EE364a: Convex Optimization I S. Boyd
March 13–14 or March 14–15, 2015

Final Exam Solutions

1. *Optimal evacuation planning.* We consider the problem of evacuating people from a dangerous area in a way that minimizes risk exposure. We model the area as a connected graph with n nodes and m edges; people can assemble or collect at the nodes, and travel between nodes (in either direction) over the edges. We let $q_t \in \mathbf{R}^n_+$ denote the vector of the numbers of people at the nodes, in time period t, for $t = 1, \ldots, T$, where T is the number of periods we consider. (We will consider the entries of q_t as real numbers, not integers.) The initial population distribution q_1 is given. The nodes have capacity constraints, given by $q_t \preceq Q$, where $Q \in \mathbf{R}^n_+$ is the vector of node capacities.

   We use the incidence matrix $A \in \mathbf{R}^{n\times m}$ to describe the graph. We assign an arbitrary reference direction to each edge, and take

$$A_{ij} = \begin{cases} +1 & \text{if edge } j \text{ enters node } i \\ -1 & \text{if edge } j \text{ exits node } i \\ 0 & \text{otherwise.} \end{cases}$$

   The population dynamics are given by $q_{t+1} = Af_t + q_t$, $t = 1, \ldots, T-1$ where $f_t \in \mathbf{R}^m$ is the vector of population movement (flow) across the edges, for $t = 1, \ldots, T-1$. A positive flow denotes movement in the direction of the edge; negative flow denotes population flow in the reverse direction. Each edge has a capacity, *i.e.*, $|f_t| \preceq F$, where $F \in \mathbf{R}^m_+$ is the vector of edge capacities, and $|f_t|$ denotes the elementwise absolute value of f_t.

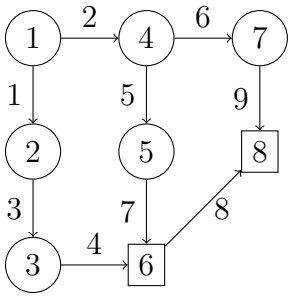
   An *evacuation plan* is a sequence $q_1, q_2, \ldots, q_T$ and $f_1, f_2, \ldots, f_{T-1}$ obeying the constraints above. The goal is to find an evacuation plan that minimizes the total risk exposure, defined as

$$R_{\text{tot}} = \sum_{t=1}^{T} \left(r^T q_t + s^T q_t^2\right) + \sum_{t=1}^{T-1} \left(\tilde{r}^T |f_t| + \tilde{s}^T f_t^2\right),$$

   where $r, s \in \mathbf{R}^n_+$ are given vectors of risk exposure coefficients associated with the nodes, and $\tilde{r}, \tilde{s} \in \mathbf{R}^m_+$ are given vectors of risk exposure coefficients associated with the edges. The notation q_t^2 and f_t^2 refers to elementwise squares of the vectors. Roughly speaking, the risk exposure is a quadratic function of the occupancy of a node, or the (absolute value of the) flow of people along an edge. The linear terms can be interpreted as the risk exposure per person; the quadratic terms can be interpreted as the additional risk associated with crowding.

A subset of nodes have zero risk ($r_i = s_i = 0$), and are designated as *safe nodes*. The population is considered *evacuated* at time t if $r^T q_t + s^T q_t^2 = 0$. The *evacuation time* t_{evac} of an evacuation plan is the smallest such t. We will assume that T is sufficiently large and that the total capacity of the safe nodes exceeds the total initial population, so evacuation is possible.

Use CVX* to find an optimal evacuation plan for the problem instance with data given in `opt_evac_data.*`. (We display the graph below, with safe nodes denoted as squares.)



Report the associated optimal risk exposure $R_{\text{tot}}^\star$. Plot the time period risk

$$R_t = r^T q_t + s^T q_t^2 + \tilde{r}^T |f_t| + \tilde{s}^T f_t^2$$

versus time. (For $t = T$, you can take the edge risk to be zero.) Plot the node occupancies q_t, and edge flows f_t versus time. Briefly comment on the results you see. Give the evacuation time t_{evac} (considering any $r^T q_t + s^T q_t^2 \leq 10^{-4}$ to be zero).

Hint. With CVXPY, use the ECOS solver with `p.solve(solver=cvxpy.ECOS)`.

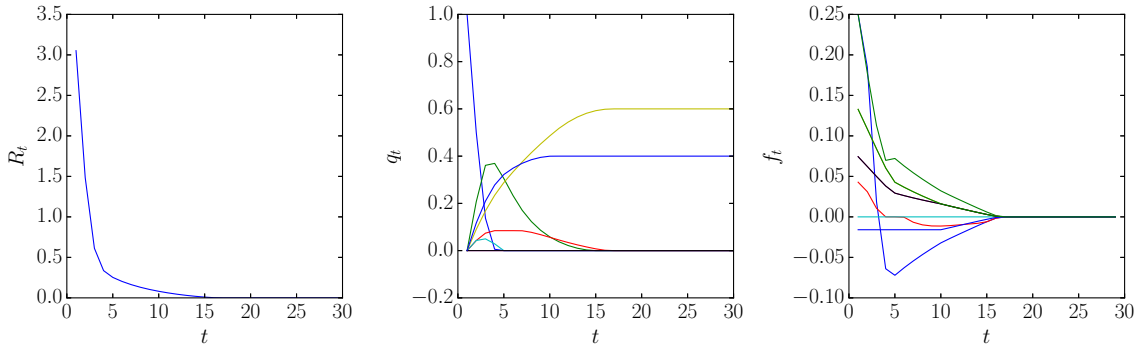
Solution. The optimization problem is given by

$$\begin{array}{ll} \text{minimize} & \sum_{t=1}^{T} \left(r^T q_t + s^T q_t^2\right) + \sum_{t=1}^{T-1} \left(\tilde{r}^T |f_t| + \tilde{s}^T f_t^2\right) \\ \text{subject to} & q_{t+1} = A f_t + q_t, \quad t = 1, \ldots, T-1 \\ & 0 \preceq q_t \preceq Q, \quad t = 2, \ldots, T \\ & |f_t| \preceq F, \quad t = 1, \ldots, T-1, \end{array}$$

with variables $q_2, \ldots, q_T$ and $f_1, \ldots, f_{T-1}$. This is evidently a convex optimization problem, since the objective is convex and the constraints are all linear.

The optimal evacuation time is $t_{\text{evac}} = 17$, with total risk exposure $R_{\text{tot}}^\star = 6.59$.

Note that two of the edge flows reverse direction during the evacuation. This is because the entire population starts at node 1, but not everyone can move to the safe nodes immediately, due to the edge capacity constraints. To avoid accumulating risk, some people move to the safer nodes 2 and 3. Once the bottleneck clears, people flow back in the reverse direction, past node 1, and towards the safe nodes.



The following Python code solves the problem.

```
# solution to optimal evacuation problem
import numpy as np
import cvxpy as cvx
import matplotlib.pyplot as plt
import matplotlib

from opt_evac_data import *

def opt_evac(A, Q, F, q1, r, s, rtild, stild, T):
    n,m = A.shape
    q = cvx.Variable(n,T)
    f = cvx.Variable(m,T-1)

    node_risk = q.T*r + cvx.square(q).T*s
    edge_risk = cvx.vstack(cvx.abs(f).T*rtild + cvx.square(f).T*stild,0)
    risk = node_risk + edge_risk

    constr = [q[:,0] == q1,
              q[:,1:] == A*f + q[:,:-1],
              0 <= q, q <= np.tile(Q,(T,1)).T,
              cvx.abs(f) <= np.tile(F,(T-1,1)).T]

    p = cvx.Problem(cvx.Minimize(sum(risk)), constr)
    p.solve(verbose=True, solver=cvx.ECOS)

    arr = lambda _: np.array(_.value)
    q, f, risk, node_risk = map(arr, (q, f, risk, node_risk))

    print "Total risk: ", p.value
```

```
    print "Evacuated at t =", (node_risk <= 1e-4).nonzero()[0][0] + 1

    return q, f, risk, node_risk

# solve
q, f, risk, node_risk = opt_evac(A, Q, F, q1, r, s, rtild, stild, T)

# plot
plt.rc('text', usetex=True)
plt.rcParams.update({'font.size': 20})
fig, axs = plt.subplots(1,3,figsize=(15,5))

axs[0].plot(np.arange(1,T+1), risk)
axs[0].set_ylabel('$R_t$')

axs[1].plot(np.arange(1,T+1), q.T)
axs[1].set_ylabel('$q_t$')

axs[2].plot(np.arange(1,T), f.T)
axs[2].set_ylabel('$f_t$')

for ax in axs:
    ax.set_xlabel('$t$')

if matplotlib.get_backend().lower() in ['agg', 'macosx']:
    fig.set_tight_layout(True)
else:
    fig.tight_layout()
#plt.tight_layout()
fig.savefig('opt_evac.pdf')
fig.savefig('opt_evac.eps')
```

The following MATLAB code solves the problem.

```
% solution to optimal evacuation problem
opt_evac_data

[n, m] = size(A);

cvx_begin
    variable q(n,T)
    variable f(m,T-1)
    risk = q'*r + square(q)'*s + [abs(f)'*rtild + square(f)'*stild; 0]
```

```
    minimize( sum(risk) )
    subject to
        q(:,2:end) == A*f + q(:,1:end-1)
        q(:,1) == q1
        0 <= q
        q <= repmat(Q,1,T)
        abs(f) <= repmat(F,1,T-1)
cvx_end

fprintf('Total risk: %f\n', sum(risk))
fprintf('Evacuated at t = %d\n', find(q'*r + (q.^2)'*s < 1e-4,1))

subplot(1,3,1)
plot(risk)
ylabel('R_t')
xlabel('t')
subplot(1,3,2)
plot(q')
ylabel('q_t')
xlabel('t')
subplot(1,3,3)
plot(f')
ylabel('f_t')
xlabel('t')
print(gcf,'-deps','opt_evac.eps')
```

The following Julia code solves the problem.

```
# solution to optimal evacuation problem
using Convex, ECOS, PyPlot
include("opt_evac_data.jl");

n,m = size(A)
q = Variable(n,T)
f = Variable(m,T-1)
risk = q'*r + square(q)'*s + [abs(f)'*rtild + square(f)'*stild,0]
p = minimize(sum(risk))
p.constraints += [q[:,1] == q1,
                  q[:,2:end] == A*f + q[:,1:end-1],
                  0 <= q, q <= repmat(Q,1,T),
                  abs(f) <= repmat(F,1,T-1)]
solve!(p, ECOSSolver(verbose=1))
```

```
risk = evaluate(risk)
q = q.value
f = f.value
println("Total risk: $(round(sum(risk),2))")
println("Evacuated at t = $(findfirst(q'*r + (q.*q)'*s .<= 1e-4))")

fig = figure("stuff",figsize=(22,5))
subplot(131)
plot(risk)
ylabel(L"R_t")
xlabel(L"t")
subplot(132)
plot(q')
ylabel(L"q_t")
xlabel(L"t")
subplot(133)
plot(f')
ylabel(L"f_t")
xlabel(L"t")
savefig("opt_evac.eps")
```

2. *Convexity of products of powers.* This problem concerns the product of powers function $f : \mathbf{R}^n_{++} \to \mathbf{R}$ given by $f(x) = x_1^{\theta_1} \cdots x_n^{\theta_n}$, where $\theta \in \mathbf{R}^n$ is a vector of powers. We are interested in finding values of θ for which f is convex or concave. You already know a few, for example when $n = 2$ and $\theta = (2, -1)$, f is convex (the quadratic-over-linear function), and when $\theta = (1/n)\mathbf{1}$, f is concave (geometric mean). Of course, if $n = 1$, f is convex when $\theta \geq 1$ or $\theta \leq 0$, and concave when $0 \leq \theta \leq 1$.

