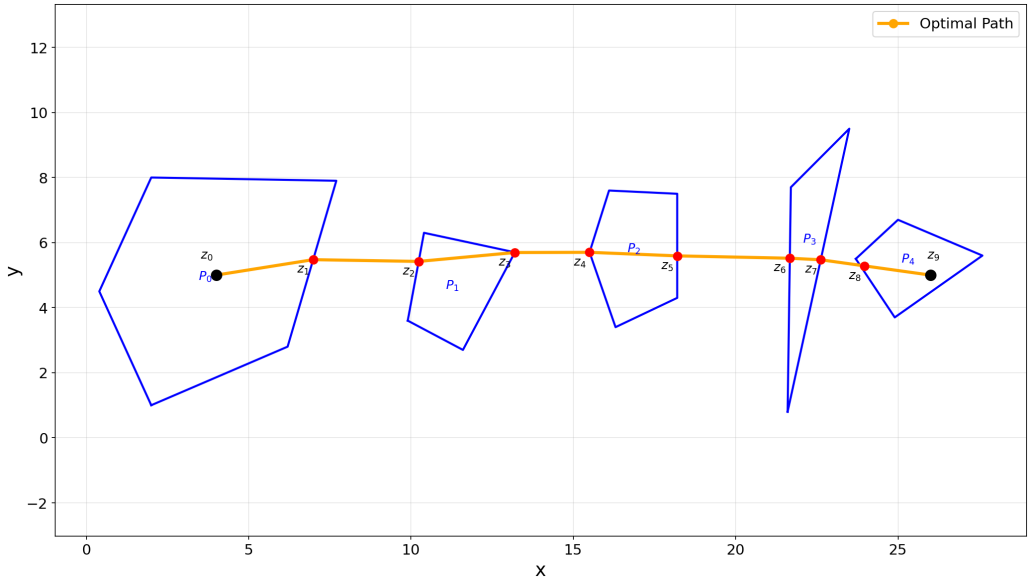

Figure 2: Optimal transmission line layout for $\alpha = 1.5$

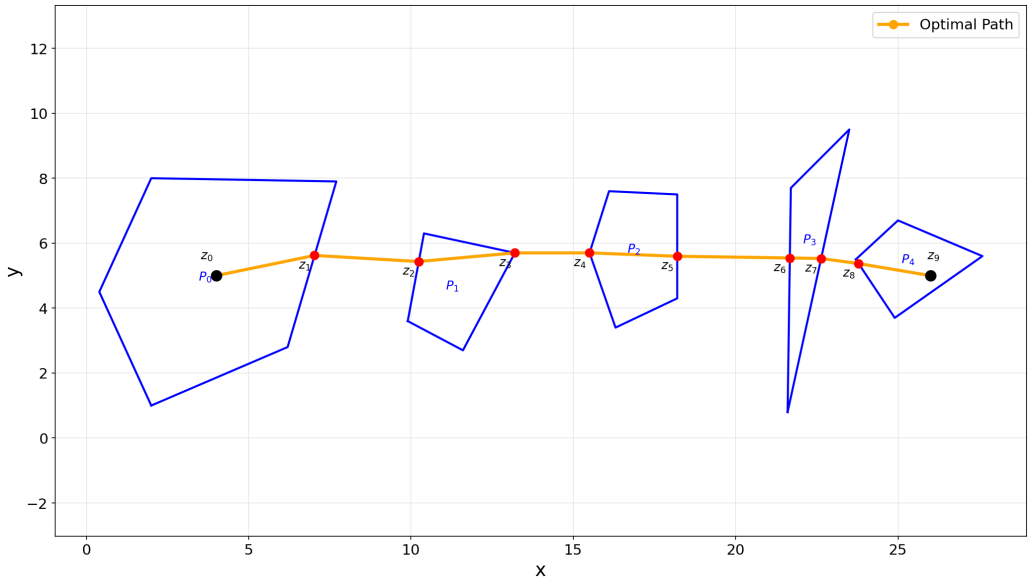

Figure 3: Optimal transmission line layout for $\alpha = 2.0$

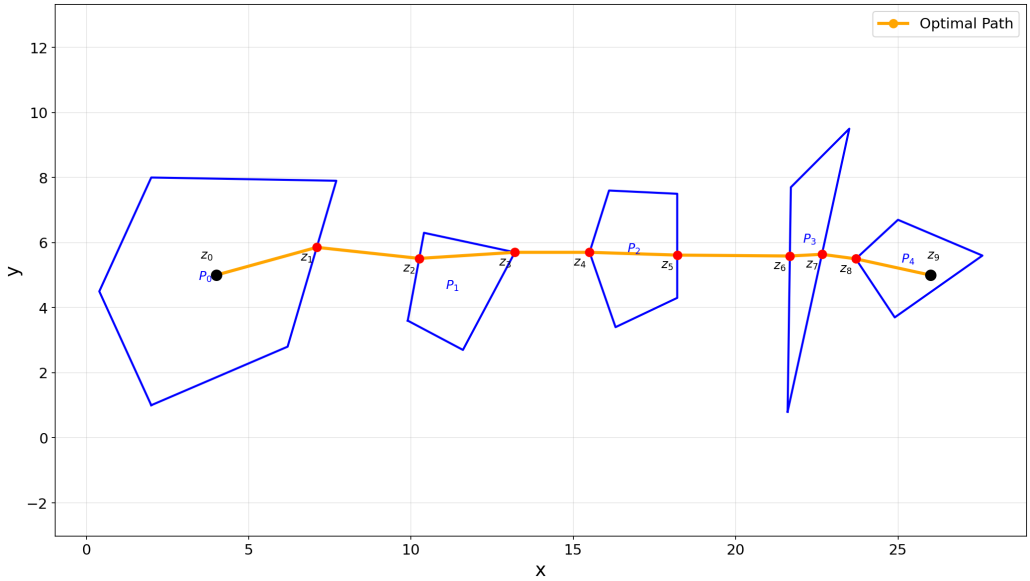

Figure 4: Optimal transmission line layout for $\alpha = 3.0$

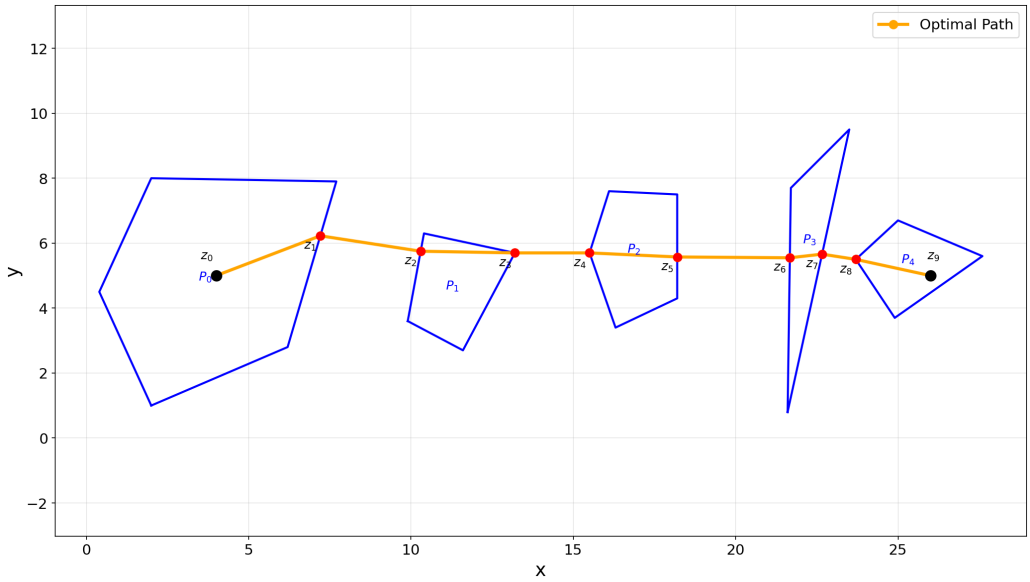

Figure 5: Optimal transmission line layout for $\alpha = 5.0$

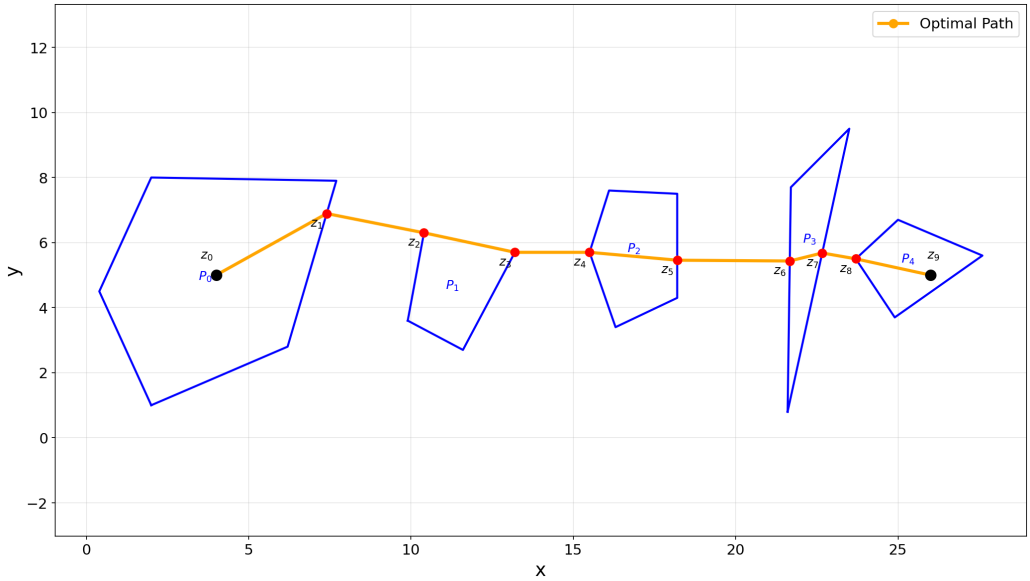

Figure 6: Optimal transmission line layout for $\alpha = 10.0$

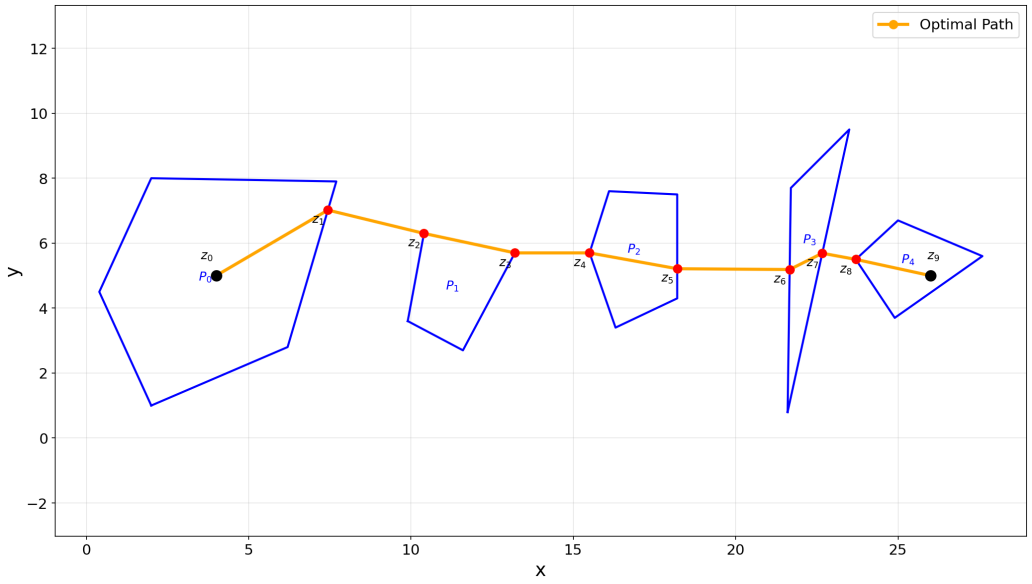

Figure 7: Optimal transmission line layout for $\alpha = 20.0$

The results demonstrate several key observations:

- As α increases, the objective function value increases significantly, from 29.7 at $\alpha = 1.0$ to 295.8 at $\alpha = 20.0$. This reflects the higher penalty for inter-community transmission lines.
