Power Assignment in a Wireless Communication System

Fork me on GitHub

by Robert Gowers, Roger Hill, Sami Al-Izzi, Timothy Pollington and Keith Briggs

from Boyd and Vandenberghe, Convex Optimization, exercise 4.20 page 196

Convex optimization can be used to maximise the minimum signal to inteference plus noise ratio (SINR) of a wireless communication system. Consider a system with n transmitters, each with power $p_j \geq 0$, transmitting to n receivers. Let $G_{ij} \geq 0$ denote the path gain from transmitter j to receiver i. These path gains form the matrix $G \in \mathbb{R}^{n \times n}$.

Each receiver is assigned to a transmitter such that the signal power at receiver i, $S_i = G_{ii}p_i$ and the interefence power at receiver i is $I_i = \sum_{k \neq i} G_{ik}p_k$. Given a noise power σ_i at each receiver, the SINR at receiver i, $\gamma_i = \frac{S_i}{I_i + \sigma_i}$.

The objective is to maximise the minimum SINR of the system under certain power constraints. These constraints are:

i - Each transmitter power $p_j \leq P_j^{\max}$

ii - If the transmitters are partitioned into m nonoverlapping groups, $K_1, \ldots, K_m$, which share a common power supply with total power P_l^{gp}: $\sum_{k \in K_l} p_k \leq P_l^{\text{gp}}$.

iii - There is a maximum power that each receiver can receive P_i^{rc}, $\sum_{k=1}^{n} G_{ik}p_k \leq P_i^{\text{rc}}$.

The objective function can be rewritten as:

minimise $\max_{i=1,\ldots,n} \frac{I_i + \sigma_i}{S_i}$

However, since this is a quasiconvex objective function we cannot solve it directly using CVXPY. Instead we must use a bisection method. First we take the step of rewriting the objective, $\alpha = \gamma^{-1} \geq 0$, as a constraint:

$$I_i + \sigma_i \leq S_i\alpha$$

Then we choose initial lower and upper bounds L_0 and U_0 for α, which should be chosen such that $L < \alpha^* < U$, where α^* is the optimal value of α. Starting with an initial value $\alpha_0 = \frac{1}{2}(L_0 + U_0)$, feasibility is checked for α_0 by using an arbitrary objective function. The new upper and lower bounds are determined from the feasibility:

If α_0 is feasible then $L_1 = L_0, U_1 = \alpha_0$ and $\alpha_1 = \frac{1}{2}(L_1 + U_1)$.

If α_0 is infeasible then $L_1 = \alpha_1, U_1 = U_0$ and $\alpha_1 = \frac{1}{2}(L_1 + U_1)$.