   Show each of the statements below. We will not read long or complicated proofs, or ones that involve Hessians. We are looking for short, snappy ones, that (where possible) use composition rules, perspective, partial minimization, or other operations, together with known convex or concave functions, such as the ones listed in the previous paragraph. Feel free to use the results of earlier statements in later ones.

   (a) When $n = 2$, $\theta \succeq 0$, and $\mathbf{1}^T\theta = 1$, f is concave.

   (b) When $\theta \succeq 0$ and $\mathbf{1}^T\theta = 1$, f is concave. (This is the same as part (a), but here it is for general n.)

   (c) When $\theta \succeq 0$ and $\mathbf{1}^T\theta \leq 1$, f is concave.

   (d) When $\theta \preceq 0$, f is convex.

   (e) When $\mathbf{1}^T\theta = 1$ and exactly *one* of the elements of θ is positive, f is convex.

   (f) When $\mathbf{1}^T\theta \geq 1$ and exactly *one* of the elements of θ is positive, f is convex.

   Remark. Parts (c), (d), and (f) exactly characterize the cases when f is either convex or concave. That is, if none of these conditions on θ hold, f is neither convex nor concave. Your teaching staff has, however, kindly refrained from asking you to show this.

   Solution. To shorten our proofs, when both x and θ are vectors, we overload notation so that

$$f(x) = x_1^{\theta_1} \cdots x_n^{\theta_n} = x^\theta.$$

   (a) Since $x_1^{\theta_1}$ is concave for $0 \leq \theta_1 \leq 1$, applying the perspective transformation gives that

$$x_2(x_1/x_2)^{\theta_1} = x_1^{\theta_1} x_2^{1-\theta_1}$$

   is concave, which is what we wanted.

   (b) The proof is by induction on n. We know the base case with $n = 1$ holds. For the induction step, if $\theta \in \mathbf{R}^{n+1}_+$, $\tilde{\theta} = (\theta_1, \ldots, \theta_n)$, $\tilde{x} = (x_1, \ldots, x_n)$, and $\mathbf{1}^T\theta = 1$, then $\tilde{x}^{\tilde{\theta}/\mathbf{1}^T\tilde{\theta}}$ is concave by the induction assumption. The function $y^{\mathbf{1}^T\tilde{\theta}} z^{1-\mathbf{1}^T\tilde{\theta}}$ is concave by (a) and nondecreasing. The composition rules give that

$$(\tilde{x}^{\tilde{\theta}/\mathbf{1}^T\theta})^{\mathbf{1}^T\tilde{\theta}} x_{n+1}^{1-\mathbf{1}^T\tilde{\theta}} = \tilde{x}^{\tilde{\theta}} x_{n+1}^{\theta_{n+1}} = x^\theta$$

   is concave.

(c) If $\mathbf{1}^T\theta \leq 1$, then $x^{\theta/\mathbf{1}^T\theta}$ is concave by (b). The function $y^{\mathbf{1}^T\theta}$ is concave and nondecreasing. Composition gives that

$$(x^{\theta/\mathbf{1}^T\theta})^{\mathbf{1}^T\theta} = x^\theta$$

is concave.

(d) If $\theta \preceq 0$, then $x^{\theta/\mathbf{1}^T\theta}$ is concave by part (b). (We can assume $\mathbf{1}^T\theta \neq 0$.) The function $y^{\mathbf{1}^T\theta}$ is convex and nonincreasing, since $\mathbf{1}^T\theta < 0$. Composition gives that

$$(x^{\theta/\mathbf{1}^T\theta})^{\mathbf{1}^T\theta} = x^\theta$$

is convex.

Here's another proof, that several people used, and which is arguably simpler than the one above. Since $\theta_i \leq 0$, $\theta_i \log x_i$ is a convex function of x_i, and therefore the sum $\sum_i \theta_i \log x_i$ is convex in x. By the composition rules, the exponential of a convex function is convex, so

$$\exp(\sum_i \theta_i \log x_i) = x^\theta$$

is convex.

(e) If $\theta \in \mathbf{R}^{n+1}$ and $\mathbf{1}^T\theta = 1$, we can assume that the single positive element is $\theta_{n+1} > 0$, so that $\tilde{\theta} = (\theta_1, \ldots, \theta_n) \preceq 0$. If $\tilde{x} = (x_1, \ldots, x_n)$, then $\tilde{x}^{\tilde{\theta}}$ is convex by part (d). Applying the perspective transformation gives that

$$x_{n+1}\left(\tilde{x}/x_{n+1}\right)^{\tilde{\theta}} = \tilde{x}^{\tilde{\theta}} x_{n+1}^{1-\mathbf{1}^T\tilde{\theta}} = \tilde{x}^{\tilde{\theta}} x_{n+1}^{\theta_{n+1}} = x^\theta$$

is convex.

(f) If $\mathbf{1}^T\theta \geq 1$ and exactly one element of θ is positive, then $x^{\theta/\mathbf{1}^T\theta}$ is convex by part (e). The function $y^{\mathbf{1}^T\theta}$ is convex and nondecreasing. Composition gives us that

$$(x^{\theta/\mathbf{1}^T\theta})^{\mathbf{1}^T\theta} = x^\theta$$

is convex.

Remark. The proofs for (c), (d), and (f) are syntactically identical.

Remark. We can also prove (c) with the following self-contained argument. A syntactically identical self-contained argument also works for (f) by substituting "convex" for "concave".

The proof is by induction on n. We know the base case: $x_1^{\theta_1}$ is concave for $0 \leq \theta_1 \leq 1$. For the inductive step, if $\theta \in \mathbf{R}_+^{n+1}$ and $\mathbf{1}^T\theta \leq 1$, let $\tilde{\theta} = (\theta_1, \ldots, \theta_n)$ and $\tilde{x} = (x_1, \ldots, x_n)$. Note that $\tilde{x}^{\tilde{\theta}/\mathbf{1}^T\theta}$ is concave by the induction assumption. Applying the perspective transformation gives that

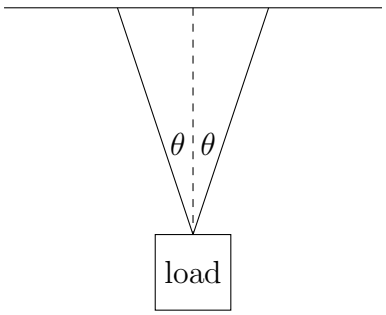
$$x_{n+1}(\tilde{x}/x_{n+1})^{\tilde{\theta}/\mathbf{1}^T\theta} = \tilde{x}^{\tilde{\theta}/\mathbf{1}^T\theta} x_{n+1}^{1-\mathbf{1}^T\tilde{\theta}/\mathbf{1}^T\theta}$$

is concave. The function $y^{\mathbf{1}^T\theta}$ is concave and nondecreasing, and composing it with the previous function shows that

$$(\tilde{x}^{\tilde{\theta}/\mathbf{1}^T\theta} x_{n+1}^{1-\mathbf{1}^T\tilde{\theta}/\mathbf{1}^T\theta})^{\mathbf{1}^T\theta} = \tilde{x}^{\tilde{\theta}} x_{n+1}^{\mathbf{1}^T\theta-\mathbf{1}^T\tilde{\theta}} = \tilde{x}^{\tilde{\theta}} x_{n+1}^{\theta_{n+1}} = x^{\theta}$$

is concave, completing the proof.

3. *Minimum time maneuver for a crane.* A crane manipulates a load with mass $m > 0$ in two dimensions using two cables attached to the load. The cables maintain angles $\pm\theta$ with respect to vertical, as shown below.



The (scalar) tensions T^{left} and T^{right} in the two cables are independently controllable, from 0 up to a given maximum tension $T^{\max}$. The total force on the load is

$$F = T^{\text{left}} \begin{bmatrix} -\sin\theta \\ \cos\theta \end{bmatrix} + T^{\text{right}} \begin{bmatrix} \sin\theta \\ \cos\theta \end{bmatrix} + mg,$$

where $g = (0, -9.8)$ is the acceleration due to gravity. The acceleration of the load is then F/m.

We approximate the motion of the load using

$$p_{i+1} = p_i + hv_i, \quad v_{i+1} = v_i + (h/m)F_i, \quad i = 1, 2, \ldots,$$

where $p_i \in \mathbf{R}^2$ is the position of the load, $v_i \in \mathbf{R}^2$ is the velocity of the load, and $F_i \in \mathbf{R}^2$ is the force on the load, at time $t = ih$. Here $h > 0$ is a small (given) time step.

The goal is to move the load, which is initially at rest at position p^{init} to the position p^{des}, also at rest, in minimum time. In other words, we seek the smallest k for which

$$p_1 = p^{\text{init}}, \quad p_k = p^{\text{des}}, \quad v_1 = v_k = (0, 0)$$

is possible, subject to the constraints described above.

(a) Explain how to solve this problem using convex (or quasiconvex) optimization.

(b) Carry out the method of part (a) for the problem instance with

$$m = 0.1, \quad \theta = 15^\circ, \quad T^{\max} = 2, \quad p^{\text{init}} = (0, 0), \quad p^{\text{des}} = (10, 2),$$

with time step $h = 0.1$. Report the minimum time $k^\star$. Plot the tensions versus time, and the load trajectory, *i.e.*, the points $p_1, \ldots, p_k$ in $\mathbf{R}^2$. Does the load move along the line segment between p^{init} and p^{des} (*i.e.*, the shortest path from p^{init} and p^{des})? Comment briefly.

Solution.

(a) The problem as stated is quasiconvex: To see if $k^\star \leq k$, we simply check if there exists a set of variables that satisfy the constraints, together with $p_k = p^{\text{des}}$, $v_k = 0$.

For a given value for k, we can solve a convex feasibility problem (in fact, an LP) to determine if such a trajectory exists. Let $T \in \mathbf{R}^{2\times k-1}$ be a matrix of the tensions, so that T_{1i}, T_{2i} are T^{left}, T^{right} at time ih, respectively. Then the force applied to the load at time ih be $F_i = MT_i + mg$ where

$$M = \begin{bmatrix} -\sin\theta & \sin\theta \\ \cos\theta & \cos\theta \end{bmatrix}.$$

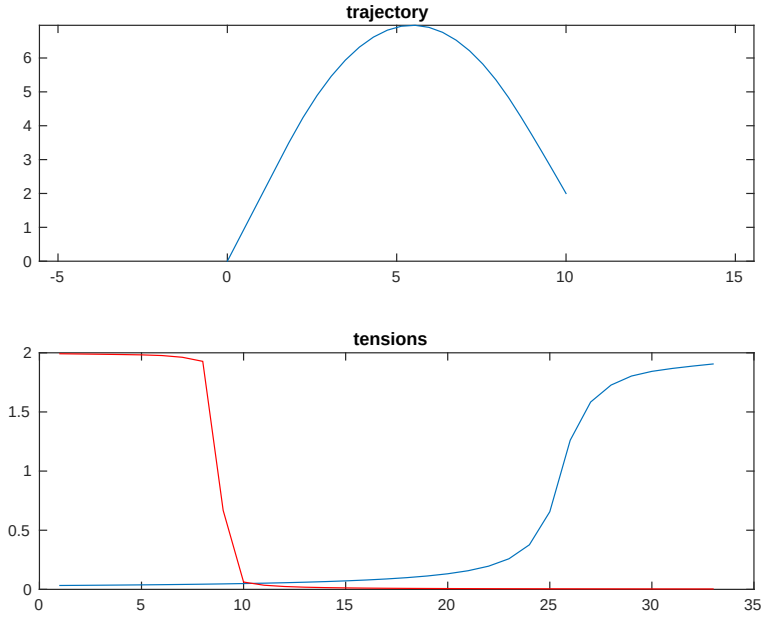
To find a feasible trajectory, we solve the LP

$$\begin{array}{ll} \text{minimize} & 0 \\ \text{subject to} & 0 \preceq T \preceq T^{\max}, \\ & v_{i+1} = v_i + (h/m)F_i, \quad i = 1, \ldots, k-1, \\ & p_{i+1} = p_i + hv_i, \quad i = 1, \ldots, k-1, \\ & p_1 = p^{\text{init}}, \quad p_k = p^{\text{des}}, \quad v_1 = v_k = 0. \end{array}$$

We can then find the minimum time by finding the smallest k for which the above problem is feasible. This can be done by bisection, or by simply increasing k until the problem becomes feasible.