- When $\alpha = \beta = 1.0$, the path is relatively straight, staying close to $y = 5$. As α increases, points z_1 and z_2 move upward within their polygons to minimize the expensive inter-community distances, even at the cost of longer intra-community segments. The path topology changes noticeably around $\alpha/\beta \approx 3 - 5$, where the optimizer begins to prioritize minimizing inter-community distances more aggressively.

3 Task 5: Line Segment Constraint

For Task 5, we modified polygon P_2 (community 3) by constraining it to a line segment along edge 0. This tests the solver's ability to handle equality constraints. We used $\alpha = 1.50$ and $\beta = 1.0$.

The optimization with the line segment constraint yielded:

- **Objective Value:** 41.827
- α/β **Ratio:** 1.5

The optimal transfer station coordinates are shown in Table 3.1.

Table 3.1: Optimal coordinates for Task 5 (line segment constraint on P_2)

Point	Coordinates
z_1	(6.8588, 5.0399)
z_2	(10.0989, 4.6741)
z_3	(12.5938, 4.5634)
z_4	(16.3000, 3.4000)
z_5	(18.2000, 4.3000)
z_6	(21.6542, 4.5404)
z_7	(22.5093, 4.9636)
z_8	(24.0041, 5.0439)

Figure 8 shows the optimal solution with the line segment constraint. Points z_4 and z_5, which belong to community P_2, are constrained to lie on the specified edge of the polygon.

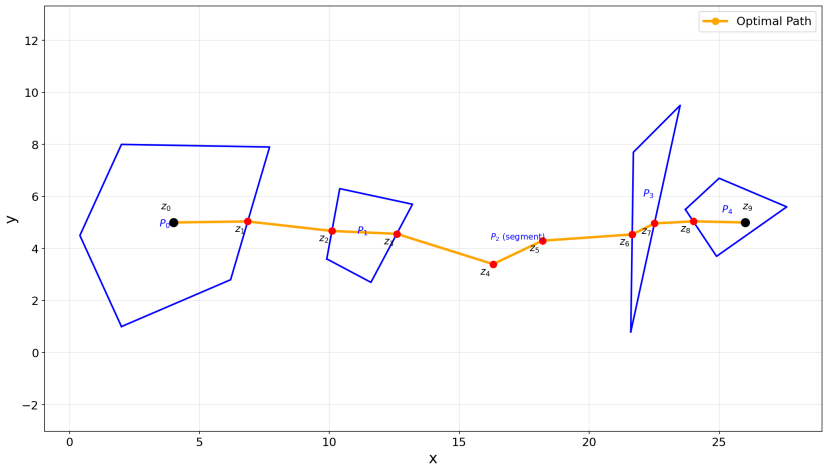

Figure 8: Optimal transmission line layout with P_2 constrained to a line segment ($\alpha = 1.5$, $\beta = 1.0$)

The line segment constraint forces z_4 and z_5 to lie on a single edge of polygon P_2, significantly restricting their placement options. Compared to the unconstrained Task 4 solution with $\alpha = 1.5$ (objective value 37.520), the constrained problem has a higher objective value of 41.827. This increase reflects the cost of the additional constraint.

4 Repository

All code and iteration logs for this project are available in a public GitHub repository:

`https://github.com/sajjad30148/WSU_Math564_Fall2025`

The repository includes the code, and result folders containing iterations from each run.

