1. SDP Relaxation and Heuristics for Two-Way Partitioning Problem

   (a) Q 5.39 textbook

$$\begin{aligned} &min\ x^T W x \\ s.t.\ &x_i^2 = 1, \forall i \in \{1, .., n\} \end{aligned}$$

   i. Show that the two-way partitioning problem can be cast as

$$\begin{aligned} &min\ tr(WX) \\ s.t.\ &X \succeq 0, rank(X) = 1 \\ &X_{ii} = 1, \forall i \in \{1, .., n\} \end{aligned}$$

$x^T W x = tr(x^T W x) = tr(W x x^T)$
$let\ X = x x^T$
$(\forall i) x_i^2 = 1 \iff x_i = \{-1, 1\} \implies x^T I x = n$
$x^T I x = tr(x x^T) = n$
$(\forall i, j) X_{ij} = \{-1, 1\}$
$((\exists i) X_{ii} = -1 \implies tr(X) < n)$
$thus, for\ tr(X) = n : (\forall i) X_{ii} = 1$

$X = x x^T = x \begin{bmatrix} a_1 & a_2 & .. & a_n \end{bmatrix} = \begin{bmatrix} a_1 x & a_2 x & .. & a_n x \end{bmatrix}, a_i \in \mathbb{R}, x \in \mathbb{R}^n$
$(\forall i)(\exists j) \beta_{ij} a_i x = a_j x \implies \beta_{ij} a_i x - a_j x = 0$
$let\ \gamma_{ij} = \beta_{ij} a_i - a_j$
$\gamma_{ij} x = 0$
$x \neq 0 \implies ((\forall i)(\exists j) \gamma_{ij} = 0 \implies linear\ dependence\ between\ all\ column\ vectors\ of\ X)$
$thus,\ rank(X) = 1$

$(\forall w) w^T X w = w^T x x^T w = (x^T w)^T x^T w$
$(\forall i, w)(x^T w)_i (x^T w)_i \geq 0 \implies (\forall w)(x^T w)^T (x^T w) \geq 0 \iff X\ is\ SPD$

Combining all constraints and objective forms the desired result.

ii. SDP relaxation of two-way partitioning problem. Using the formulation in part (a), we can form the relaxation:

$$\begin{aligned} min\ tr(WX) \\ s.t.\ X \succeq 0 \\ X_{ii} = 1, \forall i \in \{1, .., n\} \end{aligned}$$

This problem is an SDP, and therefore can be solved efficiently. Explain why its optimal value gives a lower bound on the optimal value of the two-way partitioning problem (5.113). What can you say if an optimal point X^* for this SDP has rank one?

$$L(X, Z, v) = tr(WX) - tr(XZ) + tr(diag(v)diag(X) - I)$$
$$L(X, Z, v) = -tr(diag(v)I) + tr(WX) - tr(XZ) + tr(diag(v)diag(X))$$
$$g(Z, V) = \begin{cases} -1^T v, & W - Z + diag(v) \succeq 0 \\ -\infty, & o/w \end{cases}$$
$$dual\ problem:$$
$$\max_{Z,v} -1^T v$$
$$s.t.\ W - Z + diag(v) \succeq 0$$
$$Z \succeq 0$$

If an optimal point X^* for the relaxed problem has rank one:
X^* has minimal possible rank and $X^* \neq 0$, $X^* \succeq 0$.
$X^* \succeq 0$, so primal feasible.
Functions are all differentiable, KKT conditions apply at optimality where there exists a dual solution.
Dual of relaxed problem is feasible: $W + diag(v) \succeq Z, Z \succeq 0$
Using complementary slackness: $X^* \neq 0, -tr(X^*Z^*) = 0 \implies Z^* = 0$.

Dual problem of relaxed problem at optimality:

$$\begin{aligned} &\max_{v,Z} -1^T v = [\max_{v}\ -1^T v]_{Z=Z^*} \\ &s.t.\ W + diag(v) \succeq Z^*, Z^* = 0 \implies \\ &s.t.\ W + diag(v) \succeq 0 \end{aligned}$$

This solution is equivalent to the solution of the dual of the original problem. Thus, if $rank(X^*) = 1$ of the relaxed problem, X^* obtains the same solution as the original problem where $x^*x^{*T} = X^*$ as required.

iii. We now have two SDPs that give a lower bound on the optimal value of the two-way partitioning problem (5.113): the SDP relaxation (5.115) found in part (b), and the Lagrange dual of the two-way partitioning problem, given in (5.114). What is the relation between the two SDPs? What can you say about the lower bounds found by them? Hint: Relate the two SDPs via duality.

$$\begin{aligned} (5.115)\ & min\ tr(WX) \\ & s.t.\ X \succeq 0 \\ & X_{ii} = 1, \forall i \in \{1, .., n\} \end{aligned}$$

$$\begin{aligned} (5.114)\ & maximize\ \ -1^T v \\ & s.t.\ W + diag(v) \succeq 0 \end{aligned}$$