(b) We find that $k^\star = 34$, corresponding to $t = 3.4$ seconds. From the trajectory plot, we see that the load does not travel along the line between the initial and final positions. Since the load must cross a large horizontal distance, we maximize the horizontal force which is accomplished by setting the tension in the right cable to $T^{\max}$. As the tension produces a force along the line of the cable, the load rises up in addition to accelerating in the horizontal direction.

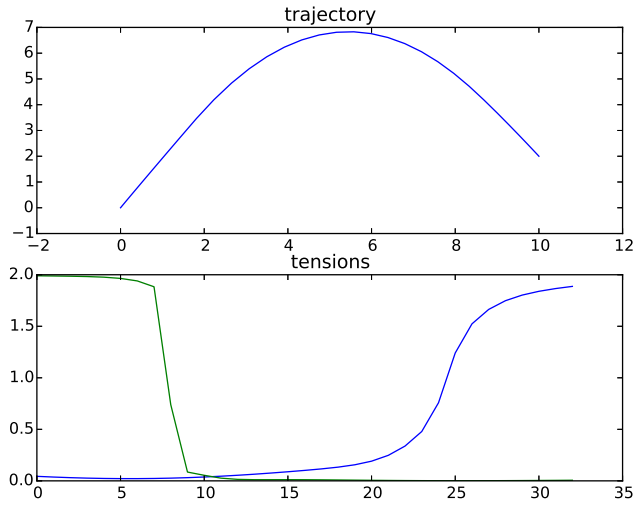
The code is shown below in Matlab.




```
clear;
% Angle of cables with respect to vertical
% For convenience, we put the coefficients in a matrix
theta = 15*pi/180;
M = [-sin(theta), sin(theta); cos(theta), cos(theta)];
T_max = 2; % Max tension that each cable can apply (kNewtons)
m = 0.1; % Mass of the load (metric tons)
g = [0;-9.8]; % Gravity (m/s^2)
p_init = [0;0]; % Init position (m)
p_des = [10;2]; % Desired position (m)
h = 0.1; % Simulation timestep (s)

T_feasible = 0;
p_feasible = 0;

%% Run the problem
lower = 10; % A lowerbound obtained by rough check (infeasible)
upper = 50; % A upperbound obtained by rough check (feasible)
while lower + 1 ~= upper
    k = floor((lower+upper)/2);
    disp(['checking: k=' num2str(k) ...
            ', lower=' num2str(lower) ...
            ', upper=' num2str(upper)]);
    cvx_begin quiet
        variables T(2,k-1) p(2,k) v(2,k)
        F = M*T + m*repmat(g,1,k-1);
```

```
        minimize 0
        subject to
            p(:,1) == p_init; p(:,end) == p_des;
            v(:,1) == 0; v(:,end) == 0;
            0 <= T <= T_max;
            v(:,2:end) == v(:,1:end-1) + h/m*F;
            p(:,2:end) == p(:,1:end-1) + h*v(:,1:end-1);
    cvx_end
    if cvx_optval == 0
        upper = k;
        T_feasible = T;
        p_feasible = p;
    else
        lower = k;
    end
end
k = upper

%% Plotting
figure(1); clf;
subplot(2,1,1); plot(p(1,:),p(2,:)); axis equal;
title('trajectory');
subplot(2,1,2); plot(T_feasible(1,:)); hold on; plot(T_feasible(2,:),'r');
title('tensions');
print -depsc crane_no_constraint
```

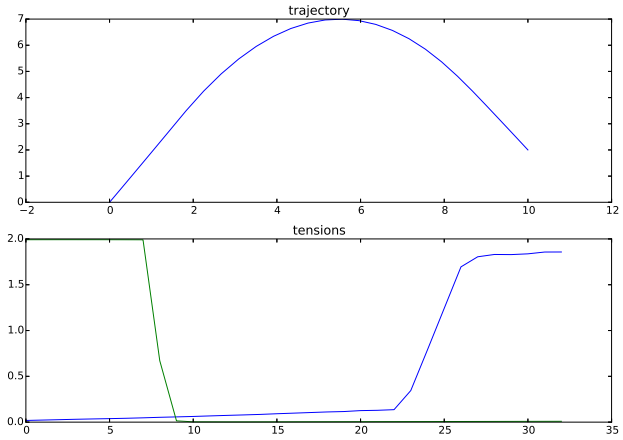
For Python, the code is given below.



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

theta = 15*3.141592/180.0
M = np.matrix([[-np.sin(theta), np.sin(theta)],
               [np.cos(theta), np.cos(theta)]])
T_max = 2.0 # Max tension that each cable can apply (kNewtons)
m = 0.1 # Mass of the load (Metric tons)
g = np.matrix('0;-9.8') # Gravity (m/s^2)
p_init = np.matrix('0.0;0.0') # Init position (m)
p_des = np.matrix('10.0;2.0') # Desired position (m)
h = 0.1 # Simulation timestep (s)

T_feasible = 0
p_feasible = 0

# Run bisection
lower = 10 # Determined by rough check, infeasible
upper = 50 # Determined by rough check, feasible
while not lower + 1 == upper:
    k = int((upper+lower)/2)
    print('checking k=' + str(k) +
            ', lower=' + str(lower) +
            ', upper=' + str(upper))

    T = cvx.Variable(2,k-1)
    v = cvx.Variable(2,k)
    p = cvx.Variable(2,k)

    F = M*T + m*np.tile(g,(1,k-1))

    constraints = [0 <= T, T <= T_max]
    constraints += [p[:,0] == p_init, p[:,k-1] == p_des]
    constraints += [v[:,0] == 0, v[:,k-1] == 0]
    constraints += [v[:,1:k] == v[:,0:k-1] + (h/m)*F]
    constraints += [p[:,1:k] == p[:,0:k-1] + h*v[:,0:k-1]]

    prob = cvx.Problem(cvx.Minimize(0),constraints)

    opt_val = prob.solve(solver=cvx.ECOS,verbose=False)
```

```
    if opt_val == 0:
        upper = k
        T_feasible = T.value
        p_feasible = p.value
    else: lower = k

k = upper;
print('minimum is ' + str(k))
plt.subplot(2,1,1)
plt.plot(p_feasible[0,:].T,p_feasible[1,:].T)
plt.title('trajectory')
plt.subplot(2,1,2)
plt.plot(T_feasible.T)
plt.title('tensions')
plt.savefig('crane_no_constraint.eps',format='ps')
```

The Julia code follows.




```
using Convex, SCS, PyPlot;
using ECOS;

theta = 15*3.141592/180;
M = [-sin(theta) sin(theta); cos(theta) cos(theta)];
T_max = 2.0; # Max tension that each cable can apply (kNewtons)
m = 0.1; # Mass of the load (Metric tons)
g = [0; -9.8]; # Gravity (m/s^2)
p_init = zeros(2,1); # Init position (m)
p_des = [10.0; 2]; # Desired position (m)
h = 0.1; # Simulation timestep (s)
```

```
T_feasible = 0;
p_feasible = 0;

lower = 10; # Determined by rough check (infeasible)
upper = 50; # Determined by rough check (feasible)
while lower + 1 != upper
    k = int((lower+upper)/2);
    println(string("checking k=",k,", lower=",lower, ", upper=", upper));
    T = Variable(2,k-1);
    v = Variable(2,k);
    p = Variable(2,k);

    F = M*T + m*repmat(g,1,k-1);

    constraints = [0 <= T, T <= T_max];
    constraints += [p[:,1] == p_init, p[:,end] == p_des];
    constraints += [v[:,1] == 0, v[:,end] == 0];
    constraints += v[:,2:end] == v[:,1:end-1] + (h/m)*F;
    constraints += p[:,2:end] == p[:,1:end-1] + h*v[:,1:end-1];

    prob = satisfy(constraints);

    #solve!(prob,SCSSolver(verbose=false));
    solve!(prob,ECOSSolver(maxit=20000,eps=1e-4,verbose=false));

    println(prob.status);
    if prob.status == :Optimal
        upper = k;
        T_feasible = T.value;
        p_feasible = p.value;
    elseif prob.status == :Infeasible
        lower = k;
    else
        println("solve failed!!!");
    end
end
k = upper;
println(string("minimum is ",k));

fig = figure("fig1",figsize=(12,8));
subplot(211);
plot(p_feasible[1,:]',p_feasible[2,:]');
```