5 Code

Listing 1: project01_main.py

```python
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import linprog
from scipy.sparse.linalg import cg
import os

#
    ================================================================================
# PROBLEM DEFINITION
#
    ================================================================================

# Fixed endpoints
z0 = np.array([4.0, 5.0])
z9 = np.array([26.0, 5.0])

# Cost coefficients
beta = 1.0
alpha_values = [1.0, 1.5, 2.0, 3.0, 5.0, 10.0, 20.0] # Task 4: various
    alpha values

# Polygon vertices (counter-clockwise order)
P0_vertices = np.array([[2, 1], [6.2, 2.8], [7.7, 7.9], [2, 8], [0.4,
    4.5]])
P1_vertices = np.array([[9.9, 3.6], [11.6, 2.7], [13.2, 5.7], [10.4,
    6.3]])
P2_vertices = np.array([[15.5, 5.7], [16.3, 3.4], [18.2, 4.3], [18.2,
    7.5], [16.1, 7.6]])
P3_vertices = np.array([[21.6, 0.8], [23.5, 9.5], [21.7, 7.7]])
P4_vertices = np.array([[23.7, 5.5], [24.9, 3.7], [27.6, 5.6], [25, 6.7]])

polygons = [P0_vertices, P1_vertices, P2_vertices, P3_vertices,
    P4_vertices]

# Task 5 settings: Convert polygon to line segment
TASK5_ENABLED = True # Set to True to run Task 5
TASK5_POLYGON_IDX = 2 # Which polygon to convert (0-4)
TASK5_EDGE_IDX = 1 # Which edge of that polygon (0 to num_vertices-1)
TASK5_ALPHA = 1.5 # Alpha value for Task 5

```

```python
#
    ================================================================================
# UTILITY FUNCTIONS
#
    ================================================================================

def vertices_to_halfspaces(vertices):
    """
    Convert counter-clockwise vertices to inequality constraints Ax <= b
    For polygon P = conv{(a1,b1),...,(ap,bp)}, compute:
    (bk+1 - bk)*x + (ak - ak+1)*y <= ak*bk+1 - bk*ak+1
    """
    n = len(vertices)
    A_ineq = []
    b_ineq = []

    for k in range(n):
        a_k, b_k = vertices[k]
        a_kp1, b_kp1 = vertices[(k + 1) % n]

        # Inequality: (b_{k+1} - b_k)*x + (a_k - a_{k+1})*y <= a_k*b_{k+1}
            - b_k*a_{k+1}
        A_row = [b_kp1 - b_k, a_k - a_kp1]
        b_val = a_k * b_kp1 - b_k * a_kp1

        A_ineq.append(A_row)
        b_ineq.append(b_val)

    return np.array(A_ineq), np.array(b_ineq)


def get_edge_as_equality(vertices, edge_idx):
    """
    Convert one edge of polygon to equality constraint
    Returns A_eq, b_eq representing the line through the edge
    """
    n = len(vertices)
    p1 = vertices[edge_idx]
    p2 = vertices[(edge_idx + 1) % n]

    # Line through p1 and p2: (y2-y1)*x - (x2-x1)*y = y2*x1 - x2*y1
    A_eq = np.array([[p2[1] - p1[1], -(p2[0] - p1[0])]])
    b_eq = np.array([p2[1] * p1[0] - p2[0] * p1[1]])

    return A_eq, b_eq, p1, p2
```

```
76
77
78  #
        ============================================================================
79  # TASK 2: BUILD QP MATRICES
80  #
        ============================================================================
81
82  def build_transmission_qp(polygons, z0, z9, beta, alpha):
83      """
84      Build QP matrices for transmission line problem.
85
86      Variables ordered as: [z1_x, z1_y, z2_x, z2_y, ..., z8_x, z8_y]
87
88      Returns: G, c, A, b, Ae, be
89      """
90      n_vars = 16 # 8 points  2 coordinates
91
92      # Initialize matrices
93      G = np.zeros((n_vars, n_vars))
94      c = np.zeros(n_vars)
95
96      # Build objective: sum of squared distances
97      # Beta terms: ||z0-z1||^2, ||z2-z3||^2, ||z4-z5||^2, ||z6-z7||^2,
            ||z8-z9||^2
98      # Alpha terms: ||z1-z2||^2, ||z3-z4||^2, ||z5-z6||^2, ||z7-z8||^2
99
100     for k in range(5): # k = 0, 1, 2, 3, 4
101         if k == 0:
102             # ||z0 - z1||^2 with coefficient beta
103             # z1 is vars[0:2]
104             G[0:2, 0:2] += beta * np.eye(2)
105             c[0:2] += -beta * z0
106         elif k == 4:
107             # ||z8 - z9||^2 with coefficient beta
108             # z8 is vars[14:16]
109             G[14:16, 14:16] += beta * np.eye(2)
110             c[14:16] += -beta * z9
111         else:
112             # ||z_{2k} - z_{2k+1}||^2 with coefficient beta
113             idx1 = 2 * (2 * k) - 2 # z_{2k} position
114             idx2 = 2 * (2 * k + 1) - 2 # z_{2k+1} position
115             G[idx1:idx1+2, idx1:idx1+2] += beta * np.eye(2)
116             G[idx2:idx2+2, idx2:idx2+2] += beta * np.eye(2)
117             G[idx1:idx1+2, idx2:idx2+2] += -beta * np.eye(2)
```

```
            G[idx2:idx2+2, idx1:idx1+2] += -beta * np.eye(2)

    for k in range(1, 5): # k = 1, 2, 3, 4
        # ||z_{2k-1} - z_{2k}||^2 with coefficient alpha
        idx1 = 2 * (2 * k - 1) - 2
        idx2 = 2 * (2 * k) - 2
        G[idx1:idx1+2, idx1:idx1+2] += alpha * np.eye(2)
        G[idx2:idx2+2, idx2:idx2+2] += alpha * np.eye(2)
        G[idx1:idx1+2, idx2:idx2+2] += -alpha * np.eye(2)
        G[idx2:idx2+2, idx1:idx1+2] += -alpha * np.eye(2)

    # Build constraint matrices
    A_list = []
    b_list = []

    for k in range(1, 9): # z1 through z8
        poly_idx = k // 2
        if isinstance(polygons[poly_idx], dict) and 'type' in
            polygons[poly_idx]:
            # Special case: line segment
            if polygons[poly_idx]['type'] == 'line_segment':
                # Handle as inequality + equality constraints
                p1, p2 = polygons[poly_idx]['p1'], polygons[poly_idx]['p2']

                # Inequality: bounding box constraints
                x_min, x_max = min(p1[0], p2[0]), max(p1[0], p2[0])
                y_min, y_max = min(p1[1], p2[1]), max(p1[1], p2[1])


                var_idx = 2 * (k - 1)
                row = np.zeros(n_vars)
                row[var_idx] = -1
                A_list.append(row)
                b_list.append(-x_min)

                row = np.zeros(n_vars)
                row[var_idx] = 1
                A_list.append(row)
                b_list.append(x_max)

                row = np.zeros(n_vars)
                row[var_idx + 1] = -1
                A_list.append(row)
                b_list.append(-y_min)

