EE364a: Convex Optimization I S. Boyd
June 7–8 or 8–9, 2017

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 5PM June 9, 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 (quite) straightforward. 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.

Some of the data files generate random data (with a fixed seed), which are not necessarily the same for Matlab, Python, and Julia.

1. *Transforming to a normal distribution.* We are given n samples $x_i \in \mathbf{R}$ from an unknown distribution. We seek an increasing piecewise-affine function $\varphi : \mathbf{R} \to \mathbf{R}$ for which $y_i = \varphi(x_i)$ has a distribution close to $\mathcal{N}(0,1)$. In other words, the nonlinear transformation $x \mapsto y = \varphi(x)$ (approximately) transforms the given distribution to a standard normal distribution.

   You can assume that the samples are distinct and sorted, *i.e.*, $x_1 < x_2 < \cdots < x_n$, and therefore we also have $y_1 < y_2 < \cdots < y_n$. The empirical CDF (cumulative distribution function) of y_i is the piecewise-constant function $F : \mathbf{R} \to \mathbf{R}$ given by

$$F(z) = \begin{cases} 0 & z < y_1, \\ k/n & y_k \leq z < y_{k+1}, \quad k = 1, \ldots, n-1, \\ 1 & z \geq y_n. \end{cases}$$

   The *Kolmogorov-Smirnov* distance between the empirical distribution of y_i and the standard normal distribution is given by

$$D = \sup_z |F(z) - \Phi(z)|,$$

   where Φ is the CDF of an $\mathcal{N}(0,1)$ random variable. We will use D as our measure of how close the transformed distribution is to normal. Note that D can be as small as $1/(2n)$ (but no smaller), by choosing $y_i = \Phi^{-1}((i - 1/2)/n)$.

   Note that D only depends on the n numbers $y_1, \ldots, y_n$. From these numbers we extend φ to a function on $\mathbf{R}$ using linear interpolation between these values, and extending outside the interval $[x_1, x_n]$ using the same slopes as the first and last segments, respectively. So $y_1, \ldots, y_n$ determine φ.

   Our regularization (measure of complexity) of φ is

$$R = \sum_{i=2}^{n-1} \left| \frac{y_{i+1} - y_i}{x_{i+1} - x_i} - \frac{y_i - y_{i-1}}{x_i - x_{i-1}} \right|.$$

   This is the sum of the absolute values of the change in slope of φ. Note that $R = 0$ if and only if φ has no kinks, *i.e.*, is affine.

   We will choose y_i (which defines φ) by minimizing R, subject to $D \leq D^{\max}$, where $D^{\max} \geq 1/(2n)$ is a parameter. It can be shown that the condition $y_i < y_{i+1}$ will hold automatically; but if you are nervous about this, you are welcome to add the constraint $y_i + \epsilon \leq y_{i+1}$, where ϵ is a small positive number.

   (a) Explain how to solve this problem using convex or quasiconvex optimization. If your formulation involves a change of variables or other transformation, justify it.

   (b) The file `transform_to_normal_data.*` contains the vector x (in sorted order) and its length n. Use the method of part (a) to find the optimal φ (*i.e.*, y) for $D^{\max} = 0.05$. Plot the empirical CDF of the original data x and the normal CDF

Φ on one plot, the empirical CDF of the transformed data y and the normal CDF Φ on another plot, and the optimal transformation φ on a third plot. Report the optimal value of R.

Hints. In Python and Julia, you should use the (default) ECOS solver to avoid warnings about inaccurate solutions. You can evaluate the normal CDF Φ using `normcdf.m`/`norminv.m` (Matlab), `scipy.stats.norm.cdf/ppf` (Python), or `normcdf`/`norminvcdf` in StatsFuns.jl (Julia). To plot the empirical CDFs of x and y, you are welcome to use the basic plot functions, which connect adjacent points with lines. But if you'd like to create step function style plots, you can use `ecdf.m` (Matlab), `matplotlib.pyplot.step` (Python), or `step` in PyPlot.jl (Julia).

Solution.

(a) The objective R is a convex function, since it is a sum of absolute values (an ℓ_1 norm) of an affine function of y.