Taken from previous section, dual of relaxed problem:

$$\begin{aligned} & L(X, Z, v) = tr(WX) - tr(XZ) + tr(diag(v)diag(X) - I) \\ & L(X, Z, v) = -tr(diag(v)I) + tr(WX) - tr(XZ) + tr(diag(v)diag(X)) \\ & g(Z, V) = \begin{cases} -1^T v, & W - Z + diag(v) \succeq 0 \\ -\infty, & o/w \end{cases} \end{aligned}$$

Dual problem:

$$\begin{aligned} & \max_{Z,v} -1^T v \\ & s.t.\ W - Z + diag(v) \succeq 0 \\ & Z \succeq 0 \end{aligned}$$

It is evident that solution to the dual of relaxed problem has a tightened generalized inequality constraint due to Z compared to (5.114):

$$W + diag(v) \succeq Z$$

This leads to potentially bigger v and hence potentially larger objective value of dual maximization problem (smaller objective value to the primal minimization problem). Thus, solution of relaxed problem provides a lower bound to the the original problem.

(b) Q 11.23(b-d) textbook

	SDP bound (11.66)	Optimum	b (11.67)	c
small	5.33445288482	5.33440509588	12.30891878	5.33440509588
medium	-42.2266162135	-38.3691372273	-13.05483539	-38.3691372273
large	66.0855132055	x	2135.80923688	457.832259292

	d-a	d-b	d-c
small	5.33440509588	12.3089187826	5.33440509588
medium	-38.3691372273	-16.4179991404	-38.3691372273
large	1053.83794696	1401.39879583	424.105035811

Comparison of heuristic parition (b) to SDP bound:
All obtained objetive values are within the expected range given by SDP lower bound, they do not close to the lower bound.

Comparison of randomized method (c) to SDP bound:
All obtained objective values are within the expected range given by SDP lower bound, and they are all closer to the SDP bound compared to (b)

100 random samples + greedy single coordinate search (d-a):
The obtained objective values work well for small to medium dimension sized problems since it effetively cover a majority of permutations possible, but becomes less effective for large problems due to exponential dimensionality explosion.

Single heuristic solution + greedy single coordinate search (d-b):
As expected, the obtained objective values are equivalent or better than Heuristic parition (b) alone, but is dependent on heuristic solution as the algorithm is greedy. Local search is show to have bigger marginal impact for higher dimensional problem.

Randomized method + greedy single coordinate search (d-c):
As expected, the obtained objective values are equivalent or better than randomized method (c) alone. Local search is show to have bigger marginal impact for higher dimensional problem. Local search plus randomized method seems to give a good balance in practice.

Optimum (exhaustive):

```
import cvxpy as cp
import numpy as np
from scipy.io import loadmat
import numpy.linalg as linalg
import math
import copy

def gen_seq_exhaustive(dim):

    q = [[1], [-1]]

    while len(q[0]) < dim:
        qq =[]
        for i in q:
            a = copy.deepcopy(i)
            b = copy.deepcopy(i)
            a.append(1)
            b.append(-1)
            s = len(i)
            assert(len(a)==len(b))
            assert(len(a)==s+1)
            qq.append(a)
            qq.append(b)
        q = qq

    ret = np.zeros((len(q), dim))
    idx = 0
    for i in q:
        ret[idx,:] = i
        idx += 1

    return ret
```

```
def solve_d_exhaustive(W):
    dim = W.shape[0]
    samples = gen_seq_exhaustive(dim)
    best_val = math.inf
    best_x = None
    for i in range(0, samples.shape[0]):
        x = samples[i,:].T
        v = (x.T).dot(W).dot(x)
        if v < best_val:
            best_val = v
            best_x = x

    print("problem size:", W.shape[0])
    print("best_objective (exhaustive): ", best_val)
    print("-----")
    return best_val, best_x

m = loadmat('../data/hw4data.mat')
w5 = np.array(m['W5'])
w10 = np.array(m['W10'])

solve_d_exhaustive(w5)
solve_d_exhaustive(w10)
```

- b) heuristic partitioning

```
def solve(W):
    print("problem size:", W.shape[0])

    #dual of original:
    print("dual of original:")
    dim = W.shape[0]
    v = cp.Variable((dim,1))
    constraints = [W + cp.diag(v) >> 0]
    prob = cp.Problem(cp.Maximize( -cp.sum(v) ),
                      constraints)
    prob.solve()

    print("prob.status:", prob.status)

    lower_bound = 0

    if prob.status not in ["infeasible", "unbounded"]:
        print("Optimal value: %s" % prob.value)
        lower_bound = prob.value

    print("lower_bound:", lower_bound)

    #dual of relaxed:
    print("dual of relaxed:")
    X = cp.Variable((dim,dim))
    constraints = [X >> 0, cp.diag(X) == np.ones((dim,))]
    prob = cp.Problem(cp.Minimize( cp.trace(cp.matmul(W,X)) ),
                      constraints)
    prob.solve()

    print("prob.status:", prob.status)

    if prob.status not in ["infeasible", "unbounded"]:
        print("Optimal value: %s" % prob.value)

    ret = prob.variables()[0].value
    eigenValues, eigenVectors = linalg.eig(ret)

    idx = eigenValues.argsort()[::-1]
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:,idx]
    x_approx = np.sign(eigenVectors[0])[:,np.newaxis]
    p_heuristic = (x_approx.T).dot(W).dot(x_approx)
    print("heuristic objective: ", p_heuristic)
    print("-----")
```

```
m = loadmat('../data/hw4data.mat')
w5 = np.array(m['W5'])
w10 = np.array(m['W10'])
w50 = np.array(m['W50'])

solve(w5)
solve(w10)
solve(w50)
```