```

```
162                 row = np.zeros(n_vars)
163                 row[var_idx + 1] = 1
164                 A_list.append(row)
165                 b_list.append(y_max)
166         else:
167 
168             A_poly, b_poly = vertices_to_halfspaces(polygons[poly_idx])
169             var_idx = 2 * (k - 1)
170 
171             for i in range(len(A_poly)):
172                 row = np.zeros(n_vars)
173                 row[var_idx:var_idx+2] = A_poly[i]
174                 A_list.append(row)
175                 b_list.append(b_poly[i])
176 
177     A = np.array(A_list) if A_list else np.zeros((0, n_vars))
178     b = np.array(b_list) if b_list else np.zeros(0)
179 
180     # Equality constraints (for line segments)
181     Ae_list = []
182     be_list = []
183 
184     for k in range(1, 9):
185         poly_idx = k // 2
186         if isinstance(polygons[poly_idx], dict) and
                polygons[poly_idx]['type'] == 'line_segment':
187             A_eq_poly = polygons[poly_idx]['A_eq']
188             b_eq_poly = polygons[poly_idx]['b_eq']
189             var_idx = 2 * (k - 1)
190 
191             row = np.zeros(n_vars)
192             row[var_idx:var_idx+2] = A_eq_poly[0]
193             Ae_list.append(row)
194             be_list.append(b_eq_poly[0])
195 
196     Ae = np.array(Ae_list) if Ae_list else np.zeros((0, n_vars))
197     be = np.array(be_list) if be_list else np.zeros(0)
198 
199     return G, c, A, b, Ae, be
200 
201 
202 #
      ============================================================================
203 # TASK 3: QP SOLVERS
```

```
204 # ============================================================================
205
206 def solve_qp(G, c, A, b, Ae, be, x0=None):
207     """
208     Solve general QP: min 0.5*x'Gx + c'x
209                       s.t. Ax <= b
210                            Ae*x = be
211
212     Handles 4 cases:
213     (a) Linear program (G = 0)
214     (b) Unconstrained QP
215     (c) Equality-constrained QP
216     (d) Inequality-constrained QP (Active Set Method)
217     """
218     n = len(c)
219
220     if x0 is None:
221         x0 = np.zeros(n)
222
223     # Check if G is zero (linear program)
224     if np.allclose(G, 0):
225         return solve_lp(c, A, b, Ae, be)
226
227     # Check if unconstrained
228     if (A.size == 0 or len(A) == 0) and (Ae.size == 0 or len(Ae) == 0):
229         return solve_unconstrained_qp(G, c)
230
231     # Check if only equality constraints
232     if A.size == 0 or len(A) == 0:
233         return solve_equality_qp(G, c, Ae, be)
234
235     # General case: inequality constraints (with possible equality
        constraints)
236     return solve_active_set_qp(G, c, A, b, Ae, be, x0)
237
238
239 def solve_lp(c, A, b, Ae, be):
240     """Solve LP using scipy.optimize.linprog"""
241     # linprog minimizes c'x subject to A_ub @ x <= b_ub and A_eq @ x ==
        b_eq
242     A_ub = A if A.size > 0 else None
243     b_ub = b if b.size > 0 else None
244     A_eq = Ae if Ae.size > 0 else None
245     b_eq = be if be.size > 0 else None
```

```
246
247     result = linprog(c, A_ub=A_ub, b_ub=b_ub, A_eq=A_eq, b_eq=b_eq,
            method='highs')
248
249     if result.success:
250         return result.x
251     else:
252         raise ValueError("LP solver failed")
253
254
255 def solve_unconstrained_qp(G, c):
256     """Solve unconstrained QP: min 0.5*x'Gx + c'x using Newton step"""
257     # Optimal: G*x + c = 0 => x = -G^{-1} * c
258     try:
259         x = np.linalg.solve(G, -c)
260         return x
261     except np.linalg.LinAlgError:
262         x = -np.linalg.pinv(G) @ c
263         return x
264
265
266 def solve_equality_qp(G, c, Ae, be):
267     """
268     Solve equality-constrained QP using CG method (null space approach)
269     min 0.5*x'Gx + c'x subject to Ae*x = be
270     """
271     # KKT system: [G Ae'] [x ] = [-c ]
272     # [Ae 0 ] [lam] [be ]
273
274     n = G.shape[0]
275     m = Ae.shape[0] if Ae.size > 0 else 0
276
277     if m == 0:
278         return solve_unconstrained_qp(G, c)
279
280     # Build KKT matrix
281     KKT = np.zeros((n + m, n + m))
282     KKT[:n, :n] = G
283     KKT[:n, n:] = Ae.T
284     KKT[n:, :n] = Ae
285
286     rhs = np.concatenate([-c, be])
287
288     try:
289         sol = np.linalg.solve(KKT, rhs)
```

```
        return sol[:n]
    except np.linalg.LinAlgError:
        # Use least squares
        sol, _, _, _ = np.linalg.lstsq(KKT, rhs, rcond=None)
        return sol[:n]


def solve_active_set_qp(G, c, A, b, Ae, be, x0, max_iter=1000, tol=1e-8):
    """
    Solve QP with inequality constraints using Active Set Method
    """
    n = len(c)
    m = len(b) if b.size > 0 else 0
    me = len(be) if be.size > 0 else 0

    # Find initial feasible point
    x = find_feasible_point(A, b, Ae, be, x0)

    # Initialize active set
    active_set = set(range(m, m + me)) # Equality constraints always
        active

    # Check which inequalities are active at x0
    if m > 0:
        slack = b - A @ x
        for i in range(m):
            if abs(slack[i]) < tol:
                active_set.add(i)

    for iteration in range(max_iter):
        # Build active constraint matrix
        active_indices = list(active_set)
        if len(active_indices) > 0:
            A_active = []
            b_active = []
            for idx in active_indices:
                if idx < m:
                    A_active.append(A[idx])
                    b_active.append(b[idx])
                else:
                    A_active.append(Ae[idx - m])
                    b_active.append(be[idx - m])
            A_active = np.array(A_active)
            b_active = np.array(b_active)
        else:
```

```
            A_active = np.zeros((0, n))
            b_active = np.zeros(0)

        # Solve equality-constrained QP for search direction