This bisection process is repeated until $U_N - L_N < \epsilon$, where ϵ is the desired tolerance.

```
#!/usr/bin/env python3
# @author: R. Gowers, S. Al-Izzi, T. Pollington, R. Hill & K. Briggs

import cvxpy as cp
import numpy as np


def maxmin_sinr(G, P_max, P_received, sigma, Group, Group_max, epsilon = 0.001):
    # find n and m from the size of the path gain matrix
    n, m = np.shape(G)

    # Checks sizes of inputs
    if m != np.size(P_max):
        print('Error: P_max dimensions do not match gain matrix dimensions\n')
        return 'Error: P_max dimensions do not match gain matrix dimensions\n', np.r

    if n != np.size(P_received):
```

```
        print('Error: P_received dimensions do not match gain matrix dimensions\n')
        return 'Error: P_received dimensions do not match gain matrix dimensions', r

    if n != np.size(sigma):
        print('Error: σ dimensions do not match gain matrix dimensions\n')
        return 'Error: σ dimensions do not match gain matrix dimensions', np.nan, np

    #I = np.zeros((n,m))
    #S = np.zeros((n,m))

    delta = np.identity(n)
    S = G*delta # signal power matrix
    I = G-S # interference power matrix

    # group matrix: number of groups by number of transmitters
    num_groups = int(np.size(Group,0))

    if num_groups != np.size(Group_max):
        print('Error: Number of groups from Group matrix does not match dimensions c
        return ('Error: Number of groups from Group matrix does not match dimensions
                np.nan, np.nan, np.nan, np.nan)

    # normalising the max power of a group so it is in the range [0,1]
    Group_norm = Group/np.sum(Group,axis=1).reshape((num_groups,1))

    # create scalar optimisation variable p: the power of the n transmitters
    p = cp.Variable(shape=n)
    best = np.zeros(n)

    # set upper and lower bounds for sub-level set
    u = 1e4
    l = 0

    # alpha defines the sub-level sets of the generalised linear fractional problem
    # in this case α is the reciprocal of the minimum SINR
    alpha = cp.Parameter(shape=1)
```

```
    # set up the constraints for the bisection feasibility test
    constraints = [I*p + sigma <= alpha*S*p, p <= P_max, p >= 0, G*p <= P_received,

    # define objective function, in our case it's constant as only want to test the
    obj = cp.Minimize(alpha)

    # now check whether the solution lies between u and l
    alpha.value = [u]
    prob = cp.Problem(obj, constraints)
    prob.solve()

    if prob.status != 'optimal':
        # in this case the level set u is below the solution
        print('No optimal solution within bounds\n')
        return 'Error: no optimal solution within bounds', np.nan, np.nan, np.nan

    alpha.value = [l]
    prob = cp.Problem(obj, constraints)
    prob.solve()

    if prob.status == 'optimal':
        # in this case the level set l is below the solution
        print('No optimal solution within bounds\n')
        return 'Error: no optimal solution within bounds', np.nan, np.nan, np.nan

    # Bisection algortithm starts
    maxLoop = int(1e7)
    for i in range(1,maxLoop):
        # First check that u is in the feasible domain and l is not, loop finishes h
        # set α as the midpoint of the interval
        alpha.value = np.atleast_1d((u + l)/2.0)

        # test the size of the interval against the specified tolerance
        if u-l <= epsilon:
            break
```

```
        # form and solve problem
        prob = cp.Problem(obj, constraints)
        prob.solve()

        # If the problem is feasible u -> α, if not l -> α, best takes the last feas
        # when the tolerance is reached the new α may be out of bounds
        if prob.status == 'optimal':
            u = alpha.value
            best = p.value
        else:
            l = alpha.value

        # final condition to check that the interval has converged to order ε, i.e.
        if u - l > epsilon and i == (maxLoop-1):
            print("Solution not converged to order epsilon")

    return l, u, float(alpha.value), best
```

Example

As a simple example, we will consider a case with $n = 5$, where $G_{ij} = 0.6$ if $i = j$ and 0.1 otherwise.

$P_j^{\max} = 1$ for all transmitters and the transmitters are split into two groups, each with $P_l^{\mathrm{gp}} = 1.8$. The first group contains transmitters 1 & 2, while the second group contains 3,4 & 5.

For all receivers $P_i^{\mathrm{rc}} = 4$ and $\sigma_i = 0.1$.

```
np.set_printoptions(precision=3)

# in this case we will use a gain matrix with a signal weight of 0.6 and interferenc
G = np.array([[0.6,0.1,0.1,0.1,0.1],
              [0.1,0.6,0.1,0.1,0.1],
```

```
              [0.1,0.1,0.6,0.1,0.1],
              [0.1,0.1,0.1,0.6,0.1],
              [0.1,0.1,0.1,0.1,0.6]])

# in this case m=n, but this generalises if we want n receivers and m transmitters
n, m = np.shape(G)

# set maximum power of each transmitter and receiver saturation level
P_max = np.array([1.]*n)

# normalised received power, total possible would be all power from all transmitters
P_received = np.array([4.,4.,4.,4.,4.])/n

# set noise level
sigma = np.array([0.1,0.1,0.1,0.1,0.1])

# group matrix: number of groups by number of transmitters
Group = np.array([[1.,1.,0,0,0],[0,0,1.,1.,1.]])

# max normalised power for groups, number of groups by 1
Group_max = np.array([1.8,1.8])

# now run the optimisation problem
l, u, alpha, best = maxmin_sinr(G, P_max, P_received, sigma, Group, Group_max)

print('Minimum SINR={:.4g}'.format(1/alpha))
print('Power={}'.format(best))
```

```
Minimum SINR=1.148
Power=[0.8 0.8 0.8 0.8 0.8]
```