```
title("trajectory")
subplot(212);
plot(T_feasible');
title("tensions")
savefig("crane_no_constraint.eps",format="ps");
```

4. *Portfolio rebalancing.* We consider the problem of rebalancing a portfolio of assets over multiple periods. We let $h_t \in \mathbf{R}^n$ denote the vector of our dollar value holdings in n assets, at the beginning of period t, for $t = 1, \ldots, T$, with negative entries meaning short positions. We will work with the portfolio weight vector, defined as $w_t = h_t/(\mathbf{1}^T h_t)$, where we assume that $\mathbf{1}^T h_t > 0$, *i.e.*, the total portfolio value is positive.

   The *target portfolio weight vector* $w^\star$ is defined as the solution of the problem

$$\begin{array}{ll} \text{maximize} & \mu^T w - \frac{\gamma}{2} w^T \Sigma w \\ \text{subject to} & \mathbf{1}^T w = 1, \end{array}$$

   where $w \in \mathbf{R}^n$ is the variable, μ is the mean return, $\Sigma \in \mathbf{S}^n_{++}$ is the return covariance, and $\gamma > 0$ is the risk aversion parameter. The data μ, Σ, and γ are given. In words, the target weights maximize the risk-adjusted expected return.

   At the beginning of each period t we are allowed to rebalance the portfolio by buying and selling assets. We call the post-trade portfolio weights $\tilde{w}_t$. They are found by solving the (rebalancing) problem

$$\begin{array}{ll} \text{maximize} & \mu^T w - \frac{\gamma}{2} w^T \Sigma w - \kappa^T |w - w_t| \\ \text{subject to} & \mathbf{1}^T w = 1, \end{array}$$

   with variable $w \in \mathbf{R}^n$, where $\kappa \in \mathbf{R}^n_+$ is the vector of (so-called linear) transaction costs for the assets. (For example, these could model bid/ask spread.) Thus, we choose the post-trade weights to maximize the risk-adjusted expected return, minus the transactions costs associated with rebalancing the portfolio. Note that the pre-trade weight vector w_t is known at the time we solve the problem. If we have $\tilde{w}_t = w_t$, it means that no rebalancing is done at the beginning of period t; we simply hold our current portfolio. (This happens if $w_t = w^\star$, for example.)

   After holding the rebalanced portfolio over the investment period, the dollar value of our portfolio becomes $h_{t+1} = \mathbf{diag}(r_t)\tilde{h}_t$, where $r_t \in \mathbf{R}^n_{++}$ is the (random) vector of asset returns over period t, and $\tilde{h}_t$ is the post-trade portfolio given in dollar values (which you do not need to know). The next weight vector is then given by

$$w_{t+1} = \frac{\mathbf{diag}\,(r_t)\tilde{w}_t}{r_t^T \tilde{w}_t}.$$

   (If $r_t^T \tilde{w}_t \leq 0$, which means our portfolio has negative value after the investment period, we have gone bust, and all trading stops.) The standard model is that r_t are IID random variables with mean and covariance μ and Σ, but this is not relevant in this problem.

   (a) *No-trade condition.* Show that $\tilde{w}_t = w_t$ is optimal in the rebalancing problem if

$$\gamma\,|\Sigma(w_t - w^\star)| \preceq \kappa$$

   holds, where the absolute value on the left is elementwise.

Interpretation. The lefthand side measures the deviation of w_t from the target portfolio $w^\star$; when this deviation is smaller than the cost of trading, you do not rebalance.

Hint. Find dual variables, that with $w = w_t$ satisfy the KKT conditions for the rebalancing problem.

(b) Starting from $w_1 = w^\star$, compute a sequence of portfolio weights $\tilde{w}_t$ for $t = 1, \ldots, T$. For each t, find $\tilde{w}_t$ by solving the rebalancing problem (with w_t a known constant); then generate a vector of returns r_t (using our supplied function) to compute w_{t+1} (The sequence of weights is random, so the results won't be the same each time you run your script. But they should look similar.)

Report the fraction of periods in which the no-trade condition holds and the fraction of periods in which the solution has only zero (or negligible) trades, defined as $\|\tilde{w}_t - w_t\|_\infty \leq 10^{-3}$. Plot the sequence $\tilde{w}_t$ for $t = 1, 2, \ldots, T$.

The file `portf_weight_rebalance_data.*` provides the data, a function to generate a (random) vector r_t of market returns, and the code to plot the sequence $\tilde{w}_t$. (The plotting code also draws a dot for every non-negligible trade.)

Carry this out for two values of κ, $\kappa = \kappa_1$ and $\kappa = \kappa_2$. Briefly comment on what you observe.

Hint. In CVXPY we recommend using the solver ECOS. But if you use SCS you should increase the default accuracy, by passing `eps=1e-4` to the `cvxpy.Problem.solve()` method.

Solution.

(a) *No-trade condition.* The solution $w^\star$ of the problem without transaction costs satisfies the KKT conditions

$$-\mu + \gamma \Sigma w^\star + \nu^\star \mathbf{1} = 0, \quad \mathbf{1}^T w^\star = 1,$$

where $\nu^\star \in \mathbf{R}$ is an optimal dual variable for the constraint $\mathbf{1}^T w = 1$. (This is a set of linear equations that we can easily solve, but we don't need this fact for this problem.)

Now we derive optimality conditions for the rebalancing problem. First we express it in the form

$$\begin{array}{ll} \text{maximize} & \mu^T w - \frac{\gamma}{2} w^T \Sigma w - \kappa^T s \\ \text{subject to} & \mathbf{1}^T w = 1, \\ & w - w_t \preceq s \\ & w_t - w \preceq s, \end{array}$$

with additional (slack) variable $s \in \mathbf{R}^n$. Defining dual variables ν, λ_+, and λ_- for

the three constraints, the optimality conditions are

$$\begin{aligned}
-\mu + \gamma\Sigma w + \nu\mathbf{1} + \lambda_+ - \lambda_- &= 0 \\
\kappa - \lambda_+ - \lambda_- &= 0 \\
\lambda_+ &\succeq 0 \\
\lambda_- &\succeq 0 \\
\lambda_+^T(w - w_t - s) &= 0 \\
\lambda_-^T(w_t - w - s) &= 0 \\
\mathbf{1}^T w &= 1 \\
w - w_t &\preceq s \\
w_t - w &\preceq s.
\end{aligned}$$

We want to find a condition under which $w = w_t$, $s = 0$ is optimal. (This means we do not trade.) With these choices for primal variables, the last 5 conditions hold. So we need to find dual variables so that the first 4 conditions hold. We'll use the dual variable $\nu^\star$ from the original problem, and seek λ_+ and λ_- so that the following 4 conditions hold:

$$\begin{aligned}
-\mu + \gamma\Sigma w_t + \nu^\star\mathbf{1} + \lambda_+ - \lambda_- &= 0 \\
\kappa - \lambda_+ - \lambda_- &= 0 \\
\lambda_+ &\succeq 0 \\
\lambda_- &\succeq 0.
\end{aligned}$$

The first and second conditions can be written (subtracting the optimal solution of the problem without transaction costs)

$$\gamma\Sigma(w_t - w^\star) + \lambda_+ - \lambda_- = 0, \quad \kappa = \lambda_+ + \lambda_-.$$

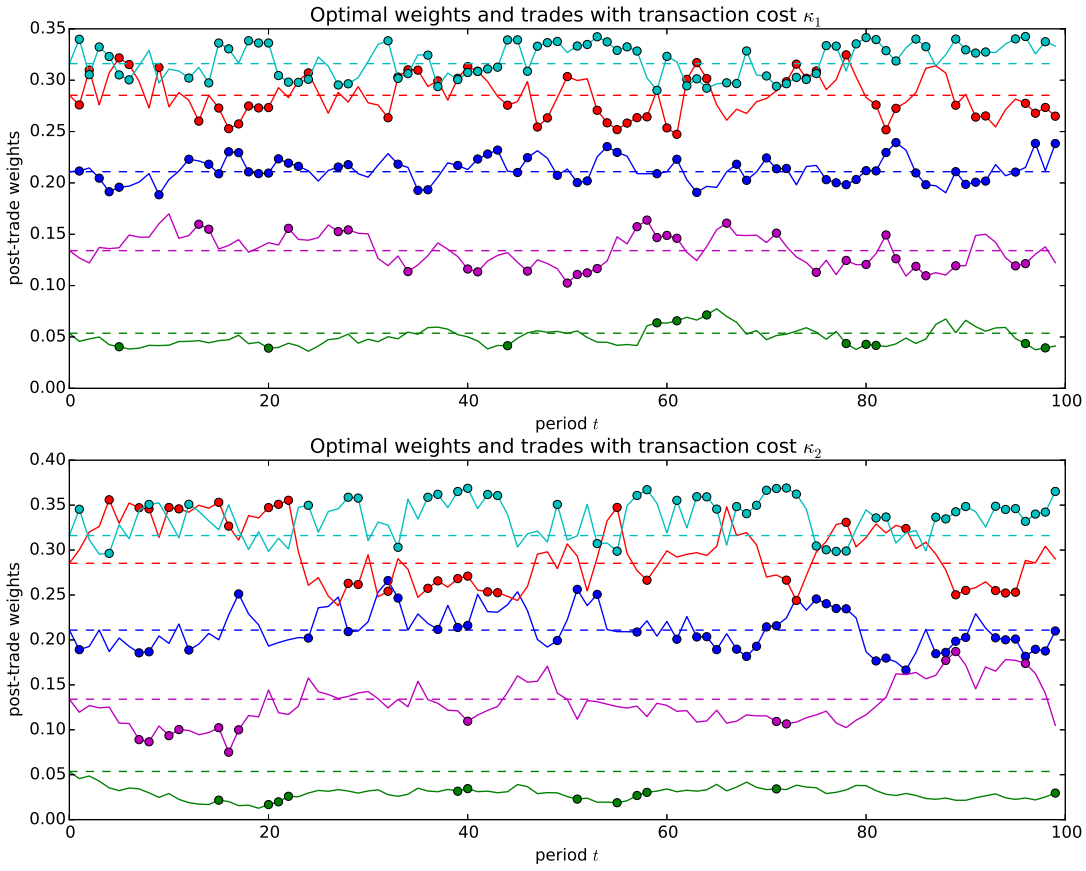
This holds for some $\lambda_+ \succeq 0$ and $\lambda_- \succeq 0$ if and only if

$$|\gamma\Sigma(w_t - w^\star)| \preceq \kappa.$$

This is what we were supposed to show. Under this condition, $w = w_t$ is optimal for the rebalancing problem, *i.e.*, we do not trade.

(b) We report the average plus or minus one standard deviations of the results, computed over multiple runs of the solution script. (The results vary since the returns are generated randomly.) With $\kappa = \kappa_1$, the no-trade condition holds $(10 \pm 3)\%$ of the times and the trades are negligible $(17 \pm 4)\%$ of the times. With $\kappa = \kappa_2$ the no-trade condition holds $(27 \pm 6)\%$ of the times and the trades are negligible $(38 \pm 7)\%$ of the times. We observe that the no-trade condition holds for less periods than the periods with negligible trades. This is correct since the no-trade condition is a sufficient but not necessary condition for having negligible trades.

We also observe that with transaction costs κ_2 we rebalance less than with κ_1, since $\kappa_2 \succeq \kappa_1$. The plots show that the sequence of $\tilde{w}_t$ deviate randomly from $w^\star$ and the trades tend to bring it closer to $w^\star$ (that is in fact rebalancing). By increasing the transaction costs, $\kappa_2 \succeq \kappa_1$, we rebalance less and let the weights diverge farther from the target $w^\star$.



The following Python code solves the problem:

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

T = 100
n = 5
gamma = 8.0
Sigma = np.array([[  1.512e-02,  1.249e-03,  2.762e-04, -5.333e-03, -7.938e-04]
```

```
 [  1.249e-03,  1.030e-02,  6.740e-05, -1.301e-03, -1.937e-04],
 [  2.762e-04,  6.740e-05,  1.001e-02, -2.877e-04, -4.283e-05],
 [ -5.333e-03, -1.301e-03, -2.877e-04,  1.556e-02,  8.271e-04],
 [ -7.938e-04, -1.937e-04, -4.283e-05,  8.271e-04,  1.012e-02]])
mu = np.array([ 1.02 , 1.028, 1.01 , 1.034, 1.017])
kappa_1 = np.array([ 0.002, 0.002, 0.002, 0.002, 0.002])
kappa_2 = np.array([ 0.004, 0.004, 0.004, 0.004, 0.004])
threshold = 0.001

## Solve target weights problem
w = cvx.Variable(n)
cvx.Problem(cvx.Maximize(w.T*mu - (gamma/2.)*cvx.quad_form(w, Sigma)),
                        [cvx.sum_entries(w) == 1]).solve()
w_star = w.value.A1

generateReturns = lambda: np.random.multivariate_normal(mu,Sigma)

## Generate market scenario
plt.figure(figsize=(13,10))
for i, kappa in [(1,kappa_1), (2,kappa_2)]:
    ws = np.zeros((T,n))
    us = np.zeros((T,n))
    no_trade_cond = np.zeros(T)
    w_t = w_star
    for t in range(T):
        # check if no-trade condition holds
        no_trade_cond[t]= max(gamma*np.abs(np.dot(Sigma, w_t - w_star)) -
            kappa) <= 0
        w = cvx.Variable(n)
        cvx.Problem(cvx.Maximize(w.T*mu -
            (gamma/2.)*cvx.quad_form(w, Sigma) -
            cvx.sum_entries(cvx.abs(w - w_t).T*kappa)),
            [cvx.sum_entries(w) == 1]).solve()
        ws[t,:] = w.value.A1
        us[t,:] = w.value.A1 - w_t
        w_t = w.value.A1 * generateReturns()
        w_t /= sum(w_t)

    neglig_trades = np.max(np.abs(us),1) < threshold
    print "The no-trade condition holds %.1f%% of the times."%(
        sum(no_trade_cond)*100./T)
    print "The optimal solution has neglig. trades %.1f%% of the times."%(
```

```
        sum(neglig_trades)*100./T)

    plt.subplot(210+i)
    colors = ['b','r','g','c','m']
    for j in range(n):
        plt.plot(range(T), ws[:,j], colors[j])
        plt.plot(range(T), [w_star[j]]*T,  colors[j]+'--')
        non_zero_trades = abs(us[:,j]) > threshold
        plt.plot(np.arange(T)[non_zero_trades],
            ws[non_zero_trades, j], colors[j]+'o')
    plt.ylabel('post-trade weights')
    plt.xlabel('period $t$')
    plt.title('Optimal weights and trades with transaction cost $\kappa_%d$'%i)
plt.savefig("portfolio_weights.eps")
```