        if len(active_indices) > 0:
            p = solve_equality_qp(G, G @ x + c, A_active,
                np.zeros(len(active_indices)))
        else:
            p = solve_unconstrained_qp(G, G @ x + c)

        # Check if p is zero (KKT point)
        if np.linalg.norm(p) < tol:
            # Compute Lagrange multipliers
            if len(active_indices) > 0:
                lam = compute_lagrange_multipliers(G, c, x, A_active)

                # Check if all multipliers for inequalities are
                    non-negative
                min_lam_idx = -1
                min_lam = 0
                for i, idx in enumerate(active_indices):
                    if idx < m: # Inequality constraint
                        if lam[i] < min_lam:
                            min_lam = lam[i]
                            min_lam_idx = idx

                if min_lam < -tol:
                    # Remove constraint with most negative multiplier
                    active_set.remove(min_lam_idx)
                    continue

            # KKT conditions satisfied
            break

        # Compute step size (maximum step before hitting constraint)
        alpha = 1.0
        blocking_constraint = -1

        if m > 0:
            Ap = A @ p
            for i in range(m):
                if i not in active_set and Ap[i] > tol:
                    alpha_i = (b[i] - A[i] @ x) / Ap[i]
                    if alpha_i < alpha:
```

```
                        alpha = alpha_i
                        blocking_constraint = i

        # Update x
        x = x + alpha * p

        # Add blocking constraint to active set
        if blocking_constraint >= 0:
            active_set.add(blocking_constraint)

    return x


def find_feasible_point(A, b, Ae, be, x0):
    """Find a feasible starting point for QP"""
    n = len(x0)

    # Check if x0 is feasible
    feasible = True
    if A.size > 0 and len(A) > 0:
        if np.any(A @ x0 > b + 1e-6):
            feasible = False
    if Ae.size > 0 and len(Ae) > 0:
        if not np.allclose(Ae @ x0, be, atol=1e-6):
            feasible = False

    if feasible:
        return x0.copy()

    # Solve phase 1 problem to find feasible point
    # min s subject to Ax <= b + s, Ae*x = be, s >= 0
    m = len(b) if b.size > 0 else 0
    me = len(be) if be.size > 0 else 0

    # Variables: [x, s]
    c_phase1 = np.zeros(n + 1)
    c_phase1[-1] = 1 # Minimize s

    # Inequality constraints: Ax - s <= b
    if m > 0:
        A_phase1 = np.hstack([A, -np.ones((m, 1))])
        b_phase1 = b
    else:
        A_phase1 = np.zeros((0, n + 1))
        b_phase1 = np.zeros(0)
```

```
422
423     # s >= 0 => -s <= 0
424     s_constraint = np.zeros((1, n + 1))
425     s_constraint[0, -1] = -1
426     A_phase1 = np.vstack([A_phase1, s_constraint]) if A_phase1.size > 0
            else s_constraint
427     b_phase1 = np.concatenate([b_phase1, [0]]) if b_phase1.size > 0 else
            np.array([0])
428
429     # Equality constraints: Ae*x = be (s doesn't appear)
430     if me > 0:
431         Ae_phase1 = np.hstack([Ae, np.zeros((me, 1))])
432         be_phase1 = be
433     else:
434         Ae_phase1 = np.zeros((0, n + 1))
435         be_phase1 = np.zeros(0)
436
437     result = linprog(c_phase1, A_ub=A_phase1, b_ub=b_phase1,
438                      A_eq=Ae_phase1 if Ae_phase1.size > 0 else None,
439                      b_eq=be_phase1 if be_phase1.size > 0 else None,
440                      method='highs')
441
442     if result.success and result.fun < 1e-6:
443         return result.x[:n]
444     else:
445         # Return x0 and hope for the best
446         return x0.copy()
447
448
449 def compute_lagrange_multipliers(G, c, x, A_active):
450     """Compute Lagrange multipliers for active constraints"""
451     # At optimum: G*x + c = A_active' * lambda
452     # => lambda = (A_active * A_active')^{-1} * A_active * (G*x + c)
453     grad = G @ x + c
454
455     try:
456         AAt = A_active @ A_active.T
457         lam = np.linalg.solve(AAt, A_active @ grad)
458         return lam
459     except np.linalg.LinAlgError:
460         # Singular, use pseudo-inverse
461         lam = np.linalg.pinv(A_active.T) @ grad
462         return lam
463
464
```

```
#
    ================================================================================
# PLOTTING AND OUTPUT
#
    ================================================================================

def plot_solution(polygons, z0, z9, z_opt, alpha, beta, filename):
    """Plot polygons and optimal path"""
    plt.figure(figsize=(14, 8))

    # Plot polygons
    for i, poly in enumerate(polygons):
        if isinstance(poly, dict) and poly['type'] == 'line_segment':
            # Plot line segment
            p1, p2 = poly['p1'], poly['p2']
            plt.plot([p1[0], p2[0]], [p1[1], p2[1]], 'b-', linewidth=2)
            plt.plot([p1[0], p2[0]], [p1[1], p2[1]], 'bo', markersize=6)
            plt.text(np.mean([p1[0], p2[0]]), np.mean([p1[1], p2[1]]) +
                0.5,
                    f'$P_{i}$ (segment)', color='blue', fontsize=10,
                        ha='center')
        else:
            # Plot polygon
            poly_closed = np.vstack([poly, poly[0]])
            plt.plot(poly_closed[:, 0], poly_closed[:, 1], 'b-',
                linewidth=2)
            centroid = np.mean(poly, axis=0)
            plt.text(centroid[0], centroid[1], f'$P_{i}$', color='blue',
                    fontsize=12, ha='center', weight='bold')

    # Plot optimal path
    all_points = [z0] + [z_opt[2*i:2*i+2] for i in range(8)] + [z9]
    path = np.array(all_points)
    plt.plot(path[:, 0], path[:, 1], 'o-', color='orange', linewidth=3,
            markersize=8, label='Optimal Path')

    # Label points
    plt.plot(z0[0], z0[1], 'ko', markersize=10)