- c) randomized method

```
import cvxpy as cp
import numpy as np
from scipy.io import loadmat
import numpy.linalg as linalg
import math

def solve(W):
    print("problem size:", W.shape[0])

    #dual of original:
    print("dual of original:")
    dim = W.shape[0]
    v = cp.Variable((dim,1))
    constraints = [W + cp.diag(v) >> 0]
    prob = cp.Problem(cp.Maximize( -cp.sum(v) ),
                      constraints)
    prob.solve()

    lower_bound = 0
    if prob.status not in ["infeasible", "unbounded"]:
        print("Optimal value: %s" % prob.value)
        lower_bound = prob.value

    print("lower_bound: ", lower_bound)

    #dual of relaxed:

    X = cp.Variable((dim,dim))
    constraints = [X >> 0, cp.diag(X) == np.ones((dim,))]
    prob = cp.Problem(cp.Minimize( cp.trace(cp.matmul(W,X)) ),
                      constraints)
    prob.solve(solver=cp.SCS, max_iters=4000,
               eps=1e-11, warm_start=True)

    if prob.status not in ["infeasible", "unbounded"]:
        print("Optimal value: %s" % prob.value)

    ret = prob.variables()[0].value

    K = 100 #number of samples
    xs_approx = np.random.multivariate_normal(np.zeros((dim,)),
                                              ret, size=(K))
    xs_approx = np.sign(xs_approx) #shape: (K,dim)

    p_best = math.inf
    for i in range(0,K):
```

```
            p = (xs_approx[i,:].dot(W)).dot(xs_approx[i,:].T)
            if p < p_best:
                p_best = p

        print("best objective (randomized): ", p_best, "size: ", K)
        print("-----")

        return p_best, xs_approx

m = loadmat('../data/hw4data.mat')
w5 = np.array(m['W5'])
w10 = np.array(m['W10'])
w50 = np.array(m['W50'])

solve(w5)
solve(w10)
solve(w50)
```

- d) greedy heuristic refinement

  d-a (100 randomized initial points, single coordinate greedy search):

```
#generate samples each having dim numbers in {-1,1}
def gen_seq(dim, samples):
    return np.sign(np.random.rand(samples, dim) - 0.5)

def greedy(x,W):
    xx = x
    val = (xx.T).dot(W).dot(xx)

    while True:
        idx = None
        for i in range(0, xx.size):
            y = xx
            y[i] = -y[i]
            v = (y.T).dot(W).dot(y)
            if v < val:
                val = v
                idx = i
        if idx is None:
            break
        else:
            xx[i] = -xx[i]
    return val, xx

def solve_d_a(W):
    dim = W.shape[0]
    K = 100
    samples = gen_seq(dim, K)
    best_val = math.inf
    best_x = None
    for i in range(0,samples.shape[0]):
        x = samples[i,:].T
        val, xx = greedy(x, W)
        if val < best_val:
            best_val = val
            best_x = xx

    print("problem size:", W.shape[0])
    print("best_objective: ", best_val)
    print("-----")
    return best_val, best_x

m = loadmat('../data/hw4data.mat')
w5 = np.array(m['W5'])
```

```
w10 = np.array(m['W10'])
w50 = np.array(m['W50'])

solve_d_a(w5)
solve_d_a(w10)
solve_d_a(w50)
```

d-b (a heuristic solution + single coordinate greedy search):

```
def solve_heur(W):

    #dual of original:
    dim = W.shape[0]
    v = cp.Variable((dim,1))
    constraints = [W + cp.diag(v) >> 0]
    prob = cp.Problem(cp.Maximize( -cp.sum(v) ),
                      constraints)
    prob.solve()

    lower_bound = 0
    if prob.status not in ["infeasible", "unbounded"]:
        print("Optimal value: %s" % prob.value)
        lower_bound = prob.value

    print("lower_bound:", lower_bound)

    #dual of relaxed:
    X = cp.Variable((dim,dim))
    constraints = [X >> 0, cp.diag(X) == np.ones((dim,))]
    prob = cp.Problem(cp.Minimize( cp.trace(cp.matmul(W,X)) ),
                      constraints)
    prob.solve()

    if prob.status not in ["infeasible", "unbounded"]:
        print("Optimal value: %s" % prob.value)

    ret = prob.variables()[0].value
    eigenValues, eigenVectors = linalg.eig(ret)

    idx = eigenValues.argsort()[::-1]
    eigenValues = eigenValues[idx]
    eigenVectors = eigenVectors[:,idx]

    x_approx = np.sign(eigenVectors[0])[:,np.newaxis]
    p_heuristic = (x_approx.T).dot(W).dot(x_approx)
    print("heuristic objective: ", p_heuristic)

    return p_heuristic, x_approx.T

def greedy(x,W):
    xx = x
    val = (xx.T).dot(W).dot(xx)
    while True:
        idx = None
        for i in range(0, xx.size):
```

```
            y = xx
            y[i] = -y[i]
            v = (y.T).dot(W).dot(y)
            if v < val:
                val = v
                idx = i
        if idx is None:
            break
        else:
            xx[i] = -xx[i]
    return val, xx

def solve_d_b(W):

    print("performing greedy search")
    dim = W.shape[0]
    best_val = math.inf
    best_x = None
    _, xs = solve_heur(W)

    for i in range(0,xs.shape[0]):
        val, xx = greedy(xs[i,:].T, W)
        if val < best_val:
            best_val = val
            best_x = xx

    print("problem size:", W.shape[0])
    print("best objective (heuristic+greedy): ", best_val )
    print("-----")
    return best_val, best_x

m = loadmat('../data/hw4data.mat')
w5 = np.array(m['W5'])
w10 = np.array(m['W10'])
w50 = np.array(m['W50'])

solve_d_b(w5)
solve_d_b(w10)
solve_d_b(w50)
```

