EE364a: Convex Optimization I S. Boyd
March 11–12 or March 12–13, 2016

Final Exam Solutions

This is a 24-hour take-home final. Please turn it in at Bytes Cafe in the Packard building, 24 hours after you pick it up.

You may use any books, notes, or computer programs, but you may not discuss the exam with anyone until March 16, after everyone has taken the exam. The only exception is that you can ask us for clarification, via the course staff email address. We've tried pretty hard to make the exam unambiguous and clear, so we're unlikely to say much.

Please make a copy of your exam, or scan it, before handing it in.

Please attach the cover page to the front of your exam. Assemble your solutions in order (problem 1, problem 2, problem 3, ...), starting a new page for each problem. Put everything associated with each problem (*e.g.*, text, code, plots) together; do not attach code or plots at the end of the final.

We will deduct points from long needlessly complex solutions, even if they are correct. Our solutions are not long, so if you find that your solution to a problem goes on and on for many pages, you should try to figure out a simpler one. We expect neat, legible exams from everyone, including those enrolled Cr/N.

When a problem involves computation you must give all of the following: a clear discussion and justification of exactly what you did, the source code that produces the result, and the final numerical results or plots.

Files containing problem data can be found in the usual place,

`http://www.stanford.edu/~boyd/cvxbook/cvxbook_additional_exercises/`

Please respect the honor code. Although we allow you to work on homework assignments in small groups, you cannot discuss the final with anyone, at least until everyone has taken it.

All problems have equal weight. Some are easy. Others, not so much.

Be sure you are using the most recent version of CVX, CVXPY, or Convex.jl. Check your email often during the exam, just in case we need to send out an important announcement.

Some problems involve applications. But you do not need to know *anything* about the problem area to solve the problem; the problem statement contains everything you need.

1. *Portfolio optimization using multiple risk models.* Let $w \in \mathbf{R}^n$ be a vector of portfolio weights, where negative values correspond to short positions, and the weights are normalized such that $\mathbf{1}^T w = 1$. The expected return of the portfolio is $\mu^T w$, where $\mu \in \mathbf{R}^n$ is the (known) vector of expected asset returns. As usual we measure the risk of the portfolio using the variance of the portfolio return. However, in this problem we do not know the covariance matrix Σ of the asset returns; instead we assume that Σ is one of M (known) covariance matrices $\Sigma^{(k)} \in \mathbf{S}^n_{++}$, $k = 1, \ldots, M$. We can think of the $\Sigma^{(k)}$ as representing M different risk models, associated with M different market regimes (say). For a weight vector w, there are M different possible values of the risk: $w^T \Sigma^{(k)} w$, $k = 1, \ldots, M$. The worst-case risk, across the different models, is given by $\max_{k=1,\ldots,M} w^T \Sigma^{(k)} w$. (This is the same as the worst-case risk over all covariance matrices in the convex hull of $\Sigma^{(1)}, \ldots, \Sigma^{(M)}$.)

   We will choose the portfolio weights in order to maximize the expected return, adjusted by the worst-case risk, *i.e.*, as the solution $w^\star$ of the problem

$$\begin{array}{ll} \text{maximize} & \mu^T w - \gamma \max_{k=1,\ldots,M} w^T \Sigma^{(k)} w \\ \text{subject to} & \mathbf{1}^T w = 1, \end{array}$$

   with variable w, where $\gamma > 0$ is a given risk-aversion parameter. We call this the mean-worst-case-risk portfolio problem.

   (a) Show that there exist $\gamma_1, \ldots, \gamma_M \geq 0$ such that $\sum_{k=1}^M \gamma_k = \gamma$ and the solution $w^\star$ of the mean-worst-case-risk portfolio problem is also the solution of the problem

$$\begin{array}{ll} \text{maximize} & \mu^T w - \sum_{k=1}^M \gamma_k w^T \Sigma^{(k)} w \\ \text{subject to} & \mathbf{1}^T w = 1, \end{array}$$

   with variable w.

   Remark. The result above has a beautiful interpretation: We can think of the γ_k as allocating our total risk aversion γ in the mean-worst-case-risk portfolio problem across the M different regimes.

   Hint. The values γ_k are not easy to find: you have to solve the mean-worst-case-risk problem to get them. Thus, this result does not help us solve the mean-worst-case-risk problem; it simply gives a nice interpretation of its solution.

   (b) Find the optimal portfolio weights for the problem instance with data given in `multi_risk_portfolio_data.*`. Report the weights and the values of γ_k, $k = 1, \ldots, M$. Give the M possible values of the risk associated with your weights, and the worst-case risk.

Solution.

(a) We can reformulate the mean-worst-case-risk portfolio problem as

$$\begin{array}{ll} \text{minimize} & -\mu^T w + \gamma t \\ \text{subject to} & w^T \Sigma^{(k)} w \leq t, \quad k = 1, ..., M, \\ & \mathbf{1}^T w = 1. \end{array}$$

with variables $w \in \mathbf{R}^n$ and $t \in \mathbf{R}$. Let λ_k be a dual variable for the kth inequality constraint, and ν be a dual variable for the equality constraint. The KKT conditions are then

$$\begin{aligned} -\mu + \nu\mathbf{1} + \textstyle\sum_k 2\lambda_k \Sigma^{(k)} w &= 0 \\ \gamma - \textstyle\sum_k \lambda_k &= 0 \\ \mathbf{1}^T w &= 1 \\ w^T \Sigma^{(k)} w &\leq t \\ \lambda_k \left(w^T \Sigma^{(k)} w - t \right) &= 0, \quad k = 1, ..., M \\ \gamma &\succeq 0. \end{aligned}$$

Similarly, the KKT conditions for the problem

$$\begin{array}{ll} \text{maximize} & \mu^T w - \sum_{k=1}^M \gamma_k w^T \Sigma^{(k)} w \\ \text{subject to} & \mathbf{1}^T w = 1 \end{array} \tag{1}$$

are

$$\begin{aligned} -\mu + \alpha\mathbf{1} + \textstyle\sum_k 2\gamma_k \Sigma^{(k)} w &= 0 \\ \mathbf{1}^T w &= 1, \end{aligned} \tag{2}$$

where α is a dual variable for the equality constraint. Let $(w^\star, t^\star, \nu^\star, \lambda^\star)$ be optimal for the mean-worst-case-risk portfolio problem, and choose $\gamma_k = \lambda_k^\star$, $k = 1, \ldots, M$. With this choice of the γ_k, we have that $\sum_k \gamma_k = \sum_k \lambda_k^\star = \gamma$ from the optimality conditions for the mean-worst-case-risk portfolio problem. Moreover, $(w, \alpha) = (w^\star, \nu^\star)$ satisfy (2), so $w^\star$ is optimal for (1).

(b) The results are

```
weights:
0.42474
0.66427
-0.11469
1.38055
1.42423
-1.52706
-0.61401
-0.49879
-0.25407
0.11484

gamma_k values:
0.29232
0.00000
0.00000
0.46580
0.14230
```

```
0.09958

risk values:
0.12188
0.08454
0.08247
0.12188
0.12188
0.12188

worst case risk:
0.12188
```

The following Matlab code solves the problem.

```
clear all; clc
multi_risk_portfolio_data;

cvx_begin quiet
    variables t w(n)
    dual variables gamma_dual{M}
    maximize(mu*w - gamma*t)
    subject to
        gamma_dual{1}: w'*Sigma_1*w <= t
        gamma_dual{2}: w'*Sigma_2*w <= t
        gamma_dual{3}: w'*Sigma_3*w <= t
        gamma_dual{4}: w'*Sigma_4*w <= t
        gamma_dual{5}: w'*Sigma_5*w <= t
        gamma_dual{6}: w'*Sigma_6*w <= t
        sum(w) == 1
cvx_end

% weights
w

% dual variables
disp('gamma_k values');
disp(gamma_dual{1});
disp(gamma_dual{2});
disp(gamma_dual{3});
disp(gamma_dual{4});
disp(gamma_dual{5});
disp(gamma_dual{6});
```

```
% risk values
disp('risk values');
disp(w'*Sigma_1*w);
disp(w'*Sigma_2*w);
disp(w'*Sigma_3*w);
disp(w'*Sigma_4*w);
disp(w'*Sigma_5*w);
disp(w'*Sigma_6*w);

% worst case risk
t
```