The following Matlab code solves the problem:

```
T = 100;
n = 5;
gamma = 8.0;
threshold = 0.001;
Sigma = [[  1.512e-02,  1.249e-03,  2.762e-04, -5.333e-03, -7.938e-04],
 [  1.249e-03,  1.030e-02,  6.740e-05, -1.301e-03, -1.937e-04],
 [  2.762e-04,  6.740e-05,  1.001e-02, -2.877e-04, -4.283e-05],
 [ -5.333e-03, -1.301e-03, -2.877e-04,  1.556e-02,  8.271e-04],
 [ -7.938e-04, -1.937e-04, -4.283e-05,  8.271e-04,  1.012e-02]];
mu = [ 1.02 , 1.028, 1.01 , 1.034, 1.017];
kappa_1 = [ 0.002, 0.002, 0.002, 0.002, 0.002];
kappa_2 = [ 0.004, 0.004, 0.004, 0.004, 0.004];

% Solve target weights problem
cvx_begin quiet
    variable w_star(n)
    maximize(mu*w_star - (gamma/2.)*w_star'*Sigma*w_star)
    subject to
        sum(w_star) == 1
cvx_end

generateReturns = @() mu + randn(1, length(mu)) * chol(Sigma);

kappas = {kappa_1, kappa_2};
figure('position', [0, 0, 900, 800]);
for i = 1:2
    kappa = kappas{i};
```

```
ws = zeros(T,n);
us = zeros(T,n);
no_trade_cond = zeros(T,1);
w_t = w_star;
for t = 1:T
    % check if no-trade condition holds
    no_trade_cond(t) = max(gamma*abs(Sigma*(w_t - w_star))-kappa') <= 0;

    cvx_begin quiet
        variable w_tilde(n)
        maximize(mu*w_tilde - (gamma/2.)*w_tilde'*Sigma*w_tilde - ...
            kappa*(abs(w_tilde - w_t)))
        subject to
            sum(w_tilde) == 1
    cvx_end
    ws(t,:) = w_tilde';
    us(t,:) = (w_tilde - w_t)';
    w_t = w_tilde.*generateReturns()';
    w_t = w_t/sum(w_t);
end

neglig_trades = max(abs(us')) < threshold;
fprintf('The no-trade condition holds %f%% of the times.\n', ...
    sum(no_trade_cond)*100./T);
fprintf(['The optimal solution has neglig. ' ...
    'trades %f%% of the times.\n'], ...
    sum(neglig_trades)*100./T);
colors = ['b','r','g','c','m'];
range = 1:T;
subplot(2,1,i)
for j = 1:n
   hold on
   plot(range, ws(:,j), ['-' colors(j)]);
   plot(range, ones(T)*w_star(j), ['--' colors(j)]);
   non_zero_trades = abs(us(:,j)) > threshold;
   plot(range(non_zero_trades), ws(non_zero_trades,j), ...
       ['o' colors(j)], 'MarkerFaceColor', colors(j));
   xlabel('Period t');
   ylabel('Post-trade weights w_t tilde');
   title(['Optimal weights and trades with transaction cost kappa_' ...
    num2str(i)]);
end
```

```
end
print -depsc portfolio_weights
```

The following Julia code solves the problem:

```
# data and starter code for multiperiod portfolio rebalancing problem
T = 100;
n = 5;
gamma = 8.0;
threshold = 0.001;
Sigma = [[  1.512e-02   1.249e-03   2.762e-04  -5.333e-03  -7.938e-04]
 [  1.249e-03   1.030e-02   6.740e-05  -1.301e-03  -1.937e-04]
 [  2.762e-04   6.740e-05   1.001e-02  -2.877e-04  -4.283e-05]
 [ -5.333e-03  -1.301e-03  -2.877e-04   1.556e-02   8.271e-04]
 [ -7.938e-04  -1.937e-04  -4.283e-05   8.271e-04   1.012e-02]];
mu = [ 1.02 , 1.028, 1.01 , 1.034, 1.017];
kappa_1 = [ 0.002, 0.002, 0.002, 0.002, 0.002];
kappa_2 = [ 0.004, 0.004, 0.004, 0.004, 0.004];

using Distributions, Convex, SCS, PyPlot
solver = SCSSolver(verbose=0)

# compute target weights
w_star = Variable(n);
problem = maximize(mu'*w_star - (gamma/2.)*quad_form(w_star,Sigma),
    sum(w_star) == 1);
solve!(problem, solver);
w_star = w_star.value;

generateReturns() = rand(MvNormal(mu, Sigma));

kappas = {kappa_1, kappa_2};
figure(figsize=(13,10));
for i = 1:2
    kappa = kappas[i];
    us = zeros(T,n);
    ws = zeros(T,n);
    no_trade_cond = zeros(T);
    w_t = w_star;
    for t = 1:T
        # check if no-trade condition holds
        no_trade_cond[t] = maximum(gamma*abs(Sigma*(w_t - w_star))
            - kappa) <= 0;
        w_tilde = Variable(n);
```

```
        problem = maximize(mu'*w_tilde - (gamma/2.)*quad_form(w_tilde,Sigma) -
                           kappa'*(abs(w_tilde - w_t)), sum(w_tilde) == 1);
        solve!(problem, solver);
        w_tilde = w_tilde.value;
        ws[t,:] = w_tilde;
        us[t,:] = (w_tilde - w_t)';
        w_t = w_tilde.*generateReturns();
        w_t = w_t/sum(w_t);
    end

    neglig_trades = maximum(abs(us),2) .< threshold;
    @printf("The no-trade condition holds %.1f%% of the times.\n",
        sum(no_trade_cond)*100/T);
    @printf("The optimal solution has neglig. trades %.1f%% of the times.\n",
        sum(neglig_trades)*100./T);

    colors = ["b","r","g","c","m"];
    subplot(210+i);
    for j = 1:n
        plot(1:T, ws[:,j], colors[j]);
        plot(1:T, w_star[j]*ones(T),  colors[j]*"--");
        non_zero_trades = abs(us[:,j]) .> threshold;
        plot((1:T)[non_zero_trades], ws[non_zero_trades,j], colors[j]*"o");
    end
    ylabel("post-trade weights");
    xlabel("period \$t\$");
    title(@sprintf "Opt. weights and trades with trans. cost \$\\kappa_%d\$" i)
end
savefig(@sprintf "portfolio_weights.eps")
```

5. *Solving nonlinear circuit equations using convex optimization.* An electrical circuit consists of b two-terminal devices (or branches) connected to n nodes, plus a so-called ground node. The goal is to compute several sets of physical quantities that characterize the circuit operation. The vector of *branch voltages* is $v \in \mathbf{R}^b$, where v_j is the voltage appearing across device j. The vector of *branch currents* is $i \in \mathbf{R}^b$, where i_j is the current flowing through device j. (The symbol i, which is often used to denote an index, is unfortunately the standard symbol used to denote current.) The vector of *node potentials* is $e \in \mathbf{R}^n$, where e_k is the potential of node k with respect to the ground node. (The ground node has potential zero by definition.)

   The circuit variables v, i, and e satisfy several physical laws. Kirchhoff's current law (KCL) can be expressed as $Ai = 0$, and Kirchhoff's voltage law (KVL) can be expressed as $v = A^T e$, where $A \in \mathbf{R}^{n\times b}$ is the reduced incidence matrix, which describes the circuit topology:

$$A_{kj} = \begin{cases} -1 & \text{branch } j \text{ enters node } k \\ +1 & \text{branch } j \text{ leaves node } k \\ 0 & \text{otherwise,} \end{cases}$$

   for $k = 1, \ldots, n$, $j = 1, \ldots, b$. (KCL states that current is conserved at each node, and KVL states that the voltage across each branch is the difference of the potentials of the nodes it is connected to.)

   The branch voltages and currents are related by

$$v_j = \phi_j(i_j), \quad j = 1, \ldots, b,$$

   where ϕ_j is a given function that depends on the *type* of device j. We will assume that these functions are continuous and nondecreasing. We give a few examples. If device j is a resistor with resistance $R_j > 0$, we have $\phi_j(i_j) = R_j i_j$ (which is called Ohm's law). If device j is a voltage source with voltage V_j and internal resistance $r_j > 0$, we have $\phi_j(i_j) = V_j + r_j i_j$. And for a more interesting example, if device j is a diode, we have $\phi_j(i_j) = V_T \log(1 + i_j/I_S)$, where I_S and V_T are known positive constants.

   (a) Find a method to solve the circuit equations, *i.e.*, find v, i, and e that satisfy KCL, KVL, and the branch equations, that relies on convex optimization. State the optimization problem clearly, indicating what the variables are. Be sure to explain how solving the convex optimization problem you propose leads to choices of the circuit variables that satisfy all of the circuit equations. You can assume that no pathologies occur in the problem that you propose, for example, it is feasible, a suitable constraint qualification holds, and so on.

   Hint. You might find the function $\psi : \mathbf{R}^b \to \mathbf{R}$,

$$\psi(i_1, \ldots, i_b) = \sum_{j=1}^{b} \int_0^{i_j} \phi_j(u_j)\, du_j,$$

   useful.

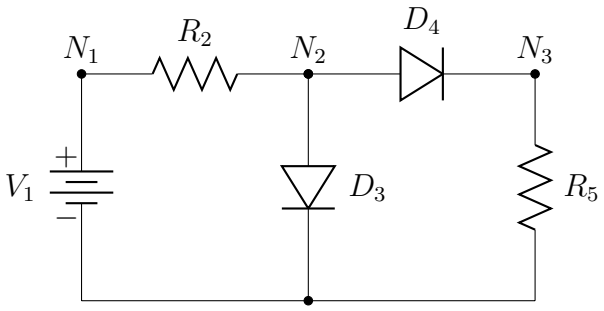
(b) Consider the circuit shown in the diagram below. Device 1 is a voltage source with parameters $V_1 = 1000$, $r_1 = 1$. Devices 2 and 5 are resistors with resistance $R_2 = 1000$, and $R_5 = 100$ respectively. Devices 3 and 4 are identical diodes with parameters $V_T = 26$, $I_S = 1$. (The units are mV, mA, and Ω.)

The nodes are labeled N_1, N_2, and N_3; the ground node is at the bottom. The incidence matrix A is

$$A = \begin{bmatrix} 1 & 1 & 0 & 0 & 0 \\ 0 & -1 & 1 & 1 & 0 \\ 0 & 0 & 0 & -1 & 1 \end{bmatrix}.$$

(The reference direction for each edge is down or to the right.)

Use the method in part (a) to compute v, i, and e. Verify that all the circuit equations hold.



Solution.

(a) We first observe that the function ψ given in the hint is convex: since ϕ_j is nondecreasing,

$$\int_0^{i_j} \phi_j(u_j)\, du_j$$

is convex in i_j, and ψ is the sum of these functions. We note that

$$\nabla\psi(i) = (\phi_1(i_1), \ldots, \phi_b(i_b)).$$

The optimization problem to solve is

$$\begin{array}{ll} \text{minimize} & \psi(i) \\ \text{subject to} & Ai = 0, \end{array}$$

with variable $i \in \mathbf{R}^b$. The optimality conditions are $Ai = 0$ (which is KCL), and $\nabla\psi(i) + A^T\nu = 0$, where ν is a dual variable associated with the constraint $Ai = 0$. Defining $v = \nabla\psi(i)$ and $e = -\nu$, the optimality conditions can be expressed as

$$Ai = 0, \qquad A^T e = v, \qquad v_j = \phi_j(i_j), \quad j = 1, \ldots, b.$$

These are *exactly* the circuit equations.