d-c (100 samples from randomized method + single coordinate greedy search):

```
def solve_rand(W):
    #dual of original:
    print("dual of original:")
    dim = W.shape[0]
    v = cp.Variable((dim,1))
    constraints = [W + cp.diag(v) >> 0]
    prob = cp.Problem(cp.Maximize( -cp.sum(v) ),
                      constraints)
    prob.solve()

    lower_bound = 0
    if prob.status not in ["infeasible", "unbounded"]:
        print("Optimal value: %s" % prob.value)
        lower_bound = prob.value

    print("lower_bound: ", lower_bound)

    #dual of relaxed:

    #restrict to PSD for randomized sampling
    #on proper covariance matrix later
    X = cp.Variable((dim,dim), PSD=True)

    constraints = [X >> 0, cp.diag(X) == np.ones((dim,))]
    prob = cp.Problem(cp.Minimize( cp.trace(cp.matmul(W,X)) ),
                      constraints)
    prob.solve(solver=cp.SCS, max_iters=4000,
               eps=1e-11, warm_start=True)

    print("prob.status:", prob.status)

    if prob.status not in ["infeasible", "unbounded"]:
        print("Optimal value: %s" % prob.value)

    ret = prob.variables()[0].value

    K = 100 #number of samples
    xs_approx = np.random.multivariate_normal(np.zeros((dim,)),
                                              ret, size=(K))
    xs_approx = np.sign(xs_approx) #shape: (K,dim)

    return xs_approx

def greedy(x,W):
    xx = x
    val = (xx.T).dot(W).dot(xx)
```

```
    while True:
        idx = None
        for i in range(0, xx.size):
            y = xx
            y[i] = -y[i]
            v = (y.T).dot(W).dot(y)
            if v < val:
                val = v
                idx = i
        if idx is None:
            break
        else:
            xx[i] = -xx[i]
    return val, xx

def solve_d_c(W):

    print("performing greedy search")
    dim = W.shape[0]
    best_val = math.inf
    best_x = None
    xs = solve_rand(W)

    for i in range(0,xs.shape[0]):
        val, xx = greedy(xs[i,:].T, W)
        if val < best_val:
            best_val = val
            best_x = xx

    print("problem size:", W.shape[0])
    print("best objective (randomized+greedy): ", best_val )
    print("-----")
    return best_val, best_x

m = loadmat('../data/hw4data.mat')
w5 = np.array(m['W5'])
w10 = np.array(m['W10'])
w50 = np.array(m['W50'])

solve_d_c(w5)
solve_d_c(w10)
solve_d_c(w50)
```

2. Interior Point Method

$$\begin{aligned} \min_x \sum_l \frac{x_l}{c_l - x_l} \\ s.t.\ A^+x = s \\ Bx \leq b \\ c_l \geq x_l \geq 0, \forall l \in \{1, .., L\} \end{aligned}$$

- a) verify minimum-delay problem is a convex optimization problem.

$f_0(x) = \sum_l \frac{x_l}{c_l - x_l}$
$f_1(x) = x - c \leq 0$
$f_2(x) = -x \leq 0$
$f_3(x) = Bx - b$
$h(x) = A^+x - s$

$reformulation:$
$\min_x f_0(x)$
$s.t.\ h(x) = 0$
$f_i(x) \leq 0, \forall i = 1, .., 3$

$f_1, f_2, f_3, h_1\ are\ affine$
$\frac{\partial}{\partial x_i} f_0(x) = \frac{1}{c_i - x_i} + \frac{x_i}{(c_i - x_i)^2}$
$\frac{\partial^2}{\partial x_i^2} f_0(x) = \frac{2}{(c_i - x_i)^2} + \frac{2x_i}{(c_i - x_i)^3}$
$\frac{\partial^2}{\partial x_i x_j} f_0(x)|_{i \neq j} = 0$
$let\ X\ be\ partial\ feasibility\ set\ abiding\ constraints\ f_2, f_3$
$x \in X \implies (\forall i) x_i \in [0, 1], (\forall i) c_i = 1 \implies \frac{\partial^2}{\partial x_i^2} f_0(x) > 0$
$(\forall x \in X)(\nabla f_0(x)_{ii} > 0 \wedge \nabla f_0(x)_{ij}|_{i \neq j} = 0) \implies \nabla f_0(x) \in S_{++}^L\ over\ feasibility\ set$

Thus, it is a convex optimization problem.