The following Python code solves the problem.

```
import numpy as np
import cvxpy as cvx

from multi_risk_portfolio_data import *

w = cvx.Variable(n)
t = cvx.Variable()
risks = [cvx.quad_form(w, Sigma) for Sigma in
        (Sigma_1, Sigma_2, Sigma_3, Sigma_4, Sigma_5, Sigma_6)]
risk_constraints = [risk <= t for risk in risks]
prob = cvx.Problem(cvx.Maximize(w.T*mu - gamma * t),
                   risk_constraints + [cvx.sum_entries(w) == 1])
prob.solve()

print('\nweights:')
print('\n'.join(['{:.5f}'.format(weight) for weight in w.value.A1]))

print('\ngamma_k values:')
print('\n'.join(['{:.5f}'.format(risk.dual_value)
                 for risk in risk_constraints]))

print('\nrisk values:')
print('\n'.join(['{:.5f}'.format(risk.value) for risk in risks]))

print('\nworst case risk:\n{:.5f}'.format(t.value))
```

The following Julia code solves the problem.

```
using Convex, SCS
set_default_solver(SCSSolver(verbose=false))
```

```
include("multi_risk_portfolio_data.jl");

w = Variable(n)
t = Variable()

risk_1 = quadform(w, Sigma_1)
risk_2 = quadform(w, Sigma_2)
risk_3 = quadform(w, Sigma_3)
risk_4 = quadform(w, Sigma_4)
risk_5 = quadform(w, Sigma_5)
risk_6 = quadform(w, Sigma_6)

p = maximize(mu*w - gamma * t, [sum(w) == 1])

p.constraints += risk_1 <= t
p.constraints += risk_2 <= t
p.constraints += risk_3 <= t
p.constraints += risk_4 <= t
p.constraints += risk_5 <= t
p.constraints += risk_6 <= t;

solve!(p)

println("\nweights:")
println(round(w.value, 5))

println("\ngamma_k values:")
for i = 2:7
    println(round(p.constraints[i].dual,5))
end

println("\nrisk values:")
for sigma in (Sigma_1, Sigma_2, Sigma_3, Sigma_4, Sigma_5, Sigma_6)
    println(round(w.value'*sigma*w.value,5))
end

println("\nworst case risk:")
println(round(t.value, 5))
```

2. *Minimum possible maximum correlation.* Let Z be a random variable taking values in $\mathbf{R}^n$, and let $\Sigma \in \mathbf{S}^n_{++}$ be its covariance matrix. We do not know Σ, but we do know the variance of m linear functions of Z. Specifically, we are given nonzero vectors $a_1, \ldots, a_m \in \mathbf{R}^n$ and $\sigma_1, \ldots, \sigma_m > 0$ for which

$$\mathbf{var}(a_i^T Z) = \sigma_i^2, \quad i = 1, \ldots, m.$$

For $i \neq j$ the correlation of Z_i and Z_j is defined to be

$$\rho_{ij} = \frac{\Sigma_{ij}}{\sqrt{\Sigma_{ii}\Sigma_{jj}}}.$$

Let $\rho^{\max} = \max_{i \neq j} |\rho_{ij}|$ be the maximum (absolute value) of the correlation among entries of Z. If $\rho^{\max}$ is large, then at least two components of Z are highly correlated (or anticorrelated).

(a) Explain how to find the smallest value of $\rho^{\max}$ that is consistent with the given information, using convex or quasiconvex optimization. If your formulation involves a change of variables or other transformation, justify it.

(b) The file `correlation_bounds_data.*` contains $\sigma_1, \ldots, \sigma_m$ and the matrix A with columns $a_1, \ldots, a_m$. Find the minimum value of $\rho^{\max}$ that is consistent with this data. Report your minimum value of $\rho^{\max}$, and give a corresponding covariance matrix Σ that achieves this value. You can report the minimum value of $\rho^{\max}$ to an accuracy of 0.01.

Solution.

(a) Using the formula for the variance of a linear function of a random variable, we have that

$$\mathbf{var}(a_i^T Z) = a_i^T \, \mathbf{var}(Z) a_i = a_i^T \Sigma a_i,$$

which is a linear function of Σ. We can find the minimum value of the maximum correlation among components of Z that is consistent with the data by solving the following optimization problem.

$$\begin{array}{ll} \text{minimize} & \max_{i \neq j} |\rho_{ij}| \\ \text{subject to} & a_i^T \Sigma a_i = \sigma_i^2, \quad i = 1, \ldots, m \\ & \Sigma \succeq 0. \end{array}$$

Observe that

$$|\rho_{ij}| = \frac{|\Sigma_{ij}|}{\sqrt{\Sigma_{ii}\Sigma_{jj}}}$$

is a quasiconvex function of Σ: the numerator is a nonnegative convex function of Σ, and the denominator is a positive concave function of Σ (it is the composition of the geometric mean and a linear function of Σ). Since the maximum of quasiconvex functions is quasiconvex, the objective in the optimization problem above

is quasiconvex. Thus, we can find the smallest value of $\rho^{\max}$ by solving a quasiconvex optimization problem. In particular, we have that $\rho^{\max} \leq t$ is consistent with the data if and only if the following convex feasibility problem is feasible:

$$\begin{aligned} |\Sigma_{ij}| &\leq t\sqrt{\Sigma_{ii}\Sigma_{jj}}, \quad i \neq j \\ a_i^T \Sigma a_i &= \sigma_i^2, \quad i = 1, \ldots, m \\ \Sigma &\succeq 0 \end{aligned}$$

We can find the minimum value of t for which this problem is feasible using bisection search starting with $t = 0$ and $t = 1$.

(b) The following Matlab code solves the problem.

```
clear all; close all; clc
correlation_bounds_data;

% find the minimum possible maximum correlation
lb = 0;
ub = 1;
Sigma_opt = nan(n,n);
while ub-lb > 1e-3
    t = (lb+ub)/2;
    cvx_begin sdp czz
        variable Sigma(n,n) symmetric
        for i = 1:(n-1)
            for j = (i+1):n
                abs(Sigma(i,j)) <= t * geo_mean([Sigma(i,i), Sigma(j,j)])
            end
        end
        for i = 1:m
            A(:,i)' * Sigma * A(:,i) == sigma(i)^2
        end
        Sigma >= 0
    cvx_end
    if strcmp(cvx_status, 'Solved')
        ub = t;
        Sigma_opt = Sigma;
    else
        lb = t;
    end
end

% print the results and check the correlation matrix
t
```

```
Sigma = Sigma_opt
C = diag(1./sqrt(diag(Sigma)));
R = C * Sigma * C;
rho_max = max(max(R - diag(diag(R))))
```

The following Python code solves the problem.

```
import cvxpy as cvx
from math import sqrt

from correlation_bounds_data import *

Sigma = cvx.Semidef(n)
t = cvx.Parameter(sign='positive')
rho_cons = []
for i in range(n - 1):
    for j in range(i + 1, n):
        Sij = cvx.vstack(Sigma[i, i], Sigma[j, j])
        rho_cons += [cvx.abs(Sigma[i, j]) <= t * cvx.geo_mean(Sij)]
var_cons = [A[:, i].T * Sigma * A[:, i] == sigma[i]**2
            for i in range(m)]
problem = cvx.Problem(cvx.Minimize(0), rho_cons + var_cons)

lb, ub = 0.0, 1.0
Sigma_opt = None
while ub - lb > 1e-3:
    t.value = (lb + ub) / 2
    problem.solve()
    if problem.status == cvx.OPTIMAL:
        ub = t.value
        Sigma_opt = Sigma.value
    else:
        lb = t.value

print('rho_max =', t.value)
print('Sigma =', Sigma_opt)

# compute the correlation matrix
C = np.diag([1 / sqrt(Sigma_opt[i, i]) for i in range(n)])
R = C * Sigma_opt * C
print('R =', R)
```