Now let's look at the constraints. The first step is to express D in terms of y. Note that the supremum must occur at one of the y_k or $\pm\infty$ (this follows since in each interval $[y_k, y_{k+1})$ F is constant and Φ is monotone increasing, where we take $y_0 = -\infty$ and $y_{n+1} = \infty$). But $\Phi(y_0) = 0$ and $\Phi(y_{n+1}) = 1$, and so we have

$$D = \max_{k=1,\dots,n} \max\{|(k-1)/n - \Phi(y_k)|, |k/n - \Phi(y_k)|\},$$

This is not a convex function of y; but it is quasiconvex, as we now show.

We have $D \leq D^{\max}$ if and only if

$$k/n - D^{\max} \leq \Phi(y_k) \leq (k-1)/n + D^{\max}, \quad k = 1, \dots, n.$$

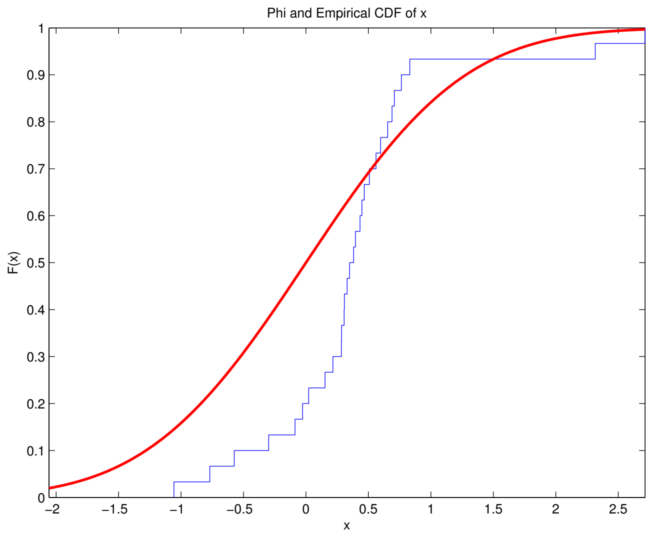
By monotonicity of Φ, this is the same as for $k = 1, \dots, n$

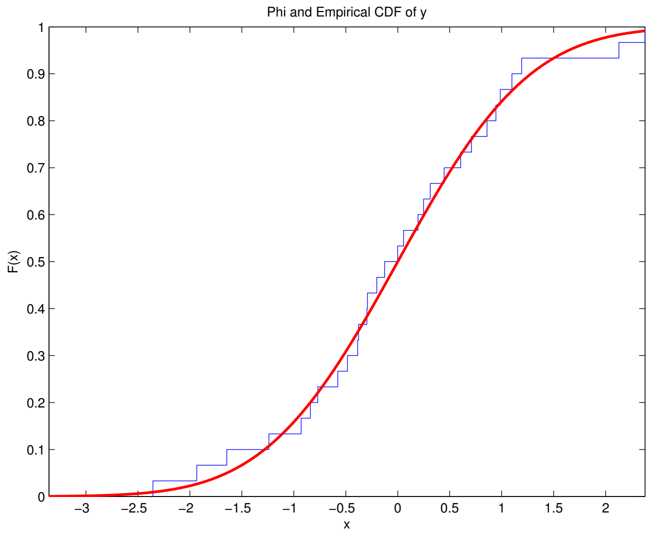
$$\Phi^{-1}(\max\{k/n - D^{\max}, 0\}) \leq y_k \leq \Phi^{-1}(\min\{(k-1)/n + D^{\max}, 1\}),$$