- b) Use an interior-point method to find the optimal $x_1, .., x_{13}$. Please use a log barrier function for inequality constraints and use the infeasible-start Newtons method for the equality constraint. Please write down the relevant gradient, Hessian, and update equations explicitly. What is the minimal delay?

  Formulation:

$$\begin{aligned}\min_{y,x}\ & f_0 - \frac{1}{t}\sum_i log(-(f_1(x)_i - y)) \\ & - \frac{1}{t}\sum_i log(-(f_2(x)_i - y)) \\ & - \frac{1}{t}\sum_i log(-(f_3(x)_i - y))\end{aligned}$$

$$\begin{aligned}\min_{y,x}\ & f = tf_0 - \sum_i log(-(f_1(x)_i - y)) \\ & - \sum_i log(-(f_2(x)_i - y)) \\ & - \sum_i log(-(f_3(x)_i - y)) \\ s.t.\ & A^+x = s\end{aligned}$$

  Initialize x for feasibility wrt. inequality constraints:

$$(\forall i)x_i^{(0)} = 0.1$$

  where inequalities are:

$$\begin{aligned}0 \leq x^{(0)} &\leq c = \mathbf{1} \\ Bx^{(0)} = \begin{bmatrix}0.2\\0.2\\0.3\end{bmatrix} &\leq b = \mathbf{1}\end{aligned}$$

Gradient:

$$\frac{\partial f}{\partial x_i} = t\left(\frac{1}{c_i - x_i} + \frac{x_i}{(c_i - x_i)^2}\right) + \frac{1}{-x_i + c_i} - \frac{1}{x_i} - \sum_{j=1}^{3}\left(\frac{B_{ji}}{B_{[j,:]}x - b_j}\right)$$

$$\nabla f(x,t) = \begin{bmatrix} \frac{\partial f}{\partial x_1} \\ .. \\ \frac{\partial f}{\partial x_L} \end{bmatrix}$$

Hessian:

$$\frac{\partial f^2}{\partial x_i x_j} = \mathbf{1}_{i=j}\left(t(\frac{2}{(c_i - x_i)^2} + \frac{2x_i}{(c_i - x_i)^3}) + \frac{1}{(x_i - c_i)^2} + \frac{1}{x_i^2}\right) + \sum_{k=1}^{3}\frac{B_{ki}B_{kj}}{(B_{[k,:]}x - b_k)^2}$$

$$\nabla^2 f(x,t) = \begin{bmatrix} \frac{\partial f^2}{\partial x_1^2} & \frac{\partial f^2}{\partial x_1 \partial x_2} & .. \\ .. & .. & .. \\ .. & \frac{\partial f^2}{\partial x_L \partial x_{L-1}} & \frac{\partial f^2}{\partial x_L^2} \end{bmatrix}$$

KKT System of Infeasible Start Newton Method for solving step direction:

$$\begin{bmatrix} \nabla^2 f(x,t) & A^{+T} \\ A^+ & 0 \end{bmatrix}\begin{bmatrix} \Delta x \\ v \end{bmatrix} = \begin{bmatrix} -\nabla f(x,t) \\ s - A^+ x \end{bmatrix}$$

Constants:
Outer loop: $t_0 = 1.0, t_{i+1} = \mu t_i, \mu = 2$
Inner loop line search: $\alpha = 0.4, \beta = 0.95, h_{i+1} = h_i * \beta$

Stopping Criterion:
Line Search: $\|(r_{prim}, r_{dual})_{i+1}\| \leq (1 - \alpha h)\|(r_{prim}, r_{dual})_i\|$
Inner loop: $residual_{prim} < 1 * 10^{-12} \wedge residual_{dual} < 1 * 10^{-12}$
Outer loop: $\frac{m}{t} < 1 * 10^{-12}$

Residuals:
$res_{pri}^{next} = A^+(x + \Delta x) - s$
$res_{dual}^{next} = \nabla f(x,t) + A^{+T} v$

Solution:

Minimal delay: 26.427066103
Flows:

$$x = \begin{bmatrix} 3.56069375 * 10^{-01} \\ 4.87861250 * 10^{-01} \\ 3.56069375 * 10^{-01} \\ 1.55128906 * 10^{-01} \\ 1.55128906 * 10^{-01} \\ 8.00940469 * 10^{-01} \\ 7.98119062 * 10^{-01} \\ 8.00940469 * 10^{-01} \\ 6.68561545 * 10^{-14} \\ 6.68561545 * 10^{-14} \\ 8.00940469 * 10^{-01} \\ 7.98119062 * 10^{-01} \\ 8.00940469 * 10^{-01} \end{bmatrix}$$

- c) Produce a plot of delay vs. the outer-loop iterations (i.e., every time you update t in the barrier function).