The following Julia code solves the problem.

```
using Convex, SCS
set_default_solver(SCSSolver(verbose=false))

include("correlation_bounds_data.jl")

lb = 0; ub = 1
t = (lb+ub)/2
Sigma_opt = []
while ub-lb > 1e-3
    t = (lb+ub)/2
    Sigma = Semidefinite(n)
    problem = satisfy()
    for i = 1:(n-1)
        for j = (i+1):n
            problem.constraints +=
            (abs(Sigma[i,j]) <= t*geomean(Sigma[i,i],Sigma[j,j]))
        end
    end
    for i = 1:m
        problem.constraints += (A[:,i]' * Sigma * A[:,i] == sigma[i]^2)
    end

    solve!(problem)
    if problem.status == :Optimal
        ub = t
        Sigma_opt = Sigma.value
    else
        lb = t
    end
end
println("t = $(round(t,4))")
println("Sigma = $(round(Sigma_opt,4))")

# compute the correlation matrix
C = diagm(Float64[1/sqrt(Sigma_opt[i,i]) for i in 1:n])
R = C * Sigma_opt * C
println("R = $(round(R,4))")
```

We find that the minimum value of the maximum correlation that is consistent

with the data is $\rho^{\max} = 0.62$. A corresponding covariance matrix is

$$\Sigma = \begin{bmatrix} 3.78 & 0.30 & 1.46 & 1.47 & 0.24 \\ 0.30 & 2.91 & -1.28 & 1.29 & 0.86 \\ 1.46 & -1.28 & 1.46 & 0.09 & 0.27 \\ 1.47 & 1.29 & 0.09 & 1.48 & 0.49 \\ 0.24 & 0.09 & 0.27 & 0.49 & 0.42 \end{bmatrix}.$$

Although we did not ask you to do so, it is a good idea to compute the correlation matrix to this value of Σ, and check that the maximum correlation is equal to $\rho^{\max}$:

$$R = \begin{bmatrix} 1.00 & 0.09 & 0.62 & 0.62 & 0.19 \\ 0.09 & 1.00 & -0.62 & 0.62 & 0.08 \\ 0.62 & -0.62 & 1.00 & 0.06 & 0.34 \\ 0.62 & 0.62 & 0.06 & 1.00 & 0.62 \\ 0.19 & 0.08 & 0.34 & 0.62 & 1.00 \end{bmatrix}.$$

3. *Bandlimited signal recovery from zero-crossings.* Let $y \in \mathbf{R}^n$ denote a *bandlimited* signal, which means that it can be expressed as a linear combination of sinusoids with frequencies in a band:

$$y_t = \sum_{j=1}^{B} a_j \cos\left(\frac{2\pi}{n}(f_{\min} + j - 1)t\right) + b_j \sin\left(\frac{2\pi}{n}(f_{\min} + j - 1)t\right), \quad t = 1, \ldots, n,$$

where $f_{\min}$ is lowest frequency in the band, B is the bandwidth, and $a, b \in \mathbf{R}^B$ are the cosine and sine coefficients, respectively. We are given $f_{\min}$ and B, but not the coefficients a, b or the signal y.

We do not know y, but we are given its sign $s = \text{sign}(y)$, where $s_t = 1$ if $y_t \geq 0$ and $s_t = -1$ if $y_t < 0$. (Up to a change of overall sign, this is the same as knowing the 'zero-crossings' of the signal, *i.e.*, when it changes sign. Hence the name of this problem.)

We seek an estimate $\hat{y}$ of y that is consistent with the bandlimited assumption and the given signs. Of course we cannot distinguish y and αy, where $\alpha > 0$, since both of these signals have the same sign pattern. Thus, we can only estimate y up to a positive scale factor. To normalize $\hat{y}$, we will require that $\|\hat{y}\|_1 = n$, *i.e.*, the average value of $|y_i|$ is one. Among all $\hat{y}$ that are consistent with the bandlimited assumption, the given signs, and the normalization, we choose the one that minimizes $\|\hat{y}\|_2$.

(a) Show how to find $\hat{y}$ using convex or quasiconvex optimization.

(b) Apply your method to the problem instance with data in `zero_crossings_data.*`. The data files also include the true signal y (which of course you cannot use to find $\hat{y}$). Plot $\hat{y}$ and y, and report the relative recovery error, $\|y - \hat{y}\|_2 / \|y\|_2$. Give one short sentence commenting on the quality of the recovery.

Solution.

(a) We can express our estimate as $\hat{y} = Ax$, where $x = (a, b) \in \mathbf{R}^{2B}$ is the vector of cosine and sinusoid coefficients, and we define the matrix

$$A = \begin{bmatrix} C & S \end{bmatrix} \in \mathbf{R}^{n \times 2B},$$

where $C, S \in \mathbf{R}^{n \times B}$ have entries

$$C_{tj} = \cos(2\pi(f_{\min} + j - 1)t/n), \quad S_{tj} = \sin(2\pi(f_{\min} + j - 1)t/n),$$

respectively.

To ensure that the signs of $\hat{y}$ are consistent with s, we need the constraints $s_t a_t^T x \geq 0$ for $t = 1, \ldots, n$, where $a_1^T, \ldots, a_n^T$ are the rows of A. To achieve the proper normalization, we also need the linear equality constraint $\|\hat{y}\|_1 = s^T A x = n$.

(Note that an ℓ_1-norm equality constraint is not convex in general, but here it is, since the signs are given.)

We have a convex objective and linear inequality and equality constraints, so our optimization problem is convex:

$$\begin{array}{ll} \text{minimize} & \|Ax\|_2 \\ \text{subject to} & s_t a_t^T x \geq 0, \quad t = 1, \ldots, n \\ & s^T A x = n. \end{array}$$

We get our estimate as $\hat{y} = Ax^\star$, where $x^\star$ is a solution of this problem.

One common mistake was to formulate the problem above without the normalization constraint. The (incorrect) argument was that you'd solve the problem, which is homogeneous, and then scale what you get so its ℓ_1 norm is one. This doesn't work, since the (unique) solution to the homogeneous problem is $x = 0$ (since $x = 0$ is feasible). However, this method did give numerical results far better than $x = 0$. The reason is that the solvers returned a very small x, for which Ax had the right sign. And no, that does not mean the error wasn't a bad one.

(b) The recovery error is 0.1208. This is very impressive considering how little information we were given.

The following matlab code solves the problem:

```
zero_crossings_data;

% Construct matrix A whose columns are bandlimited sinusoids
C = zeros(n,B);
S = zeros(n,B);
for j = 1:B
    C(:,j) = cos(2*pi * (f_min+j-1) * (1:n) / n);
    S(:,j) = sin(2*pi * (f_min+j-1) * (1:n) / n);
end
A = [C S];

% Minimize norm subject to L1 normalization and sign constraints
cvx_begin quiet
    variable x(2*B)
    minimize norm(A*x)
    subject to
        s .* (A*x) >= 0
        s' * (A*x) == n
cvx_end

y_hat = A*x;
```

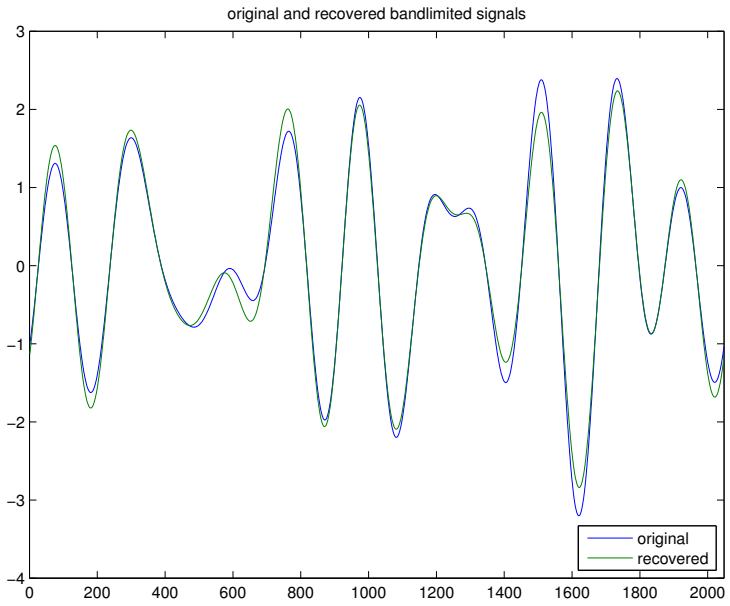

Figure 1: The original bandlimited signal y and the estimate $\hat{y}$ recovered from zero crossings.

```
fprintf('Recovery error: %f\n', norm(y - y_hat) / norm(y));
figure
plot(y)
hold all
plot(y_hat)
xlim([0,n])
legend('original', 'recovered', 'Location', 'SouthEast');
title('original and recovered bandlimited signals');
```