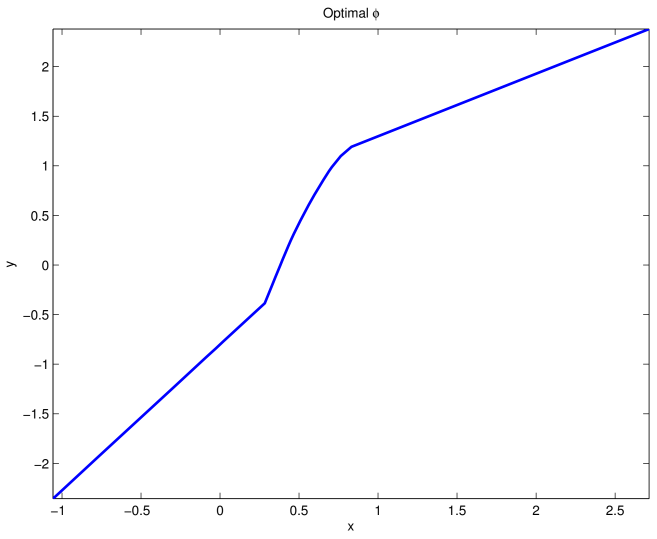
for $k = 0, \dots, n$. These are $2n$ bound constraints on y, evidently convex.

So now we have a convex problem: minimize R subject to these bound constraints.

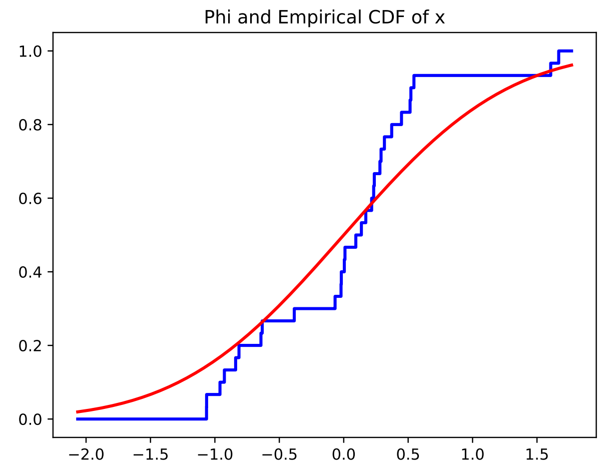
(b) For Matlab, the optimal R is 5.7391. The requested plots are shown below.