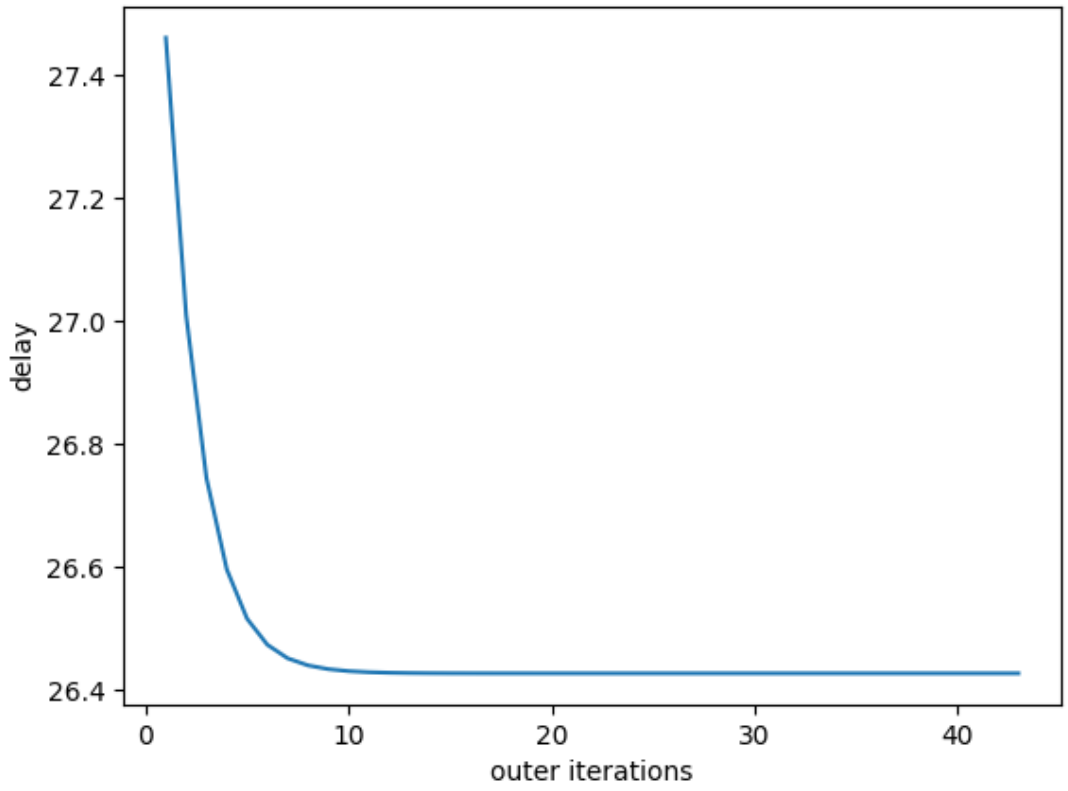

Figure 1: Delay vs. Outer Iterations

- d) Produce a plot of residue (defined as the sum of the norm of residue of the infeasible-start Newtons method and the residue associated with each outer iteration in the interior-point method, namely $\frac{m}{t}$, i.e., residue = $\|r\|_2+\frac{m}{t}$) in log scale vs. the number of inner-loop iterations (i.e., every time you update the flow rate variables).