The following Python code solves the problem:

```
import numpy as np
import cvxpy as cvx
import matplotlib.pyplot as plt

from zero_crossings_data import *

# Construct matrix A whose columns are bandlimited sinusoids
C = np.zeros((n, B))
S = np.zeros((n, B))
for j in range(B):
```

```
    C[:, j] = np.cos(2 * np.pi * (f_min + j) * np.arange(1, n + 1) / n)
    S[:, j] = np.sin(2 * np.pi * (f_min + j) * np.arange(1, n + 1) / n)
A = np.hstack((C, S))

# Minimize norm subject to L1 normalization and sign constraints
x = cvx.Variable(2 * B)
obj = cvx.norm(A * x)
constraints = [cvx.mul_elemwise(s, A * x) >= 0,
               s.T * (A * x) == n]
problem = cvx.Problem(cvx.Minimize(obj), constraints)
problem.solve()

y_hat = np.dot(A, x.value.A1)
print('Recovery error: {}'
      .format(np.linalg.norm(y - y_hat) / np.linalg.norm(y)))
plt.figure()
plt.plot(np.arange(0, n), y, label='original');
plt.plot(np.arange(0, n), y_hat, label='recovered');
plt.xlim([0, n])
plt.legend(loc='lower left')
plt.show()
```

The following Julia code solves the problem:

```
using Convex, SCS, Gadfly
set_default_solver(SCSSolver(verbose=false))

include("zero_crossings_data.jl")

# Construct matrix A whose columns are bandlimited sinusoids
C = zeros(n, B);
S = zeros(n, B);
for j in 1:B
    C[:, j] = cos(2 * pi * (f_min + j - 1) * (1:n) / n);
    S[:, j] = sin(2 * pi * (f_min + j - 1) * (1:n) / n);
end
A = [C S];

# Minimize norm subject to L1 normalization and sign constraints
x = Variable(2 * B)
obj = norm(A * x, 2)
constraints = [s .* (A * x) >= 0,
               s' * (A * x) == n]
problem = minimize(obj, constraints)
```

```
solve!(problem)

y_hat = A * x.value
println("Recovery error: $(norm(y - y_hat) / norm(y))")
pl = plot(
  layer(x=1:n, y=y, Geom.line, Theme(default_color = colorant"blue")),
  layer(x=1:n, y=y_hat, Geom.line, Theme(default_color = colorant"green")),
);
display(pl);
```

4. *Satisfying a minimum number of constraints.* Consider the problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \leq 0 \text{ holds for at least } k \text{ values of } i, \end{array}$$

with variable $x \in \mathbf{R}^n$, where the objective f_0 and the constraint functions f_i, $i = 1, \ldots, m$ (with $m \geq k$), are convex. Here we require that only k of the constraints hold, instead of all m of them. In general this is a hard combinatorial problem; the brute force solution is to solve all $\binom{m}{k}$ convex problems obtained by choosing subsets of k constraints to impose, and selecting one with smallest objective value.

In this problem we explore a convex restriction that can be an effective heuristic for the problem.

(a) Suppose $\lambda > 0$. Show that the constraint

$$\sum_{i=1}^{m} (1 + \lambda f_i(x))_+ \leq m - k$$

guarantees that $f_i(x) \leq 0$ holds for at least k values of i. ($(u)_+$ means $\max\{u, 0\}$.)
Hint. For each $u \in \mathbf{R}$, $(1 + \lambda u)_+ \geq 1(u > 0)$, where $1(u > 0) = 1$ for $u > 0$, and $1(u > 0) = 0$ for $u \leq 0$.

(b) Consider the problem

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & \sum_{i=1}^{m} (1 + \lambda f_i(x))_+ \leq m - k \\ & \lambda > 0, \end{array}$$

with variables x and λ. This is a restriction of the original problem: If (x, λ) are feasible for it, then x is feasible for the original problem. Show how to solve this problem using convex optimization. (This may involve a change of variables.)

(c) Apply the method of part (b) to the problem instance

$$\begin{array}{ll} \text{minimize} & c^T x \\ \text{subject to} & a_i^T x \leq b_i \text{ holds for at least } k \text{ values of } i, \end{array}$$

with $m = 70$, $k = 58$, and $n = 12$. The vectors b, c and the matrix A with rows a_i^T are given in the file `satisfy_some_constraints_data.*`.
Report the optimal value of λ, the objective value, and the actual number of constraints that are satisfied (which should be larger than or equal to k). To determine if a constraint is satisfied, you can use the tolerance $a_i^T x - b_i \leq \epsilon^{\text{feas}}$, with $\epsilon^{\text{feas}} = 10^{-5}$.
A standard trick is to take this tentative solution, choose the k constraints with the smallest values of $f_i(x)$, and then minimize $f_0(x)$ subject to these k constraints (*i.e.*, ignoring the other $m - k$ constraints). This improves the objective value over the one found using the restriction. Carry this out for the problem instance, and report the objective value obtained.

Solution.

(a) We first prove the hint. If $u > 0$ then $1(u > 0) = 1$ and $1 + \lambda u > 1$, so $(1 + \lambda u)_+ > 1$. If $u \leq 0$ then $1(u > 0) = 0$ and $(1 + \lambda u)_+ \geq 0$. Hence $(1 + \lambda u)_+ \geq 1(u > 0)$ for all $u \in \mathbf{R}$.

Applying this to $u = f_i(x)$, we have that $(1 + \lambda f_i(x))_+ \geq 1(f_i(x) > 0)$ for all i. Hence

$$\sum_{i=1}^{m} 1(f_i(x) > 0) \leq \sum_{i=1}^{m} (1 + \lambda f_i(x))_+.$$

The constraint $\sum_{i=1}^{m} (1 + \lambda f_i(x))_+ \leq m - k$ guarantees that $\sum_{i=1}^{m} 1(f_i(x) > 0) \leq m - k$, so $f_i(x) > 0$ holds for at most $m - k$ values of i. In other words, $f_i(x) \leq 0$ holds for at least k values of i.

(b) If $\lambda > 0$ then $(\lambda u)_+ = \lambda (u)_+$ for all $u \in \mathbf{R}$. Hence $(1+\lambda f_i(x))_+ = \lambda \left(1/\lambda + f_i(x)\right)_+$. The constraint $\sum_{i=1}^{m} (1 + \lambda f_i(x))_+ \leq m - k$ can then be written as

$$\sum_{i=1}^{m} \lambda \left(\frac{1}{\lambda} + f_i(x) \right)_+ \leq m - k,$$

or equivalently,

$$\sum_{i=1}^{m} \left(\frac{1}{\lambda} + f_i(x) \right)_+ \leq (m - k) \frac{1}{\lambda}.$$

Letting $\mu = 1/\lambda$, the restricted problem can be expressed as

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & \sum_{i=1}^{m} (\mu + f_i(x))_+ \leq (m - k)\mu \\ & \mu > 0, \end{array}$$

with variables x and μ. Since the function $(\cdot)_+$ is convex and nondecreasing, and $\mu + f_i(x)$ is convex in both μ and x, $(\mu + f_i(x))_+$ is convex, so this is a convex optimization problem.

After solving this problem and obtaining an optimal value $\mu^\star$ for μ, the optimal value of λ is $1/\mu^\star$. Note that we can replace the constraint $\mu > 0$ by $\mu \geq 0$, since if $\mu = 0$ then the constraint $\sum_{i=1}^{m} (\mu + f_i(x))_+ \leq (m - k)\mu$ is the same as all the constraints $f_i(x) \leq 0$ hold. Certainly in this case at least k of them hold. Alternatively we can introduce a new variable t and replace the constraint $\mu > 0$ with $\mu \geq e^t$.

(c) The optimal value of λ is 282.98 and the objective value is -8.45. The number of constraints satisfied is 66, which exceeds our required minimum, $k = 58$.