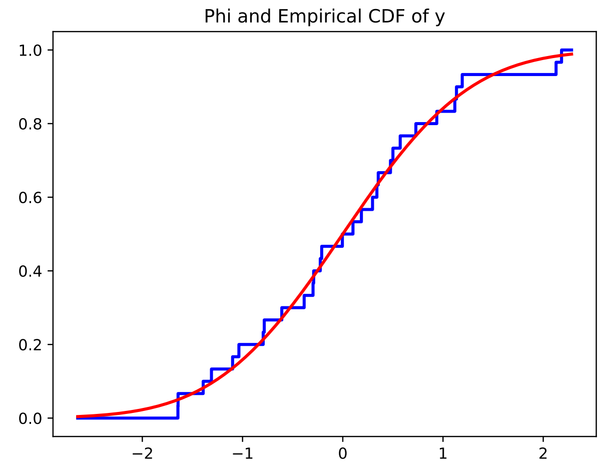

Phi and Empirical CDF of x
F(x)
x

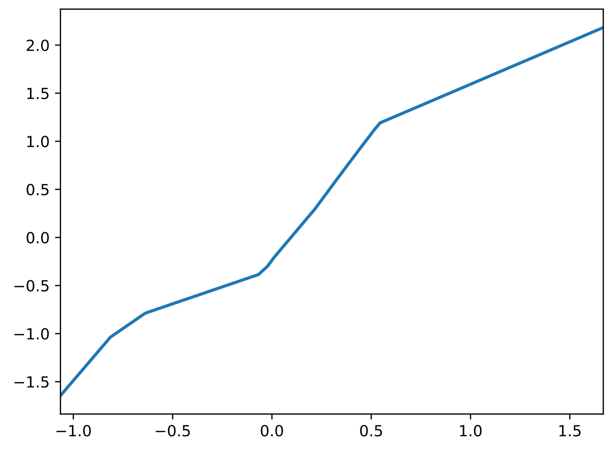

Phi and Empirical CDF of y
F(x)
x


Optimal ϕ
y
x

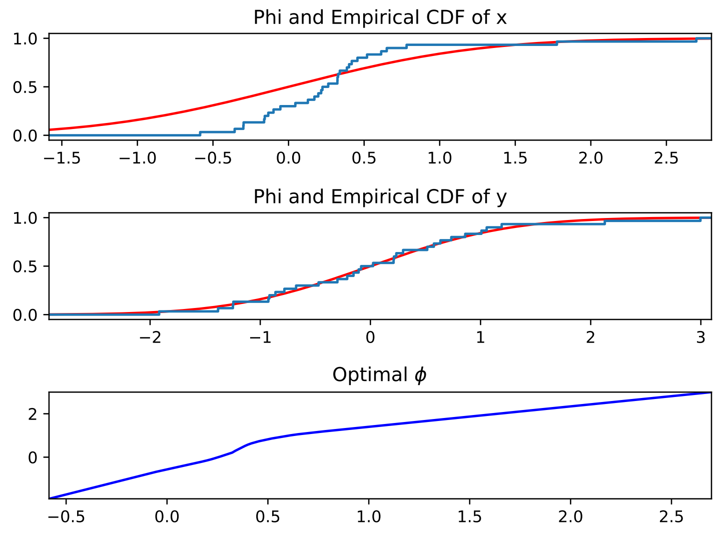
For Julia, the optimal R is 6.4386. The requested plots are shown below.







For Python, the optimal R is 8.0915. The plots are shown below.



The following Matlab code solves the problem.

```
transform_to_normal_data;
Dmax = 0.05;
lb = norminv(max([(1:n)/n-Dmax; zeros(1, n)]), 0, 1)';
ub = norminv(min([(0:(n-1))/n+Dmax; ones(1, n)]), 0, 1)';
idlb = (lb ~= -Inf); lb = lb(idlb);
idub = (ub ~= Inf); ub = ub(idub);

% CVX Solve
cvx_begin
variable y(n)
y2xdiff = (y(2 : n) - y(1 : n-1)) ./ (x(2 : n) - x(1 : n-1));
minimize(sum(abs(y2xdiff(2 : n-1) - y2xdiff(1 : n-2))));
lb <= y(idlb); y(idub) <= ub;
cvx_end

% Plots: emp-cdf of x & y, optimal phi
xnormp = linspace(min(x)-1,max(x),1000); px = normcdf(xnormp, 0, 1);
ynormp = linspace(min(y)-1,max(y),1000); py = normcdf(ynormp, 0, 1);
p1 = figure(1); ecdf(x); hold on; plot(xnormp, px, 'r', 'LineWidth', 2);
xlim([min(xnormp), max(xnormp)]); title('Phi and Empirical CDF of x');
p2 = figure(2); ecdf(y); hold on; plot(ynormp, py, 'r', 'LineWidth', 2);
xlim([min(ynormp), max(ynormp)]); title('Phi and Empirical CDF of y');
p3 = figure(3); plot(x, y, 'LineWidth', 2); xlabel('x'); ylabel('y');
xlim([min(x), max(x)]); ylim([min(y), max(y)]); title('Optimal \phi');
print(p1,'-depsc','t2n1.eps'); print(p2,'-depsc','t2n2.eps');
print(p3,'-depsc','t2n3.eps');
```