By the way, this characterization of a circuit in terms of an optimization problem was known to J. C. Maxwell. The function ψ is called the *content function* for the circuit.

(b) Let us first compute the content function for each component.

For a resistor

$$\int_0^i ru \; du = (1/2)ri^2.$$

For a voltage source

$$\int_0^i V + ru \; du = Vi + (1/2)ri^2.$$

For a diode

$$V_T \int_0^i \log(1 + u/I_S) \; du = V_T I_S((1 + i/I_S)\log(1 + i/I_S) - i/I_S).$$

We add the individual content functions for the circuit, yielding in the optimization problem

$$\begin{array}{ll} \text{minimize} & V_1 i_1 + (1/2)\sum_{j\in\{1,2,5\}} R_j i_j^2 + \sum_{j\in\{3,4\}} V_T I_S((1 + i_j/I_S)\log(1 + i_j/I_S) - i_j/I_S) \\ \text{subject to} & Ai = 0. \end{array}$$

Solving this problem in CVX* results in

$$A^T e = \begin{bmatrix} 999.017 \\ 982.966 \\ 16.051 \\ 3.154 \\ 12.897 \end{bmatrix}, \quad i = \begin{bmatrix} -0.983 \\ 0.983 \\ 0.854 \\ 0.129 \\ 0.129 \end{bmatrix}, \quad e = \begin{bmatrix} 999.017 \\ 16.051 \\ 12.897 \end{bmatrix}, \quad \phi(i) = \begin{bmatrix} 999.017 \\ 982.967 \\ 16.051 \\ 3.154 \\ 12.897 \end{bmatrix}$$

and

$$\frac{\|\phi(i) - A^T e\|_2}{\|\phi(i)\|_2} = 3.996 \cdot 10^{-7}.$$

We conclude that the solutions match.

In CVX (and POGS)

```
% Setup
A = [ 1  1  0  0  0
      0 -1  1  1  0
      0  0  0 -1  1
     -1  0 -1  0 -1];

% Remove redundant ground constraint (ie force ground potential = 0)
A = A(1:end-1, :);
```

```
R1 = 1;
R2 = 1e3;
R5 = 1e2;

VT = 26;
IS = 1;
VS = 1e3;

cvx_begin
  variable ii(5)
  dual variable e
  OBJ1 = VS*ii(1) + (1/2)*R1*ii(1)^2;
  OBJ2 = (1/2)*R2*ii(2)^2;
  OBJ3 = VT*IS*(-entr(1 + ii(3)/IS) - ii(3)/IS);
  OBJ4 = VT*IS*(-entr(1 + ii(4)/IS) - ii(4)/IS);
  OBJ5 = (1/2)*R5*ii(5)^2;
  minimize(OBJ1 + OBJ2 + OBJ3 + OBJ4 + OBJ5)
  subject to
    e : A * ii == 0;
cvx_end

% Check Constraints
v = [VS + R1*ii(1)
     R2*ii(2)
     VT*log(1 + ii(3)/IS)
     VT*log(1 + ii(4)/IS)
     R5*ii(5)];

v_err = v - A' * e;

fprintf('Relative error in voltage: %e\n', norm(v_err) / norm(v))

%% POGS

f.h = kIndEq0;
g.h = [kSquare; kSquare; kNegEntr; kNegEntr; kSquare];
g.a = [ 1;  1;  1/IS;  1/IS;  1];
g.b = [ 0;  0;    -1;    -1;  0];
g.c = [R1; R2; VT*IS; VT*IS; R5];
g.d = [VS;  0;   -VT;   -VT;  0];
```

```
[ii, ~, l] = pogs(A, f, g);
e = -l;

% Check Constraints
v = [VS + R1*ii(1)
     R2*ii(2)
     VT*log(1 + ii(3)/IS)
     VT*log(1 + ii(4)/IS)
     R5*ii(5)];

v_err = v - A' * e;

fprintf('Relative error in voltage: %e\n', norm(v_err) / norm(v))
```

In Julia

```
# Pkg.update()
# Pkg.add("Convex")
# Pkg.add("SCS")

using Convex

# Setup
A = [ 1  1  0  0  0
      0 -1  1  1  0
      0  0  0 -1  1
     -1  0 -1  0 -1];

# Remove redundant ground constraint
A = A[1:end-1, :];

R1 = 1
R2 = 1e3
R5 = 1e2

VT = 26
IS = 1
VS = 1e3

ii = Variable(5)

OBJ1 = VS*ii[1] + (1./2)*R1*ii[1]^2
OBJ2 = (1./2)*R2*ii[2]^2
```

```
OBJ3 = VT*IS*(-entropy(1. + ii[3]/IS) - ii[3]/IS)
OBJ4 = VT*IS*(-entropy(1. + ii[4]/IS) - ii[4]/IS)
OBJ5 = (1./2)*R5*ii[5]^2

problem = minimize(OBJ1 + OBJ2 + OBJ3 + OBJ4 + OBJ5, [A * ii == 0])
solve!(problem)
ee = -problem.constraints[1].dual

v = [VS + R1*ii.value[1]
     R2*ii.value[2]
     VT*log(1 + ii.value[3]/IS)
     VT*log(1 + ii.value[4]/IS)
     R5*ii.value[5]]

v_err = v - A' * ee

@printf("Relative error in voltage: %e\n", norm(v_err) / norm(v))

println(v)
println(ii)
```

In CVXPY

```
from cvxpy import *
import numpy as np
import math

A = np.array([[ 1,  1,  0,  0,  0],
              [ 0, -1,  1,  1,  0],
              [ 0,  0,  0, -1,  1],
              [-1,  0, -1,  0, -1]], dtype=np.float64)
A = A[0:-1,:]

R1 = 1.
R2 = 1e3
R5 = 1e2

VT = 26
IS = 1
VS = 1e3

ii = Variable(5)

OBJ1 = VS*ii[0] + (1./2)*R1*square(ii[0])
```

```
OBJ2 = (1./2)*R2*square(ii[1])
OBJ3 = VT*IS*(-entr(1. + ii[2]/IS) - ii[2]/IS)
OBJ4 = VT*IS*(-entr(1. + ii[3]/IS) - ii[3]/IS)
OBJ5 = (1./2)*R5*square(ii[4])

obj = Minimize(OBJ1 + OBJ2 + OBJ3 + OBJ4 + OBJ5)
constr = [A * ii == 0.]
problem = Problem(obj, constr)

#problem.solve(verbose=True, solver=SCS, eps=1e-4)
problem.solve(verbose=True)

e = -problem.constraints[0].dual_value

v = np.array([[VS + R1*float(ii.value[0])],
              [R2*float(ii.value[1])],
              [VT*math.log(1. + float(ii.value[2])/IS)],
              [VT*math.log(1. + float(ii.value[3])/IS)],
              [R5*float(ii.value[4])]])

v_err = v - np.transpose(A) * e
rel_err = np.linalg.norm(v_err) / np.linalg.norm(v)

print "Relative error in voltage: %e\n" % rel_err

print v
print ii.value
```

6. *Optimal material blending.* A standard industrial operation is to blend or mix raw materials (typically fluids such as different grades of crude oil) to create blended materials or products. This problem addresses optimizing the blending operation. We produce n blended materials from m raw materials. Each raw and blended material is characterized by a vector that gives the concentration of each of q constituents (such as different octane hydrocarbons). Let $c_1, \ldots, c_m \in \mathbf{R}^q_+$ and $\tilde{c}_1, \ldots, \tilde{c}_n \in \mathbf{R}^q_+$ be the concentration vectors of the raw materials and the blended materials, respectively. We have $\mathbf{1}^T c_j = \mathbf{1}^T \tilde{c}_i = 1$ for $i = 1, \ldots, n$ and $j = 1, \ldots, m$. The raw material concentrations are given; the blended product concentrations must lie between some given bounds, $\tilde{c}_i^{\min} \preceq \tilde{c}_i \preceq \tilde{c}_i^{\max}$.

   Each blended material is created by pumping raw materials (continuously) into a vat or container where they are mixed to produce the blended material (which continuously flows out of the mixing vat). Let $f_{ij} \geq 0$ denote the flow of raw material j (say, in kg/s) into the vat for product i, for $i = 1, \ldots, n$, $j = 1, \ldots, m$. These flows are limited by the total availability of each raw material: $\sum_{i=1}^n f_{ij} \leq F_j$, $j = 1, \ldots, m$, where $F_j > 0$ is the maximum total flow of raw material j available. Let $\tilde{f}_i \geq 0$ denote the flow rates of the blended materials. These also have limits: $\tilde{f}_i \leq \tilde{F}_i$, $i = 1, \ldots, n$.

   The raw and blended material flows are related by the (mass conservation) equations

$$\sum_{j=1}^m f_{ij} c_j = \tilde{f}_i \tilde{c}_i, \quad i = 1, \ldots, n.$$

   (The lefthand side is the vector of incoming constituent mass flows and the righthand side is the vector of outgoing constituent mass flows.)

   Each raw and blended material has a (positive) price, p_j, $j = 1, \ldots, m$ (for the raw materials), and $\tilde{p}_i$, $i = 1, \ldots, n$ (for the blended materials). We pay for the raw materials, and get paid for the blended materials. The total profit for the blending process is

$$-\sum_{i=1}^n \sum_{j=1}^m f_{ij} p_j + \sum_{i=1}^n \tilde{f}_i \tilde{p}_i.$$

   The goal is to choose the variables f_{ij}, $\tilde{f}_i$, and $\tilde{c}_i$ so as to maximize the profit, subject to the constraints. The problem data are c_j, $\tilde{c}_i^{\min}$, $\tilde{c}_i^{\max}$, F_j, $\tilde{F}_i$, p_j, and $\tilde{p}_j$.

   (a) Explain how to solve this problem using convex or quasi-convex optimization. You must justify any change of variables or problem transformation, and explain how you recover the solution of the blending problem from the solution of your proposed problem.

   (b) Carry out the method of part (a) on the problem instance given in `material_blending_data.*`. Report the optimal profit, and the associated values of f_{ij}, $\tilde{f}_i$, and $\tilde{c}_i$.

Solution.

(a) The problem we are to solve is

$$\begin{array}{ll} \text{maximize} & -\sum_{i,j} f_{ij}p_j + \sum_i \tilde{f}_i\tilde{p}_i \\ \text{subject to} & \sum_j f_{ij}c_j = \tilde{f}_i\tilde{c}_i \\ & \mathbf{1}^T\tilde{c}_i = 1 \\ & c_i^{\min} \preceq \tilde{c}_i \preceq c_i^{\max} \\ & 0 \le \tilde{f}_i \le \tilde{F}_i \\ & 0 \le f_{ij} \\ & \sum_i f_{ij} \le F_j \end{array}$$

with variables f_{ij}, $\tilde{f}_i$, and $\tilde{c}_i$. Each constraint that is indexed by i must hold for $i = 1, \ldots, n$, and each constraint is indexed by j must hold for $j = 1, \ldots, m$.

The objective and all constraints except the first set of equality constraints are linear. On the right hand side of the first set of inequalities we have the product of two variables $\tilde{f}_i$ and $\tilde{c}_i$, so these constraints are not convex.

To deal with this, we introduce new variables $m_i = \tilde{f}_i\tilde{c}_i \in \mathbf{R}^q$ for $i = 1, \ldots, n$, and reformulate the problem as an optimization problem with decision variables f_{ij}, $\tilde{f}_i$, and m_i, removing the variables $\tilde{c}_i$. The vectors m_i are the blended product constituent mass flows.