When we take this tentative solution, choose the k constraints with the smallest values of $f_i(x)$, and then minimize $f_0(x)$ subject to these k constraints, we get an objective value between -8.75 and -8.86 (depending on the solver; these numbers

should be the same . . .). In any case, it gives a modest improvement in objective compared to the restriction.

The actual optimal value (which we obtained using branch and bound, a global optimization method) is -9.57. To compute this takes far more effort than to solve the restriction; for larger problem sizes solving the global problem is prohibitively slow.

The following Matlab code solves the problem:

```
satisfy_some_constraints_data;

cvx_begin quiet
    variables x(n) mu_var
    minimize(c' * x)
    subject to
        sum(pos(mu_var + A * x - b)) <= (m - k) * mu_var
        mu_var >= 0
cvx_end
fprintf('Optimal value of lambda: %f\n', 1 / mu_var)
fprintf('Objective value: %f\n', cvx_optval)

fprintf('Number of constraints satisfied: %d\n', nnz(A * x - b <= 1e-5))

% Choose k least violated inequalities as constraints
[~, idx] = sort(A * x - b);
least_violated = idx(1:k);
cvx_begin quiet
    variables x(n)
    minimize(c' * x)
    subject to
        A(least_violated, :) * x <= b(least_violated)
cvx_end
fprintf('Objective after minimizing wrt k constraints: %f\n', cvx_optval)
```

The following Python code solves the problem:

```
import cvxpy as cvx

from satisfy_some_constraints_data import *

x = cvx.Variable(n)
mu = cvx.Variable()
constraints = [cvx.sum_entries(cvx.pos(mu + A * x - b)) <= (m - k) * mu,
               mu >= 0]
problem = cvx.Problem(cvx.Minimize(c.T * x), constraints)
```

```
problem.solve()
print('Optimal value of lambda: {}'.format(1 / mu.value))
print('Objective value: {}'.format(problem.value))

print('Number of constraints satisfied: {}'
      .format(np.count_nonzero(A.dot(x.value.A1) - b <= 1e-5)))

# Choose k least violated inequalities as constraints
least_violated = np.argsort(A.dot(x.value).A1 - b)[:k]
constraints = [A[least_violated] * x <= b[least_violated]]
problem = cvx.Problem(cvx.Minimize(c.T * x), constraints)
problem.solve()
print('Objective after minimizing wrt k constraints: {}'
      .format(problem.value))
```

The following Julia code solves the problem:

```
using Convex, SCS
set_default_solver(SCSSolver(verbose=false))

include("satisfy_some_constraints_data.jl")

x = Variable(n)
mu = Variable()
constraints = [sum(pos(mu + A * x - b)) <= (m - k) * mu,
               mu >= 0]
problem = minimize(c' * x, constraints)
solve!(problem)
println("Optimal value of lambda: $(1 / mu.value)")
println("Objective value: $(problem.optval)")

num_constraints_satisfied = countnz(A * x.value - b .<= 1e-5)
println("Number of constraints satisfied: $num_constraints_satisfied")

# Choose k least violated inequalities as constraints
least_violated = sortperm((A * x.value - b)[:])[1:k]
constraints = [A[least_violated, :] * x <= b[least_violated]]
problem = minimize(c' * x, constraints)
solve!(problem)
println("Objective after minimizing wrt k constraints: $(problem.optval)")
```

5. *Ideal preference point.* A set of K choices for a decision maker is parametrized by a set of vectors $c^{(1)}, \ldots, c^{(K)} \in \mathbf{R}^n$. We will assume that the entries c_i of each choice are normalized to lie in the range $[0, 1]$. The *ideal preference point model* posits that there is an ideal choice vector c^{ideal} with entries in the range $[0, 1]$; when the decision maker is asked to choose between two candidate choices c and $\tilde{c}$, she will choose the one that is closest (in Euclidean norm) to her ideal point. Now suppose that the decision maker has chosen between all $K(K-1)/2$ pairs of given choices $c^{(1)}, \ldots, c^{(K)}$. The decisions are represented by a list of pairs of integers, where the pair (i, j) means that $c^{(i)}$ is chosen when given the choices $c^{(i)}, c^{(j)}$. You are given these vectors and the associated choices.

   (a) How would you determine if the decision maker's choices are consistent with the ideal preference point model?

   (b) Assuming they are consistent, how would you determine the bounding box of ideal choice vectors consistent with her decisions? (That is, how would you find the minimum and maximum values of c_i^{ideal}, for c^{ideal} consistent with being the ideal preference point.)

   (c) Carry out the method of part (b) using the data given in `ideal_pref_point_data.*`. These files give the points $c^{(1)}, \ldots, c^{(K)}$ and the choices, and include the code for plotting the results. Report the width and the height of the bounding box and include your plot.

Solution.

(a) The decision that c^{ideal} is closer to $c^{(i)}$ than $c^{(j)}$ means that c^{ideal} lies in the half-space

$$\left\{ x \in \mathbf{R}^n \;\middle|\; (c^{(j)} - c^{(i)})^T x \leq \frac{1}{2}(c^{(i)} + c^{(j)})^T (c^{(j)} - c^{(i)}) \right\}.$$

Thus, the decision maker's choices are consistent with an ideal preference point model if and only if the polyhedron obtained by intersecting the hypercube $[0, 1]^n$ and those half-spaces for all decisions (i, j) is nonempty.

Remark: It is insufficient to only check whether the decisions satisfy transitivity (i.e., if (i, j) and (j, k) are decisions, then so is (i, k)). Consider $c^{(i)} = [i] \in \mathbf{R}^1$, $i = 1, 2, 3$, then the decisions $(1, 2), (1, 3), (3, 2)$ satisfy transitivity, though there is no c^{ideal} satisfying the constraints.

(b) The minimum/maximum value of c_k^{ideal} can be obtained by minimizing/maximizing c_k^{ideal} subject to the constraint that c^{ideal} lies in the aforementioned polyhedron. In other words, for each $k = 1, \ldots, n$, to find the minimum value of c_k^{ideal}, we solve the problem

$$\begin{array}{ll} \text{minimize} & c_k^{\text{ideal}} \\ \text{subject to} & 0 \preceq c^{\text{ideal}} \preceq \mathbf{1}, \\ & (c^{(j)} - c^{(i)})^T c^{\text{ideal}} \leq \frac{1}{2}(c^{(i)} + c^{(j)})^T (c^{(j)} - c^{(i)}) \text{ for decisions } (i, j). \end{array}$$

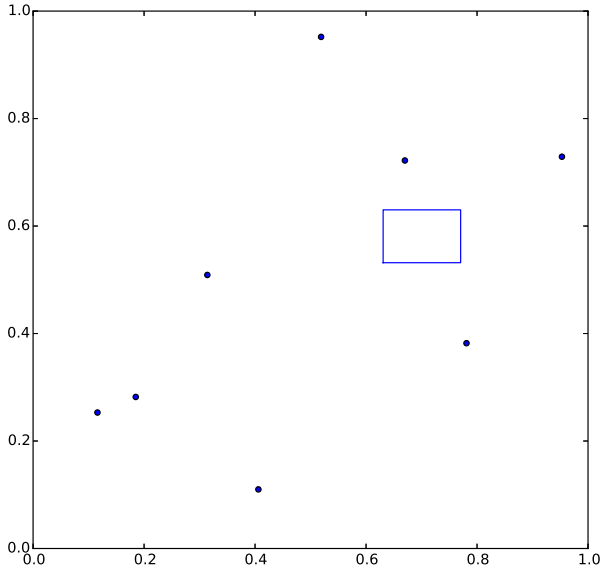

Figure 2: Plot of the points $c^{(i)}$ and the bounding box.

The maximum value is obtained by using maximize instead of minimize.

Remark : It is possible to reduce the number of decisions needed to be considered in the convex problem from $K(K-1)/2$ to $K-1$, as long as the decisions satisfy transitivity (which should be the case if there is no tie). Assume the candidate choices are ordered as $c^{(k_1)}, \ldots, c^{(k_K)}$ from closest to farthest from the ideal point (the ordering can be obtained by counting the number of times $c^{(i)}$ is preferred in the decisions (i, j)), then we need only to consider the decisions $(k_1, k_2), \ldots, (k_{K-1}, k_K)$.