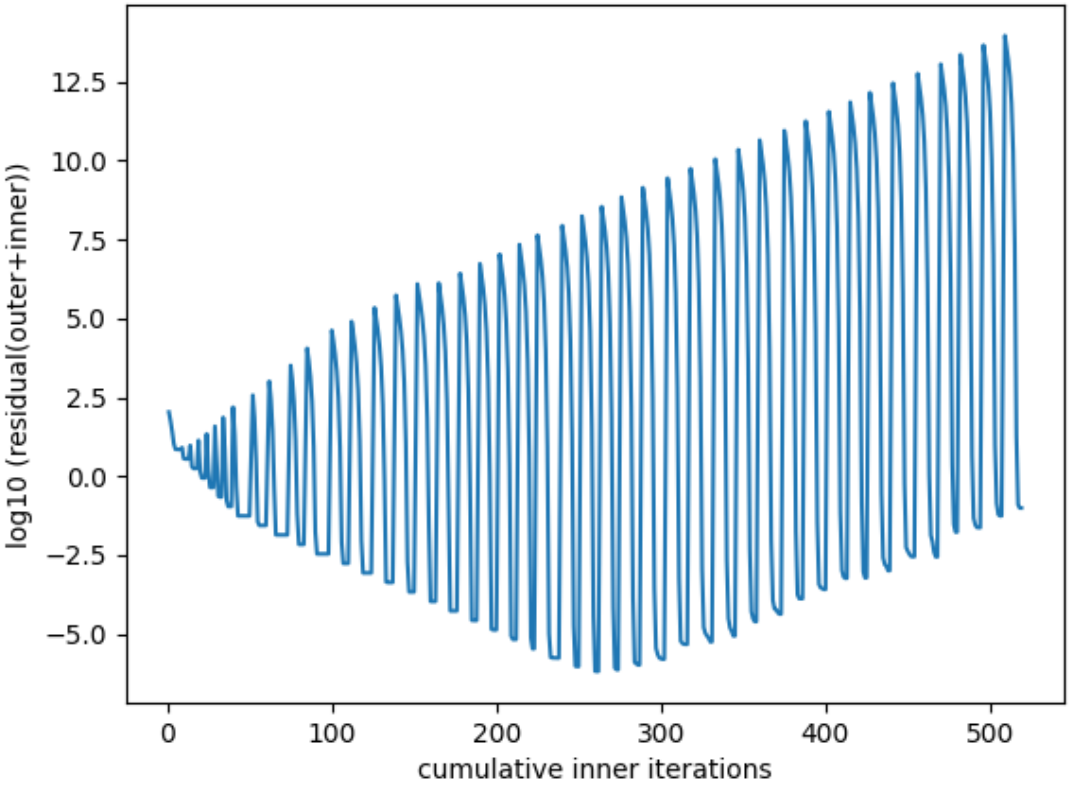

Figure 2: Inner + Outer Residual vs. Cumulative Inner Iterations

- Solver code:

```
import cvxpy as cp
import numpy as np
import matplotlib.pylab as pl
import scipy.sparse as sps
import seaborn as sns
from scipy.io import loadmat
from os.path import dirname, join as pjoin
import numpy.linalg as linalg
from scipy import linalg as scipy_linalg

def objective(t, A, B, x, c, b, s):
    obj = x/(c-x)
    return np.sum(obj)

def grad(t, A, B, x, c, b, s):

    gradient = (t * (1.0/(c-x) + x/((c-x)**2))
                + 1.0/(-x+c)
                - 1.0/x
                - 1.0 * (((B[0,:].T)/(B[0,:].dot(x)-b[0,0]) +
                          (B[1,:].T)/(B[1,:].dot(x)-b[1,0]) +
                          (B[2,:].T)/(B[2,:].dot(x)-b[2,0]) )[..., np.newaxis]))

    assert(gradient.shape==(x.size,1))

    return gradient

def residual_prim(t, A, B, x, c, b, s, v):
    return A.dot(x)-s

def residual_dual(t, A, B, x, c, b, s, v):
    return grad(t, A, B, x, c, b, s) + (A.T).dot(v)

def kkt_rhs(t, A, B, x, c, b, s):

    L = x.size
    r = np.zeros((L+A.shape[0], 1))

    gradient = grad(t, A, B, x, c, b, s)

    r[0:x.size,:] = -1.0 * gradient
    r[x.size:,:] = s - A.dot(x)

    return r
```

```
def init_point(A, B, x, c, b, s):

    #manually init x to be feasible
    x[:,] = 0.1

    return x

def hessian(t, A, B, x, c, b, s):

    m = np.zeros((x.size, x.size))

    m[np.diag_indices_from(m)] = ( t * (2.0/((c-x)**2) + 2.0*x/((c-x)**3))
                                   + 1.0/np.power(x-c,2)
                                   + 1.0/np.power(x,2) )[:,0]

    denom = np.power(B.dot(x)-b, 2)

    for i in range(0, x.size):
        for j in range(0, x.size):
            m[i,j] +=  1.0 * ( (B[0,i]*B[0,j])/denom[0] +
                               (B[1,i]*B[1,j])/denom[1] +
                               (B[2,i]*B[2,j])/denom[2] )
    return m

def kkt_matrix(t, A, B, x, c, b, s):

    m = np.zeros((x.size + A.shape[0], x.size + A.shape[0]))

    m[0:x.size, 0:x.size] = hessian(t, A, B, x, c, b, s)

    m[x.size:x.size+A.shape[0], 0:A.shape[1]] = A
    m[0:A.shape[1], x.size:x.size+A.shape[0]] = A.T

    return m

def formulate():

    N = 8
    L = 13

    A = np.zeros((N-1,L))
    B = np.zeros((3,L))
    x = np.zeros((L,1)) #to be initialized later
    c = np.zeros((L,1))
    b = np.zeros((3,1))
    s = np.zeros((N-1,1))
```

```
#node1, links: out: 1,2,3, in:
A[0,0] = 1.
A[0,1] = 1.
A[0,2] = 1.

#node2, links: out: 4,6, in: 1
A[1,3] = 1.
A[1,5] = 1.
A[1,0] = -1.

#node3, links: out: 5,8, in: 3
A[2,4] = 1.
A[2,7] = 1.
A[2,2] = -1.

#node4, links: out: 7, in: 2,4,5
A[3,6] = 1.
A[3,1] = -1.
A[3,3] = -1.
A[3,4] = -1.

#node5, links: out: 9,10,12, in: 7
A[4,8] = 1.
A[4,9] = 1.
A[4,11] = 1.
A[4,6] = -1.

#node6, links: out: 11, in: 6,9
A[5,10] = 1.
A[5,5] = -1.
A[5,8] = -1.

#node7, links: out: 13, in: 8,10
A[6,12] = 1.
A[6,7] = -1.
A[6,9] = -1.

B[0,3] = 1.
B[0,5] = 1.

B[1,4] = 1.
B[1,7] = 1.

B[2,8] = 1.
B[2,9] = 1.
B[2,11] = 1.
```

```
    c[:,0] = 1.

    b[:,0] = 1.

    s[0,0] = 1.2
    s[1,0] = 0.6
    s[2,0] = 0.6
    s[3:,0] = 0.

    return [A, B, x, c, b, s]

def solve_kkt(t, A, B, x, c, b, s):
    kkt_m = kkt_matrix(t, A, B, x, c, b, s)
    res = kkt_rhs(t, A, B, x, c, b, s)
    # ax = sns.heatmap(kkt_m)
    # pl.show()
    return scipy_linalg.solve(kkt_m, res, assume_a='sym')

def solve_inner(t, A, B, x, c, b, s, v,
                loop_outer, loop_inner_accum,
                record_residual, outer_residual):

    eps1 = 1e-12
    eps2 = 1e-12

    first_iter = False

    loop_inner = 0

    max_iter = 100

    y_prev = None

    while True:
        loop_inner += 1