The variables $\tilde{c}_i$ only appear in the first three sets of constraints. In the first set of equality constraints, we can simply replace $\tilde{f}_i\tilde{c}_i$ with m_i, which results in a set of linear equality constraints. We express $\mathbf{1}^T\tilde{c}_i = 1$ as $\mathbf{1}^T m_i = \tilde{f}_i$; these are equivalent since $\mathbf{1}^T m_i = \tilde{f}_i\mathbf{1}^T\tilde{c}_i = \tilde{f}_i$. The third set of constraints is equivalent to $\tilde{f}_i c_i^{\min} \preceq m_i \preceq \tilde{f}_i c_i^{\max}$. Therefore, the problem becomes

$$\begin{array}{ll} \text{maximize} & -\sum_{i,j} f_{ij}p_j + \sum_i \tilde{f}_i\tilde{p}_i \\ \text{subject to} & \sum_j f_{ij}c_j = m_i, \quad i = 1, \ldots, n \\ & \mathbf{1}^T m_i = \tilde{f}_i, \quad i = 1, \ldots, n \\ & \tilde{f}_i c_i^{\min} \preceq m_i \preceq \tilde{f}_i c_i^{\max}, \quad i = 1, \ldots, n \\ & 0 \le f_{ij}, \quad i = 1, \ldots, n \quad j = 1, \ldots, m \\ & 0 \le \tilde{f}_i \le \tilde{F}_i, \quad i = 1, \ldots, n \\ & \sum_i f_{ij} \le F_j, \quad j = 1, \ldots, m, \end{array}$$

with variables f_{ij}, $\tilde{f}_i$, and m_i. This is an LP. In order to reconstruct the solution to the original problem, we find $\tilde{c}_i$ by $\tilde{c}_i = m_i/\tilde{f}_i$.

(b) The following MATLAB code solves the problem.

```
clear all
material_blending_data

cvx_begin
```

```
variables f(2,4) ftilde(2) m(3,2)
maximize -sum(f*p)+ sum(m*pTilde)
subject to
    m == C*f'
    sum(m) == ftilde'
    0 <= f
    0 <= ftilde <= FTilde
    c_minTilde*diag(ftilde) <= m <= c_maxTilde*diag(ftilde)
    sum(f)' <= F
cvx_end
```

The following Python code solves the problem.

```
from cvxpy import *
from material_blending_data import *

f = Variable(2,4)
ftilde = Variable(2)
m = Variable(3,2)

objective = Maximize(-sum_entries(f * p)+ sum_entries(m * pTilde))
constraints = [m == C*f.T,
               np.ones([1,3])*m == ftilde.T,
               0 <= f,
               0 <= ftilde,
               ftilde <= FTilde,
               c_minTilde * diag(ftilde) <= m,
               m <= c_maxTilde * diag(ftilde),
               (np.ones([1,2]) * f).T <= F]

prob = Problem(objective, constraints)

result = prob.solve()
print prob.value
```

The following Julia code solves the problem.

```
using Convex, ECOS
include("material_blending_data.jl");

f = Variable(2,4)
ftilde = Variable(2)
m = Variable(3,2)

obj = (-sum(f * p)+ sum(m * pTilde))
```

```
prob = maximize(obj)
prob.constraints += [m == C*f',
                ones(1,3)*m == ftilde',
                0 <= f,
                0 <= ftilde,
                ftilde <= FTilde,
                c_minTilde * diagm(ftilde) <= m,
                m <= c_maxTilde * diagm(ftilde),
                f'*ones(2,1) <= F]


solve!(prob, ECOSSolver(verbose=0,max_iters=20000))

println(prob.optval)
```

We find that the optimal value is 127, with a solution

$$f = \begin{bmatrix} 6.0555 & 0.9167 & 0.7065 & 0.3213 \\ 0.9445 & 1.0833 & 5.2935 & 2.6787 \end{bmatrix},$$

$$\tilde{f} = \begin{bmatrix} 8 \\ 10 \end{bmatrix}, \quad \tilde{c}_1 = \begin{bmatrix} 0.8588 \\ 0.1000 \\ 0.0412 \end{bmatrix}, \quad \tilde{c}_2 = \begin{bmatrix} 0.7029 \\ 0.1800 \\ 0.1171 \end{bmatrix}.$$

7. *Graph isomorphism via linear programming.* An (undirected) graph with n vertices can be described by its adjacency matrix $A \in \mathbf{S}^n$, given by

$$A_{ij} = \begin{cases} 1 & \text{there is an edge between vertices } i \text{ and } j \\ 0 & \text{otherwise.} \end{cases}$$

Two (undirected) graphs are *isomorphic* if we can permute the vertices of one so it is the same as the other (*i.e.*, the same pairs of vertices are connected by edges). If we describe them by their adjacency matrices A and B, isomorphism is equivalent to the existence of a permutation matrix $P \in \mathbf{R}^{n\times n}$ such that $PAP^T = B$. (Recall that a matrix P is a permutation matrix if each row and column has exactly one entry 1, and all other entries 0.) Determining if two graphs are isomorphic, and if so, finding a suitable permutation matrix P, is called the *graph isomorphism problem.*

Remarks (not needed to solve the problem). It is not currently known if the graph isomorphism problem is NP-complete or solvable in polynomial time. The graph isomorphism problem comes up in several applications, such as determining if two descriptions of a molecule are the same, or whether the physical layout of an electronic circuit correctly reflects the given circuit schematic diagram.

(a) Find a set of linear equalities and inequalities on $P \in \mathbf{R}^{n\times n}$, that together with the Boolean constraint $P_{ij} \in \{0, 1\}$, are necessary and sufficient for P to be a permutation matrix satisfying $PAP^T = B$. Thus, the graph isomorphism problem is equivalent to a Boolean feasibility LP.

(b) Consider the relaxed version of the Boolean feasibility LP found in part (a), *i.e.*, the LP that results when the constraints $P_{ij} \in \{0, 1\}$ are replaced with $P_{ij} \in [0, 1]$. When this LP is infeasible, we can be sure that the two graphs are not isomorphic. If a solution of the LP is found that satisfies $P_{ij} \in \{0, 1\}$, then the graphs are isomporphic and we have solved the graph isomorphism problem. This of course does not always happen, even if the graphs are isomorphic.

A standard trick to encourage the entries of P to take on the values 0 and 1 is to add a random linear objective to the relaxed feasibility LP. (This doesn't change whether the problem is feasible or not.) In other words, we minimize $\sum_{i,j} W_{ij}P_{ij}$, where W_{ij} are chosen randomly (say, from $\mathcal{N}(0, 1)$). (This can be repeated with different choices of W.)

Carry out this scheme for the two isomorphic graphs with adjacency matrices A and B given in `graph_isomorphism_data.*` to find a permutation matrix P that satisfies $PAP^T = B$. Report the permutation vector, given by the matrix-vector product Pv, where $v = (1, 2, \ldots, n)$. Verify that all the required conditions on P hold. To check that the entries of the solution of the LP are (close to) $\{0, 1\}$, report $\max_{i,j} P_{ij}(1 - P_{ij})$. And yes, you might have to try more than one instance of the randomized method described above before you find a permutation that establishes isomorphism of the two graphs.

Solution.

(a) P is a permutation matrix if and only if $P\mathbf{1} = \mathbf{1}$, $P^T\mathbf{1} = \mathbf{1}$, and $P_{ij} \in \{0, 1\}$. The condition $PAP^T = B$ is a quadratic equality on P. But we observe that since P is a permutation matrix, we have $P^{-1} = P^T$. Multiplying $PAP^T = B$ on the right by P we get $PA = BP$, a set of linear equations in P. So, P is a permutation matrix that satisfies $PAP^T = B$ if and only if

$$P\mathbf{1} = \mathbf{1}, \quad P^T\mathbf{1} = \mathbf{1}, \quad PA = BP, \quad P_{ij} \in \{0, 1\}.$$

This is a set of linear equations in P, together with the Boolean condition $P_{ij} \in \{0, 1\}$.

(b) The LP relaxation, with the random cost function suggested, is

$$\begin{array}{ll} \text{minimize} & \mathbf{Tr}(W^T P) \\ \text{subject to} & P\mathbf{1} = \mathbf{1}, \quad P^T\mathbf{1} = \mathbf{1}, \quad PA = BP \\ & 0 \leq P_{ij} \leq 1. \end{array}$$

(The constraints $P_{ij} \leq 1$ are redundant and can be removed.)

When we solve this problem for the given data, we find that there are two different permutation vectors π_1 and π_2 that relate A and B. The quantity $\max_{i,j} P_{ij}(1 - P_{ij})$ is suitably small, on the order of 10^{-6}. These two vectors are

$$\begin{aligned} \pi_1 = (&16, 22, 27, 9, 5, 13, 1, 25, 21, 23, 19, 14, 26, 6, 2, 7, \\ &30, 11, 10, 17, 15, 24, 8, 18, 20, 3, 12, 28, 4, 29), \end{aligned}$$

and

$$\begin{aligned} \pi_2 = (&6, 22, 27, 19, 15, 3, 11, 25, 21, 23, 9, 4, 26, 16, 12, 17, \\ &30, 1, 20, 7, 5, 24, 18, 8, 10, 13, 2, 28, 14, 29). \end{aligned}$$

It is interesting to notice that solving the feasibility problem with itself (*i.e.*, with objective function 0) usually doesn't end up finding a permutation matrix. However adding the linear random objective does help us find a permutation matrix.

The following MATLAB code solves the problem

```
graph_isomorphism_data
n = size(A,1);
W = randn(n);

cvx_begin quiet
    variable P(n,n)
    minimize trace(W*P)
```

```
    subject to
        P'*ones(n,1) == ones(n,1);
        P*ones(n,1) == ones(n,1);
        P*A - B*P == 0;
        0 <= P <= 1
cvx_end
fprintf(['maximum p(1-p) where p is an entry of P is ' ...
         '%d\n'], max(max(P.*(1-P))))
fprintf(['norm of the residual for first constraint is ' ...
         '%d\n'], norm(P'*ones(n,1) - ones(n,1)))
fprintf(['norm of the residual for second constraint is ' ...
         '%d\n'], norm(P*ones(n,1) - ones(n,1)))
fprintf(['norm of the residual for third constraint is ' ...
         '%d\n'], norm(P*A - B*P))
P*(1:n)'
```

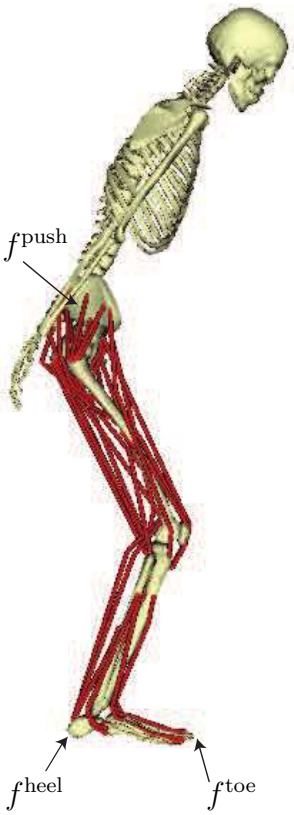
The following Python code solves the problem

```
from cvxpy import *
from graph_isomorphism_data import *

n = A.shape[0]
W = np.random.randn(n, n)

P = Variable(n,n)
objective = Minimize(trace(W*P))
constraints = [ P*np.ones([n,1]) == np.ones([n,1]),
                P.T*np.ones([n,1]) == np.ones([n,1]),
                P*A == B*P,
                0 <= P,
                P <= 1]

prob = Problem(objective, constraints)
result = prob.solve()
P = P.value
print('maximum p(1-p) where p is an entry of P is '
        + str(np.max(np.multiply(P,1-P))))
print('norm of the residual for first constraint is '
        + str(np.linalg.norm(P*np.ones([n,1])- np.ones([n,1]))))
print('norm of the residual for second constraint is '
        + str(np.linalg.norm(P.T*np.ones([n,1]) - np.ones([n,1]))))
print('norm of the residual for third constraint is '
        + str(np.linalg.norm(P*A-B*P)))
print(np.dot(P,np.arange(n)+1))
```