(c) The width and the height of the bounding box are 0.140 and 0.098 respectively. The following Matlab code solves the problem:

```
% Problem data
K = 8;
n = 2;

% List of candidate choices as row vectors
c = [0.314 0.509; 0.185 0.282; 0.670 0.722; 0.116 0.253; ...
     0.781 0.382; 0.519 0.952; 0.953 0.729; 0.406 0.110];

% List of decisions. Row [i j] means c(i) preferred over c(j)
d = [1 2; 3 1; 3 2; 1 4; 2 4; 3 4; 5 1; ...
     5 2; 3 5; 5 4; 1 6; 6 2; 3 6; 6 4; ...
```

```
     5 6; 7 1; 7 2; 3 7; 7 4; 5 7; 7 6; ...
     1 8; 8 2; 3 8; 8 4; 5 8; 6 8; 7 8];

box = zeros(n, 2);

% Put your code for finding the bounding box here.
% box(i, 1) and box(i, 2) should be the lower and upper bounds
% of the i-th coordinate respectively.
for a = 1:n
    for s = [-1, 1]
        cvx_begin
            variables c_ideal(n)
            maximize c_ideal(a) * s
            subject to
                0 <= c_ideal;
                c_ideal <= 1;
                for b = 1:size(d,1)
                    c_ideal' * (c(d(b,2),:) - c(d(b,1),:))' <= (c(d(b,1),:) ...
                        + c(d(b,2),:)) * (c(d(b,2),:) - c(d(b,1),:))' / 2;
                end
        cvx_end
        box(a, (s + 3) / 2) = cvx_optval * s;
    end
end

% Drawing the bounding box
figure;
scatter(c(:,1), c(:,2));
hold on
plot([box(1,1);box(1,2);box(1,2);box(1,1);box(1,1)], ...
     [box(2,1);box(2,1);box(2,2);box(2,2);box(2,1)]);
hold off
xlim([0 1]);
ylim([0 1]);
disp(['Width of bounding box = ' num2str(box(1,2) - box(1,1))])
disp(['Height of bounding box = ' num2str(box(2,2) - box(2,1))])
```

The following Python code solves the problem:

```
from cvxpy import *
import numpy as np
import matplotlib.pyplot as plt
```

```
# Problem data
K = 8
n = 2

# List of candidate choices
c = [[0.314, 0.509], [0.185, 0.282], [0.670, 0.722], [0.116, 0.253],
     [0.781, 0.382], [0.519, 0.952], [0.953, 0.729], [0.406, 0.110]]
c = [np.array(x) for x in c]

# List of decisions. [i, j] means c[i] preferred over c[j]
d = [[0, 1], [2, 0], [2, 1], [0, 3], [1, 3], [2, 3], [4, 0],
     [4, 1], [2, 4], [4, 3], [0, 5], [5, 1], [2, 5], [5, 3],
     [4, 5], [6, 0], [6, 1], [2, 6], [6, 3], [4, 6], [6, 5],
     [0, 7], [7, 1], [2, 7], [7, 3], [4, 7], [5, 7], [6, 7]]

box = [[0] * 2 for a in range(n)]

# Put your code for finding the bounding box here.
# box[i][0] and box[i][1] should be the lower and upper bounds
# of the i-th coordinate respectively.
for a in range(n):
    for s in [-1, 1]:
        c_ideal = Variable(n)
        objective = Maximize(c_ideal[a] * s)
        constraints = [0 <= c_ideal, c_ideal <= 1]
        for b in d:
            constraints.append(c_ideal.T * (c[b[1]] - c[b[0]])
                <= np.dot(c[b[0]] + c[b[1]], c[b[1]] - c[b[0]]) / 2)
        prob = Problem(objective, constraints)
        box[a][(s + 1) // 2] = prob.solve() * s

# Drawing the bounding box
plt.figure()
plt.xlim(0, 1)
plt.ylim(0, 1)
plt.scatter([c[i][0] for i in range(K)], [c[i][1] for i in range(K)])
plt.plot([box[0][0], box[0][1], box[0][1], box[0][0], box[0][0]],
         [box[1][0], box[1][0], box[1][1], box[1][1], box[1][0]])
plt.gca().set_aspect('equal', adjustable='box')
plt.show()
print('Width of bounding box = ' + str(box[0][1] - box[0][0]))
print('Height of bounding box = ' + str(box[1][1] - box[1][0]))
```

The following Julia code solves the problem:

```
using Convex
using PyPlot

# Problem data
K = 8
n = 2

# List of candidate choices as row vectors
c = [0.314 0.509; 0.185 0.282; 0.670 0.722; 0.116 0.253;
     0.781 0.382; 0.519 0.952; 0.953 0.729; 0.406 0.110]

# List of decisions. Row [i j] means c[i] preferred over c[j]
d = [1 2; 3 1; 3 2; 1 4; 2 4; 3 4; 5 1;
     5 2; 3 5; 5 4; 1 6; 6 2; 3 6; 6 4;
     5 6; 7 1; 7 2; 3 7; 7 4; 5 7; 7 6;
     1 8; 8 2; 3 8; 8 4; 5 8; 6 8; 7 8]

box = zeros(n, 2)

# Put your code for finding the bounding box here.
# box[i, 1] and box[i, 2] should be the lower and upper bounds
# of the i-th coordinate respectively.
for a in 1:n
    for s in [-1, 1]
        c_ideal = Variable(n)
        constraints = [0 <= c_ideal; c_ideal <= 1]
        for b = 1:size(d,1)
            push!(constraints, c_ideal' * (c[d[b,2],:] - c[d[b,1],:])'
            <= (c[d[b,1],:] + c[d[b,2],:]) * (c[d[b,2],:] - c[d[b,1],:])' / 2)
        end
        prob = maximize(c_ideal[a] * s, constraints)
        solve!(prob)
        box[a, (s + 3) / 2] = prob.optval * s
    end
end

# Drawing the bounding box
figure()
xlim(0, 1)
ylim(0, 1)
```

```
scatter(c[:,1], c[:,2])
plot([box[1,1];box[1,2];box[1,2];box[1,1];box[1,1]],
     [box[2,1];box[2,1];box[2,2];box[2,2];box[2,1]])
println("Width of bounding box = $(box[1,2] - box[1,1])")
println("Height of bounding box = $(box[2,2] - box[2,1])")
```

6. *Matrix equilibration.* We say that a matrix is ℓ_p *equilibrated* if each of its rows has the same ℓ_p norm, and each of its columns has the same ℓ_p norm. (The row and column ℓ_p norms are related by m, n, and p.) Suppose we are given a matrix $A \in \mathbf{R}^{m\times n}$. We seek diagonal invertible matrices $D \in \mathbf{R}^{m\times m}$ and $E \in \mathbf{R}^{n\times n}$ for which DAE is ℓ_p equilibrated.

   (a) Explain how to find D and E using convex optimization. (Some matrices cannot be equilibrated. But you can assume that all entries of A are nonzero, which is enough to guarantee that it can be equilibrated.)

   (b) Equilibrate the matrix A given in the file `matrix_equilibration_data.*`, with

   $$m = 20, \quad n = 10, \quad p = 2.$$

   Print the row ℓ_p norms and the column ℓ_p norms of the equilibrated matrix as vectors to check that each matches.

   Hints.

   - Work with the matrix B, with $B_{ij} = |A_{ij}|^p$.
   - Consider the problem of minimizing $\sum_{i=1}^m \sum_{j=1}^n B_{ij} e^{u_i+v_j}$ subject to $\mathbf{1}^T u = 0$, $\mathbf{1}^T v = 0$. (Several variations on this idea will work.)
   - We have found that expressing the terms in the objective as $e^{\log B_{ij}+u_i+v_j}$ leads to fewer numerical problems.