    plt.text(z0[0] - 0.1, z0[1]+ 0.5, '$z_0$', fontsize=12, ha='right')

    for i in range(8):
        zi = z_opt[2*i:2*i+2]
        plt.plot(zi[0], zi[1], 'ro', markersize=8)
        # Shift label up and to the right
```

```
        plt.text(zi[0] - 0.5, zi[1] - 0.5, f'$z_{i+1}$', fontsize=12,
            ha='left', va='bottom')

    plt.plot(z9[0], z9[1], 'ko', markersize=10)
    plt.text(z9[0] - 0.1, z9[1]+ 0.5, '$z_9$', fontsize=12, ha='left')

    plt.xlabel('x', fontsize=18)
    plt.xticks(fontsize=14)
    plt.ylabel('y', fontsize=18)
    plt.yticks(fontsize=14)
    # plt.title(f'Optimal Transmission Line Layout (={alpha}, ={beta})',
        fontsize=14)
    plt.legend(fontsize=14)
    plt.grid(True, alpha=0.3)
    plt.axis('equal')
    plt.tight_layout()
    plt.savefig(filename, dpi=150, bbox_inches='tight')
    plt.close()



def compute_objective(z_opt, z0, z9, beta, alpha):
    """Compute the objective function value"""
    obj = 0.0

    # Beta terms
    for k in range(5):
        if k == 0:
            z1 = z_opt[0:2]
            obj += beta / 2 * np.sum((z0 - z1) ** 2)
        elif k == 4:
            z8 = z_opt[14:16]
            obj += beta / 2 * np.sum((z8 - z9) ** 2)
        else:
            z2k = z_opt[2*(2*k)-2:2*(2*k)]
            z2k1 = z_opt[2*(2*k+1)-2:2*(2*k+1)]
            obj += beta / 2 * np.sum((z2k - z2k1) ** 2)

    # Alpha terms
    for k in range(1, 5):
        z2k_1 = z_opt[2*(2*k-1)-2:2*(2*k-1)]
        z2k = z_opt[2*(2*k)-2:2*(2*k)]
        obj += alpha / 2 * np.sum((z2k_1 - z2k) ** 2)

    return obj
```

```
547
548
549 def save_results(results, filename):
550     """Save results to text file"""
551     with open(filename, 'w') as f:
552         f.write("=" * 80 + "\n")
553         f.write("TRANSMISSION LINE OPTIMIZATION RESULTS\n")
554         f.write("=" * 80 + "\n\n")
555
556         for result in results:
557             f.write(f"Alpha: {result['alpha']:.2f}, Beta:
                    {result['beta']:.2f}\n")
558             f.write(f"Objective Value: {result['objective']:.6f}\n")
559             f.write(f"Alpha/Beta Ratio:
                    {result['alpha']/result['beta']:.2f}\n")
560             f.write("\nOptimal Points:\n")
561             for i in range(8):
562                 zi = result['z_opt'][2*i:2*i+2]
563                 f.write(f" z{i+1} = ({zi[0]:.4f}, {zi[1]:.4f})\n")
564             f.write("\n" + "-" * 80 + "\n\n")
565
566
567 #
    ================================================================================
568 # TASK 4: SOLVE FOR VARIOUS ALPHA VALUES
569 #
    ================================================================================
570
571 def run_task4():
572     """Run Task 4: Solve for various alpha values"""
573     print("\n" + "=" * 80)
574     print("TASK 4: Solving transmission line problem for various alpha
        values")
575     print("=" * 80)
576
577     results = []
578
579     for alpha in alpha_values:
580         print(f"\nSolving for alpha = {alpha}, beta = {beta}...")
581
582         # Build QP
583         G, c, A, b, Ae, be = build_transmission_qp(polygons, z0, z9, beta,
            alpha)
584
585         # Initial guess (centroid of each polygon)
```

```
        x0 = []
        for k in range(1, 9):
            poly_idx = k // 2
            centroid = np.mean(polygons[poly_idx], axis=0)
            x0.extend(centroid)
        x0 = np.array(x0)

        # Solve QP
        z_opt = solve_qp(G, c, A, b, Ae, be, x0)

        # Compute objective
        obj = compute_objective(z_opt, z0, z9, beta, alpha)

        print(f" Objective value: {obj:.6f}")

        # Save results
        result = {
            'alpha': alpha,
            'beta': beta,
            'z_opt': z_opt,
            'objective': obj
        }
        results.append(result)

        # Plot
        plot_filename = f"task4_alpha_{alpha:.1f}.png"
        plot_solution(polygons, z0, z9, z_opt, alpha, beta, plot_filename)
        print(f" Plot saved to {plot_filename}")

    # Save all results
    save_results(results, "task4_results.txt")
    print("\nAll results saved to task4_results.txt")

    return results


#
    ================================================================================
# TASK 5: LINE SEGMENT CONSTRAINT
#
    ================================================================================

def run_task5():
    """Run Task 5: Convert one polygon to line segment"""
    print("\n" + "=" * 80)
```

```
    print("TASK 5: Solving with line segment constraint")
    print("=" * 80)

    # Modify polygons
    polygons_task5 = polygons.copy()
    vertices = polygons_task5[TASK5_POLYGON_IDX]

    # Get edge as line segment
    A_eq, b_eq, p1, p2 = get_edge_as_equality(vertices, TASK5_EDGE_IDX)

    print(f"\nConverting polygon P{TASK5_POLYGON_IDX}, edge
        {TASK5_EDGE_IDX} to line segment")
    print(f" Segment endpoints: {p1} to {p2}")

    # Replace polygon with line segment info
    polygons_task5[TASK5_POLYGON_IDX] = {
        'type': 'line_segment',
        'p1': p1,
        'p2': p2,
        'A_eq': A_eq,
        'b_eq': b_eq
    }

    # Build and solve QP
    print(f"Solving for alpha = {TASK5_ALPHA}, beta = {beta}...")
    G, c, A, b, Ae, be = build_transmission_qp(polygons_task5, z0, z9,
        beta, TASK5_ALPHA)

    # Initial guess
    x0 = []
    for k in range(1, 9):
        poly_idx = k // 2
        if isinstance(polygons_task5[poly_idx], dict):
            # Use midpoint of line segment
            p1, p2 = polygons_task5[poly_idx]['p1'],
                polygons_task5[poly_idx]['p2']
            centroid = (p1 + p2) / 2
        else:
            centroid = np.mean(polygons_task5[poly_idx], axis=0)
        x0.extend(centroid)
    x0 = np.array(x0)

    # Solve
    z_opt = solve_qp(G, c, A, b, Ae, be, x0)