The following Julia code solves the problem

```
using Convex, SCS
include("graph_isomorphism_data.jl");

n = size(A,1);
W = randn(n,n);

P = Variable(n, n);
obj = trace(W*P);
constraints = [
  sum(P, 2) == ones(n),
  sum(P, 1) == ones(1, n),
  P*A == B*P,
  P >= 0,
  P <= 1
];
prob = minimize(obj, constraints);
solve!(prob, SCSSolver(verbose=false, max_iters=20000));
P = P.value;
println(maximum(P.*(1-P)));
println(norm(sum(P, 2) - ones(n)));
println(norm(sum(P, 1) - ones(1, n)));
println(vecnorm(P*A-B*P));
println(P);
```

8. *Maintaining static balance.* In this problem we study a human's ability to maintain balance against an applied external force. We will use a planar (two-dimensional) model to characterize the set of push forces a human can sustain before he or she is unable to maintain balance. We model the human as a linkage of 4 body segments, which we consider to be rigid bodies: the foot, lower leg, upper leg, and pelvis (into which we lump the upper body). The pose is given by the joint angles, but this won't matter in this problem, since we consider a fixed pose. A set of 40 muscles act on the body segments; each of these develops a (scalar) tension t_i that satisfies $0 \leq t_i \leq T_i^{\max}$, where $T_i^{\max}$ is the maximum possible tension for muscle i. (The maximum muscle tensions depend on the pose, and the person, but here they are known constants.) An external pushing force $f^{\text{push}} \in \mathbf{R}^2$ acts on the pelvis. Two (ground contact) forces act on the foot: $f^{\text{heel}} \in \mathbf{R}^2$ and $f^{\text{toe}} \in \mathbf{R}^2$. (These are shown at right.) These must satisfy



$$|f_1^{\text{heel}}| \leq \mu f_2^{\text{heel}}, \quad |f_1^{\text{toe}}| \leq \mu f_2^{\text{toe}},$$

where $\mu > 0$ is the coefficient of friction of the ground. There are also joint forces that act at the joints between the body segments, and gravity forces for each body segment, but we won't need them explicitly in this problem.

To maintain balance, the net force and torque on each each body segment must be satisfied. These equations can be written out from the geometry of the body (*e.g.*, attachment points for the muscles) and the pose. They can be reduced to a set of 6 linear equations:

$$A^{\text{musc}} t + A^{\text{toe}} f^{\text{toe}} + A^{\text{heel}} f^{\text{heel}} + A^{\text{push}} f^{\text{push}} = b,$$

where $t \in \mathbf{R}^{40}$ is the vector of muscle tensions, and A^{musc}, A^{toe}, A^{heel}, and A^{push} are known matrices and $b \in \mathbf{R}^6$ is a known vector. These data depend on the pose, body weight and dimensions, and muscle lines of action. Fortunately for you, our biomechanics expert Apoorva has worked them out; you will find them in `static_balance_data.*` (along with $T^{\max}$ and μ).

We say that the push force f^{push} can be *resisted* if there exist muscle tensions and ground contact forces that satisfy the constraints above. (This raises a philosophical question: Does a person solve an optimization to decide whether he or she should lose their balance? In any case, this approach makes good predictions.)

Find $\mathcal{F}^{\text{res}} \subset \mathbf{R}^2$, the set of push forces that can be resisted. Plot it as a shaded region.

Hints. Show that $\mathcal{F}^{\text{res}}$ is a convex set. For the given data, $0 \in \mathcal{F}^{\text{res}}$. Then for $\theta = 1^\circ, 2^\circ, \ldots, 360^\circ$, determine the maximum push force, applied in the direction θ, that can be resisted. To make a filled region on a plot, you can use the command `fill()`

in Matlab. For Python and Julia, `fill()` is also available through PyPlot. In Julia, make sure to use the ECOS solver with `solver = ECOSSolver(verbose=false)`.

Remark. A person can resist a much larger force applied to the hip than you might think.

Solution. The set of vectors $(t, f^{\text{toe}}, f^{\text{heel}}, f^{\text{push}})$ which satisfy the constraints

$$\begin{array}{l} A^{\text{musc}}t + A^{\text{toe}}f^{\text{toe}} + A^{\text{heel}}f^{\text{heel}} + A^{\text{push}}f^{\text{push}} = b \\ |f_1^{\text{toe}}| \leq \mu f_2^{\text{toe}} \\ |f_1^{\text{heel}}| \leq \mu f_2^{\text{heel}} \\ 0 \preceq t \preceq T^{\max}, \end{array}$$

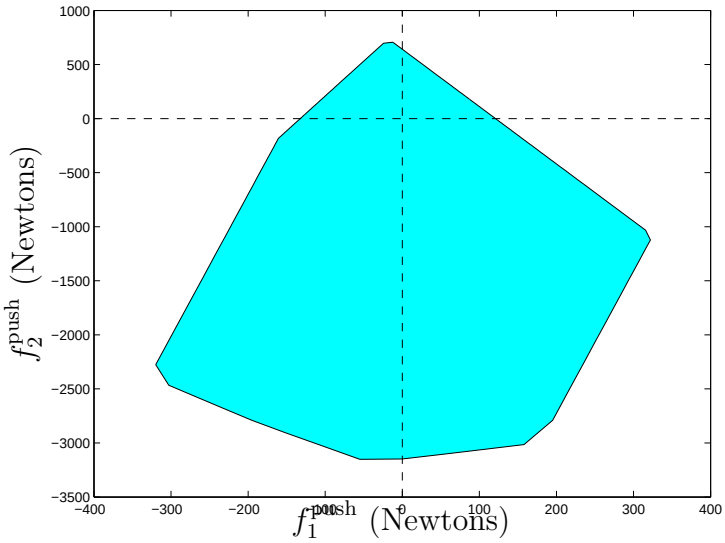
forms a convex set, a polyhedron. It follows the set of push forces that can be resisted, $\mathcal{F}^{\text{res}}$, is a convex set, since it is the projection of this set onto the variable f^{push}. In fact, $\mathcal{F}^{\text{res}}$ is also a polyhedron.

To find the maximum push force for a given direction θ, we solve the LP

$$\begin{array}{ll} \text{maximize} & z \\ \text{subject to} & A^{\text{musc}}t + A^{\text{toe}}f^{\text{toe}} + A^{\text{heel}}f^{\text{heel}} + A^{\text{push}}z(\cos\theta, \sin\theta) = b \\ & |f_1^{\text{toe}}| \leq \mu f_2^{\text{toe}} \\ & |f_1^{\text{heel}}| \leq \mu f_2^{\text{heel}} \\ & 0 \preceq t \preceq T^{\max}. \end{array}$$

with variables $t, f^{\text{toe}}, f^{\text{heel}}$ and $z \in \mathbf{R}$. We solve this problem for a number of values of θ, and for each one we record the value $f^{\text{push}} = z^\star(\cos\theta, \sin\theta)$, which is on the boundary of $\mathcal{F}^{\text{res}}$. We use these values to fill out the set when plotting.

The set $\mathcal{F}^{\text{res}}$ for the given data is shown below.



We see some interesting things from the plot. One is that it's much easier to make someone lose their static balance by pulling them up, instead of pushing them down. Another interpretation (maybe more positive) is that a 75 kg person can maintain this posture even with a load of 3150 Newtons (321 kg, 708 lbs) attached to their hips. This is over 4 times body weight! And it's a little easier to make them lose their balance pushing them forward, compared to pulling them backward. This analysis does not take into account other factors such as the maximum compressive load that can be supported by bones and joints. That said, the human body can produce and withstand extremely high forces. For reference, in running, the force in the Achilles tendon can be 6 to 8 times body weight and compressive forces in the lower leg can be 10 to 14 times body weight.

The following Matlab code solves the problem.

```
static_balance_data

theta_push = pi/180.*(0:1:360);
f_push_max = zeros(size(theta_push));
t_muscle = zeros(n_musc, length(theta_push));

for i = 1:length(theta_push)
    theta = theta_push(i);
    cvx_begin
    cvx_quiet(true)
        variable f_push;
        variable t(40,1);
        variables f_toe(2,1) f_heel(2,1);

        maximize(f_push);

        A_musc*t + A_heel*f_heel + A_toe*f_toe + ...
             A_push*f_push*[cos(theta); sin(theta)] == b;
        abs(f_toe(1)) <= mu*f_toe(2);
        abs(f_heel(1)) <= mu*f_heel(2);
        0 <= t;
        t <= T_max;
    cvx_end
    f_push_max(i) = f_push;
    t_muscle(:,i) = t;
end

% plot results
figure
fill(f_push_max.*cos(theta_push), f_push_max.*sin(theta_push), 'c'), hold on
```

```
xlabel('f^{push}_1 (Newtons)')
ylabel('f^{push}_2 (Newtons)')
plot([-400,400], [0,0], 'k--'), hold on
plot([0,0], [-3500,1000], 'k--')

print -depsc static_balance_fres_mat
```

The following Python code solves the problem.

```
# solution to static balance problem
import numpy as np
import cvxpy as cvx
import matplotlib.pyplot as plt
import matplotlib

from static_balance_data import *

theta_push = np.pi/180. * np.arange(360)
f_push_max = np.zeros(len(theta_push));

for i in range(len(theta_push)):
   theta = theta_push[i];
   f_push = cvx.Variable(1)
   t = cvx.Variable(40,1)
   f_toe = cvx.Variable(2,1)
   f_heel = cvx.Variable(2,1)

   constr = [A_musc*t + A_heel*f_heel + A_toe*f_toe \
       + A_push*f_push*np.array([np.cos(theta), np.sin(theta)]) == b,
       cvx.abs(f_toe[0]) <= mu*f_toe[1],
       cvx.abs(f_heel[0]) <= mu*f_heel[1],
       0 <= t,
       t <= T_max]

   p = cvx.Problem(cvx.Maximize(f_push), constr)

   p.solve(verbose = False)

   f_push_max[i] = f_push.value

# plot results
plt.figure(1)
plt.fill(f_push_max*np.cos(theta_push), f_push_max*np.sin(theta_push),'c')
```

```
plt.plot(np.array([-400,400]), np.array([0,0]), 'k--')
plt.plot(np.array([0,0]), np.array([-3500,1000]), 'k--')
plt.xlabel('$f^\\mathrm{push}_1$ (Newtons)')
plt.ylabel('$f^\\mathrm{push}_2$ (Newtons)')

plt.savefig('static_balance_fres_py.eps')

plt.show()
```

The following Julia code solves the problem.

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

using Convex, ECOS, PyPlot
solver = ECOSSolver(verbose=false);

theta_push = pi/180 * [0:359];
f_push_max = zeros(length(theta_push));

for i = 1:length(theta_push)
  theta = theta_push[i];
  f_push = Variable(1);
  t = Variable(40);
  f_toe = Variable(2);
  f_heel = Variable(2)

  constraints  = A_musc*t + A_heel*f_heel + A_toe*f_toe +
                 A_push*f_push*[cos(theta) sin(theta)]' == b;
  constraints += abs(f_toe[1]) <= mu*f_toe[2];
  constraints += abs(f_heel[1]) <= mu*f_heel[2];
  constraints += 0 <= t;
  constraints += t <= T_max;

  prob = maximize(f_push, constraints);
  solve!(prob, solver)

  f_push_max[i] = prob.optval
end

# plot results
fill(f_push_max.*cos(theta_push), f_push_max.*sin(theta_push),"c")
plot([-400,400], [0,0], "k--")
```

```
plot([0,0], [-3500,1000], "k--")
xlabel("\$f^\\mathrm{push}_1\$ (Newtons)")
ylabel("\$f^\\mathrm{push}_2\$ (Newtons)")
savefig("static_balance_fres_jl.eps")
```