   Solution. Following the hint, we find the optimality conditions for the suggested problem. The Lagrangian is

   $$L(u, v, \nu, \omega) = \sum_{i=1}^m \sum_{j=1}^n B_{ij} e^{u_i+v_j} + \nu \mathbf{1}^T u + \omega \mathbf{1}^T v,$$

   with dual variables ν and ω. The optimality conditions are

   $$\frac{\partial L}{\partial u_i} = \sum_{j=1}^n B_{ij} e^{u_i+v_j} + \nu = 0, \quad i = 1, \ldots, m,$$

   and

   $$\frac{\partial L}{\partial v_j} = \sum_{i=1}^m B_{ij} e^{u_i+v_j} + \omega = 0, \quad j = 1, \ldots, n,$$

   along with $\mathbf{1}^T u = \mathbf{1}^T v = 0$. Defining $D = \mathbf{diag}(e^{u/p})$ and $E = \mathbf{diag}(e^{v/p})$, where the exponentials are elementwise, we can write the optimality conditions as

   $$\mathbf{1}^T D^p B E^p = -\nu \mathbf{1}^T, \qquad D^p B E^p \mathbf{1} = -\omega \mathbf{1},$$

i.e., D^pBE^p has all column sums equal to $-\nu$, and all row sums equal to $-\omega$. Therefore, the matrix DAE is ℓ_p equilibrated, since $|(DAE)_{ij}^p| = (D^pBE^p)_{ij}$.

For the given matrix A, we find the equilibrated matrix has row norms and column norms as follows.

```
row_norms =

    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849
    4.2849


col_norms =

    6.0598
    6.0598
    6.0598
    6.0598
    6.0598
    6.0598
    6.0598
    6.0598
    6.0598
    6.0598
    6.0598
```

Code solutions for each language follow.

Matlab:

```
matrix_equilibration_data;
B = abs(A).^p;

cvx_begin
    variables u(m) v(n);
    expression obj;
    for i = 1:m
        for j = 1:n
            obj = obj + exp(log(B(i,j))+u(i)+v(j));
        end
    end
    minimize(obj);
    subject to
        sum(u) == 0;
        sum(v) == 0;
cvx_end

D = diag(exp(u./p));
E = diag(exp(v./p));
A_eq= D*A*E;

row_norms = norms(A_eq,p,2)
col_norms = norms(A_eq,p,1)'
```

Python:

```
from cvxpy import *
import numpy as np

from matrix_equilibration_data import *
B = np.power(np.abs(A), p)

u = Variable(m)
v = Variable(n)

obj = 0
for i in range(m):
    for j in range(n):
        obj += exp(log(B[i, j]) + u[i] + v[j])
obj = Minimize(obj)
```

```
constraints = [sum(u) == 0, sum(v) == 0]

prob = Problem(obj, constraints)
prob.solve(verbose=True)

D = np.diagflat(np.exp(u.value / p))
E = np.diagflat(np.exp(v.value / p))
A_eq = D * A * E

row_norms = np.linalg.norm(A_eq, p, 1)
col_norms = np.linalg.norm(A_eq.T, p, 1)

print(row_norms)
print(col_norms)
```

Julia:

```
# Compute the equilibration
include("matrix_equilibration_data.jl");
B = abs(A).^p;

using Convex;
u = Variable(m); v = Variable(n);

objective = 0;
for i = 1:m
    for j = 1:n
        objective += exp(log(B[i,j])+u[i]+v[j]);
    end
end
constraints = [sum(u) == 0, sum(v) == 0];

problem = minimize(objective,constraints);
solve!(problem);

D = diagm(exp(u.value[:,1]./p));
E = diagm(exp(v.value[:,1]./p));
A_eq = D*A*E;

row_norms = sum(abs(A_eq).^p,2).^(1/p)
col_norms = sum(abs(A_eq').^p,2).^(1/p)
```

7. *Colorization with total variation regularization.* A $m \times n$ color image is represented as three matrices of intensities $R, G, B \in \mathbf{R}^{m \times n}$, with entries in $[0, 1]$, representing the red, green, and blue pixel intensities, respectively. A color image is converted to a monochrome image, represented as one matrix $M \in \mathbf{R}^{m \times n}$, using

$$M = 0.299R + 0.587G + 0.114B.$$

(These weights come from different perceived brightness of the three primary colors.)

In *colorization*, we are given M, the monochrome version of an image, and the color values of *some* of the pixels; we are to guess its color version, *i.e.*, the matrices R, G, B. Of course that's a very underdetermined problem. A very simple technique is to minimize the total variation of (R, G, B), defined as

$$\mathbf{tv}(R, G, B) = \sum_{i=1}^{m-1} \sum_{j=1}^{n-1} \left\| \begin{bmatrix} R_{ij} - R_{i,j+1} \\ G_{ij} - G_{i,j+1} \\ B_{ij} - B_{i,j+1} \\ R_{ij} - R_{i+1,j} \\ G_{ij} - G_{i+1,j} \\ B_{ij} - B_{i+1,j} \end{bmatrix} \right\|_2,$$

subject to consistency with the given monochrome image, the known ranges of the entries of (R, G, B) (*i.e.*, in $[0, 1]$), and the given color entries. Note that the sum above is of the norm of 6-vectors, and not the norm-squared. (The 6-vector is an approximation of the spatial gradient of (R, G, B).)

Carry out this method on the data given in `image_colorization_data.*`. The file loads `flower.png` and provides the monochrome version of the image, `M`, along with vectors of known color intensities, `R_known`, `G_known`, and `B_known`, and `known_ind`, the indices of the pixels with known values. If `R` denotes the red channel of an image, then `R(known_ind)` returns the known red color intensities in Matlab, and `R[known_ind]` returns the same in Python and Julia. The file also creates an image, `flower_given.png`, that is monochrome, with the known pixels colored.

The `tv` function, invoked as `tv(R,G,B)`, gives the total variation. CVXPY has the `tv` function built-in, but CVX and CVX.jl do not, so we have provided the files `tv.m` and `tv.jl` which contain implementations for you to use.

In Python and Julia we have also provided the function `save_img(filename,R,G,B)` which writes the image defined by the matrices `R`, `G`, `B`, to the file `filename`. To view an image in Matlab use the `imshow` function.

The problem instance is a small image, 75×75, so the solve time is reasonable, say, under ten seconds or so in CVX or CVXPY, and around 60 seconds in Julia.

Report your optimal objective value and, if you have access to a color printer, attach your reconstructed image. If you don't have access to a color printer, it's OK to just give the optimal objective value.

Solution.

Let R, G, B denote the matrices for the image, $R^{\text{known}}, G^{\text{known}}, B^{\text{known}}$ the matrices of known color values (only defined for some entries), and $\mathcal{K}$ denote the indices of color values. Image colorization is then the following optimization problem:

$$\begin{array}{ll} \text{minimize} & \mathbf{tv}(R, G, B) \\ \text{subject to} & 0.299R + 0.587G + 0.114B = M \\ & R_{ij} = R_{ij}^{\text{known}}, \quad (i,j) \in \mathcal{K} \\ & G_{ij} = G_{ij}^{\text{known}}, \quad (i,j) \in \mathcal{K} \\ & B_{ij} = B_{ij}^{\text{known}}, \quad (i,j) \in \mathcal{K} \\ & 0 \leq R_{ij}, G_{ij}, B_{ij} \leq 1. \end{array}$$

The following Matlab code solves this problem.

```
image_colorization_data;
cvx_begin quiet
    variables R(m,n) G(m,n) B(m,n)
    minimize tv(R,G,B)
    subject to
        % grayscale reconstruction matches given grayscale
        0.299*R +0.587*G + 0.114*B == M
        % colors match given colors
        R(known_ind) == R_known
        G(known_ind) == G_known
        B(known_ind) == B_known
        % colors in range
        R >= 0; G >= 0; B >= 0
        R <= 1; G <= 1; B <= 1
cvx_end
outim = cat(3,R,G,B);
imshow(outim);
imwrite(outim,'flower_reconstructed.png');
```

Here is the solution in Python.

```
import cvxpy as cvx
from image_colorization_data import *

R = cvx.Variable(m,n)
G = cvx.Variable(m,n)
B = cvx.Variable(m,n)
constraints = [
```

```
  0.299*R + 0.587*G + 0.114*B == M,
  R[known_ind] == R_known,
  G[known_ind] == G_known,
  B[known_ind] == B_known,
  0 <= R, 0 <= G, 0 <= B,
  1 >= R, 1 >= G, 1 >= B,
]
optval = cvx.Problem(cvx.Minimize(cvx.tv(R,G,B)), constraints).solve()
print(optval)
save_img('flower_reconstructed.png', R.value, G.value, B.value)
```

Here is the solution in Julia.

```
using Convex
include("image_colorization_data.jl");
include("tv.jl");

R = Variable(m,n);
G = Variable(m,n);
B = Variable(m,n);

constraints = [
  0.299*R + 0.587*G + 0.114*B == M;
  R[known_ind] == R_known;
  G[known_ind] == G_known;
  B[known_ind] == B_known;
  0 <= R; 0 <= G; 0 <= B;
  1 >= R; 1 >= G; 1 >= B;
];
problem = minimize(tv(R,G,B), constraints)
solve!(problem)
save_img("flower_reconstructed.png", R.value, G.value, B.value)
```

The results are shown below.