```

```
    # Compute objective
    obj = compute_objective(z_opt, z0, z9, beta, TASK5_ALPHA)

    print(f" Objective value: {obj:.6f}")

    # Verify points on line segment
    k_segment = [k for k in range(1, 9) if k // 2 == TASK5_POLYGON_IDX]
    print(f"\nVerifying points on line segment:")
    for k in k_segment:
        zi = z_opt[2*(k-1):2*(k-1)+2]
        dist_to_line = abs(A_eq[0, 0] * zi[0] + A_eq[0, 1] * zi[1] -
            b_eq[0]) / np.linalg.norm(A_eq[0])
        print(f" z{k} = ({zi[0]:.4f}, {zi[1]:.4f}), distance to line:
            {dist_to_line:.6e}")

    # Save results
    result = [{
        'alpha': TASK5_ALPHA,
        'beta': beta,
        'z_opt': z_opt,
        'objective': obj
    }]
    save_results(result, "task5_results.txt")

    # Plot
    plot_solution(polygons_task5, z0, z9, z_opt, TASK5_ALPHA, beta,
        "task5_solution.png")
    print("\nPlot saved to task5_solution.png")
    print("Results saved to task5_results.txt")


#
    ================================================================================
# MAIN EXECUTION
#
    ================================================================================

if __name__ == "__main__":

    # Create results folder in same directory as script
    script_dir = os.path.dirname(os.path.abspath(__file__))
    results_dir = os.path.join(script_dir, "results-p3")
    os.makedirs(results_dir, exist_ok=True)
    os.chdir(results_dir)

```

```
    # Run Task 4
    task4_results = run_task4()

    # Run Task 5 if enabled
    if TASK5_ENABLED:
        run_task5()

    print("\n" + "=" * 80)
    print("ALL TASKS COMPLETED")
    print("=" * 80)
```