        print("loop inner", loop_inner)
        y = solve_kkt(t, A, B, x, c, b, s)

        if y_prev is not None and linalg.norm(y-y_prev) < 1e-15:
            break

        y_prev = y

        delta_x = y[0:x.size,:]
        v_new = y[x.size:,:]
        if v is None:
            first_iter = True
```

```
    v = v_new
    print("first_iter")
delta_v = v_new - v

#backtracking line search
beta = 0.95
alpha = 0.4
h = 1.0

loopc = 0
res_inner = None

while True:
    loopc+= 1
    assert(v.shape == delta_v.shape)

    res_prim_next = residual_prim(t, A, B, x+h*delta_x,
                                  c, b, s, v+h*delta_v)
    res_prim_cur = residual_prim(t, A, B, x, c, b, s, v)

    res_dual_phase1_next = residual_dual(t, A, B,
                                         x+h*delta_x,
                                         c, b, s,
                                         v+h*delta_v)
    res_dual_phase1_cur = residual_dual(t, A, B, x, c,
                                        b, s, v)

    r1 = np.concatenate((res_prim_next,res_dual_phase1_next), axis=0)
    r2 = np.concatenate((res_prim_cur,res_dual_phase1_cur), axis=0)

    res_inner = linalg.norm(r1)

    if res_inner <= (1-alpha*h)*linalg.norm(r2):
        break

    h = beta * h

x = x + h * delta_x
v = v + h * delta_v

record_residual[0].append(loop_inner_accum + loop_inner)
record_residual[1].append(res_inner+outer_residual)

res_prim_cur = residual_prim(t, A, B, x, c, b, s, v)
res_dual_cur = residual_dual(t, A, B, x, c, b, s, v)

if linalg.norm(res_prim_cur,2) <= eps1 and
```

```
            np.linalg.norm(res_dual_cur,2) <= eps2:
                break
        if loop_inner > max_iter:
            break

    return x, v, objective(t, A, B, x, c, b, s),
           loop_inner_accum + loop_inner, record_residual

def solve(A, B, x, c, b, s):

    rec_res = ([],[])
    record_delay_vs_outer_iter = ([],[])

    x = init_point(A, B, x, c, b, s)

    #sanity check
    assert(np.all(B.dot(x) <= b))
    assert(np.all(x<=c))
    assert(np.all(x>=0))

    t = 1.0
    mu = 2.0

    m = A.shape[0]
    eps = 1e-12
    v = None

    loop_outer = 0
    loop_inner_accum = 0

    while True:
        loop_outer += 1
        x, v, obj, loop_inner_accum, rec_res = solve_inner(t, A, B, x, c,
                                                           b, s, v,
                                                           loop_outer,
                                                           loop_inner_accum,
                                                           rec_res, m/t )
        if m/t <= eps:
            break
        t = mu * t

        record_delay_vs_outer_iter[0].append(loop_outer)
        record_delay_vs_outer_iter[1].append(obj)

        if loop_inner_accum > 5000:
            break
```

```
    return x, obj, record_delay_vs_outer_iter, rec_res

prob = formulate()
A, B, x, c, b, s = prob
solution, objective, record1, record2 = solve(*prob)

print("objective achieved: ", objective)
print("solution: ", solution)

#sanity check
assert(np.all(np.abs(A.dot(solution)-s)<1e-14))
assert(np.all(B.dot(solution) <= b))
assert(np.all(solution<=c))
assert(np.all(solution>=0))

#delay vs outer iterations
# pl.plot(record1[0],record1[1])
pl.ylabel('delay')
pl.xlabel('outer iterations')
# pl.show()
pl.savefig('../imgs/delay_vs_outer.png')

#residual (sum of norm of residual of infeasible start
#newton method and outer iteration residual): ||r||+m/t

res = np.array(record2[1])
assert(np.all(res>=0))

log_vals = np.log10(res)

pl.plot(record2[0], log_vals)
pl.ylabel('log10 (residual(outer+inner))')
pl.xlabel('cumulative inner iterations')
# pl.show()
pl.savefig('../imgs/res_vs_inner_iter.png')
```