Original image on the left, image in monochrome with colored pixels shown in the middle, and reconstructed image on the right.

In Matlab our optimum objective value was 342.29. In Python it was 349.70. In Julia it was 345.94. (These values should be the same, of course.)

8. *Computing market-clearing prices.* We consider n commodities or goods, with $p \in \mathbf{R}^n_{++}$ the vector of prices (per unit quantity) of them. The (nonnegative) demand for the products is a function of the prices, which we denote $D : \mathbf{R}^n \to \mathbf{R}^n$, so $D(p)$ is the demand when the product prices are p. The (nonnegative) supply of the products (*i.e.*, the amounts that manufacturers are willing to produce) is also a function of the prices, which we denote $S : \mathbf{R}^n \to \mathbf{R}^n$, so $S(p)$ is the supply when the product prices are p. We say that the market *clears* if $S(p) = D(p)$, *i.e.*, supply equals demand, and we refer to p in this case as a set of *market-clearing prices*.

   Elementary economics courses consider the special case $n = 1$, *i.e.*, a single commodity, so supply and demand can be plotted (vertically) against the price (on the horizontal axis). It is assumed that demand decreases with increasing price, and supply increases; the market clearing price can be found 'graphically', as the point where the supply and demand curves intersect. In this problem we examine some cases in which market-clearing prices (for the general case $n > 1$) can be computed using convex optimization.

   We assume that the demand function is *Hicksian*, which means it has the form $D(p) = \nabla E(p)$, where $E : \mathbf{R}^n \to \mathbf{R}$ is a differentiable function that is concave and increasing in each argument, called the *expenditure function*. (While not relevant in this problem, Hicksian demand arises from a model in which consumers make purchases by maximizing a concave utility function.)

   We will assume that the producers are independent, so $S(p)_i = S_i(p_i)$, $i = 1, \ldots, n$, where $S_i : \mathbf{R} \to \mathbf{R}$ is the supply function for good i. We will assume that the supply functions are positive and increasing on their domain $\mathbf{R}_+$.

   (a) Explain how to use convex optimization to find market-clearing prices under the assumptions given above. (You do not need to worry about technical details like zero prices, or cases in which there are no market-clearing prices.)

   (b) Compute market-clearing prices for the specific case with $n = 4$,

$$E(p) = \left(\prod_{i=1}^{4} p_i \right)^{1/4},$$

$$S(p) = (0.2p_1 + 0.5, 0.02p_2 + 0.1, 0.04p_3, 0.1p_4 + 0.2).$$

   Give the market-clearing prices and the demand and supply (which should match) at those prices.

   Hint: In CVX and CVXPY, `geo_mean` gives the geometric mean of the entries of a vector argument. Julia does not yet have a vector argument `geom_mean` function, but you can get the geometric mean of 4 variables a, b, c, d using `geomean(geomean(a, b), geomean(c, d))`.

Solution.

(a) Consider the function $L(p) = \sum_{i=1}^{n} \int_0^{p_i} S_i(t)dt$. We claim that $L(p)$ is a convex function of p. Indeed, each separate term $\int_0^{p_i} S_i(t)dt$ is convex in p because it is convex in p_i (its derivative is $S_i(p_i)$ which is increasing by assumption). Thus, $L(p)$ is convex in p being a sum of convex functions. Moreover, note that $S(p) = \nabla L(p)$. We also know that $E(p)$ is concave in p and $D(p) = \nabla E(p)$ by assumption.

We now claim that the market-clearing prices p are exactly the optimal point of the following convex optimization problem:

$$\text{minimize} \quad L(p) - E(p),$$

with variable p. This is a convex problem because L is convex and E is concave. The optimal point $p^\star$ of this problem has to satisfy the condition

$$\nabla(L(p) - E(p))\Big|_{p=p^\star} = 0.$$

Since $S(p) = \nabla L(p)$ and $D(p) = \nabla E(p)$, we conclude that $p^\star$ satisfies $S(p^\star) = D(p^\star)$ which means that $p^\star$ is the market-clearing price vector.

(b) For this problem instance we have

$$L(p) = \sum_{i=1}^{4} \int_0^{p_i} S_i(t)dt = 0.1p_1^2 + 0.5p_1 + 0.01p_2^2 + 0.1p_2 + 0.02p_3^2 + 0.05p_4^2 + 0.2p_4,$$

so all we need to do is solve the problem

$$\text{minimize} \quad L(p) - E(p),$$

with L as above and E as given, *i.e.*, the geometric mean.

Solving this problem we find the following market-clearing prices:

$$p = (0.6363, 2.6193, 3.1589, 1.2341).$$

The supply and demand at these prices are:

$$S(p) = D(p) = (0.6272, 0.1523, 0.1263, 0.3234).$$

The following Matlab code solves the problem.

```
cvx_begin quiet
    variable p(4)
    minimize(0.1 * square(p(1)) + 0.5 * p(1) ...
             + 0.01 * square(p(2)) + 0.1 * p(2) ...
             + 0.02 * square(p(3)) ...
             + 0.05 * square(p(4)) + 0.2 * p(4) ...
             - geo_mean(p))
```

```
cvx_end
fprintf('The market-clearing prices are: %f, %f, %f, %f\n', p)
fprintf('The supply at these prices: %f, %f, %f, %f\n', ...
        [0.2 * p(1) + 0.5, 0.02 * p(2) + 0.1, 0.04 * p(3), 0.1 * p(4) + 0.2])
g = geo_mean(p);
fprintf('The demand at these prices: %f, %f, %f, %f\n', ...
        [g / (4.0 * p(1)), g / (4.0 * p(2)), g / (4.0 * p(3)), g / (4.0 * p(4))
```

The following Python code solves the problem.

```
import cvxpy as cvx
import numpy as np

p = cvx.Variable(4)
obj = 0.1 * cvx.square(p[0]) + 0.5 * p[0] \
      + 0.01 * cvx.square(p[1]) + 0.1 * p[1] \
      + 0.02 * cvx.square(p[2]) \
      + 0.05 * cvx.square(p[3]) + 0.2 * p[3] \
      - cvx.geo_mean(p)
problem = cvx.Problem(cvx.Minimize(obj))
problem.solve()

prices = [p.value.A1[i] for i in range(4)]
print('The market-clearing prices are: {}'.format(prices))
supply = [0.2 * prices[0] + 0.5, 0.02 * prices[1] + 0.1, \
          0.04 * prices[2], 0.1 * prices[3] + 0.2]
print('The supply at these prices: {}'.format(supply))
g = np.power(np.prod(prices), 1.0 / 4)
demand = [g / (4 * prices[i]) for i in range(4)]
print('The demand at these prices: {}'.format(demand))
```

The following Julia code solves the problem.

```
using Convex, SCS
set_default_solver(SCSSolver(verbose=false))

p = Variable(4)
obj = 0.1 * square(p[1]) + 0.5 * p[1] +
      0.01 * square(p[2]) + 0.1 * p[2] +
      0.02 * square(p[3]) +
      0.05 * square(p[4]) + 0.2 * p[4] -
      geomean(geomean(p[1], p[2]), geomean(p[3], p[4]))

problem = minimize(obj)
solve!(problem)
```

```
prices = vec(p.value)
println("Market-clearing prices are: $prices")
supply = [0.2 * prices[1] + 0.5, 0.02 * prices[2] + 0.1,
          0.04 * prices[3], 0.1 * prices[4] + 0.2]
println("Supply at these prices: $supply")
g = (prices[1] * prices[2] * prices[3] * prices[4])^(1.0 / 4)
demand = [g / (4 * prices[1]), g / (4 * prices[2]),
          g / (4 * prices[3]), g / (4 * prices[4])]
println("Demand at these prices: $demand")
```