The following Julia code also solves the problem.

```
using Convex, ECOS, StatsFuns, PyPlot
include("transform_to_normal_data.jl")

# Constraints
Dmax = 0.05; z = zeros(n); e = ones(n);
lb_const = collect(1 : n) / n; lb_val = max(lb_const - Dmax, z);
ub_const = collect(0 : n-1) / n; ub_val = min(ub_const + Dmax, e);
lb = [norminvcdf(lb_val[i]) for i in 1 : n];
ub = [norminvcdf(ub_val[i]) for i in 1 : n];
idlb = collect(1:n)[lb.!=-Inf]; lb = lb[idlb];
idub = collect(1:n)[ub.!=Inf]; ub = ub[idub];


# Convex.jl Solve
y = Variable(n);
y2xdiff = (y[2:n] - y[1:n-1]) ./ (x[2:n] - x[1:n-1]);
p = minimize(sum(abs(y2xdiff[2:n-1] - y2xdiff[1:n-2])));
p.constraints += [y[idlb] >= lb; y[idub] <= ub];
solve!(p, ECOSSolver()); y = squeeze(y.value, 2);
println(p.optval);

# Plots
xl = minimum(x)-1; xr = maximum(x)+0.1;
yl = minimum(y)-1; yr = maximum(y)+0.1;
xnormp = linspace(xl,xr,1000); ynormp = linspace(yl,yr,1000);
px = [normcdf(xnormp[i]) for i in 1 : 1000];
py = [normcdf(ynormp[i]) for i in 1 : 1000];
figure(); step([xl;x;xr], [0;0;lb_const], color="blue", linewidth=2.0);
plot(xnormp, px, color="red", linewidth=2.0);
title("Phi and Empirical CDF of x"); savefig("t2n1_jl.eps");
figure(); step([yl;y;yr], [0;0;lb_const], color="blue", linewidth=2.0);
plot(ynormp, py, color="red", linewidth=2.0);
title("Phi and Empirical CDF of y"); savefig("t2n2_jl.eps");
figure(); ax = axes(); plot(x, y, linewidth=2.0);
ax[:set_xlim]([minimum(x), maximum(x)]); savefig("t2n3_jl.eps");
```

The following Python code also solves the problem.

```
import numpy as np
import scipy as sc
import cvxpy as cp
import matplotlib.pyplot as plt
from scipy.stats import norm as normal
from transform_to_normal_data import *

# Constraints
Dmax = 0.05; z = np.zeros(n, ); e = np.ones(n, )
lb_const = np.array(range(1,n+1))*1.0/n; lb_Dmax = lb_const-Dmax
ub_const = np.array(range(0,n))*1.0/n; ub_Dmax = ub_const+Dmax
lb = normal.ppf(np.amax(np.column_stack((lb_Dmax,z)), axis=1))
ub = normal.ppf(np.amin(np.column_stack((ub_Dmax,e)), axis=1))
idlb = (lb != -np.Inf); lb = lb[idlb]
idub = (ub != np.Inf); ub = ub[idub]

# CVXPY Solve
y = cp.Variable(n)
y2xdiff = cp.mul_elemwise(1 / (x[1:n] - x[0:n-1]), y[1:n] - y[0:n-1])
objective = cp.Minimize(sum(cp.abs(y2xdiff[1:n-1] - y2xdiff[0:n-2])))
constraints = [lb <= y[idlb], y[idub] <= ub]
t2n = cp.Problem(objective, constraints)
R = t2n.solve(); y = np.squeeze(np.array(y.value)); print R

# Plots
xl = np.amin(x)-1; xr = np.amax(x)+0.1;
yl = np.amin(y)-1; yr = np.amax(y)+0.1
xnormp = np.linspace(xl,xr,num=1000); px = normal.cdf(xnormp);
ynormp = np.linspace(yl,yr,num=1000); py = normal.cdf(ynormp);
figx, ax = plt.subplots(3,1);
ax[0].set_xlim((xl,xr)); ax[0].plot(xnormp, px, color='red');
ax[0].set_title('Phi and Empirical CDF of x')
ax[0].step(np.insert(x,[0,n],[xl,xr]), np.insert(lb_const,[0,0],[0,0]))
ax[1].set_xlim((yl,yr)); ax[1].plot(ynormp, py, color='red')
ax[1].set_title('Phi and Empirical CDF of y')
ax[1].step(np.insert(y,[0,n],[yl,yr]), np.insert(lb_const,[0,0],[0,0]))
ax[2].plot(x, y, color='blue'); ax[2].set_title('Optimal $\phi$')
ax[2].set_xlim((min(x),max(x))); ax[2].set_ylim((min(y),max(y)));
plt.show(); figx.tight_layout(); figx.savefig('transform_to_normal_py.eps')
```

2. *Inverse of product.* The function $f(x, y) = 1/(xy)$ with $x, y \in \mathbf{R}$, $\mathbf{dom}\, f = \mathbf{R}^2_{++}$, is convex. How do we represent it using disciplined convex programming (DCP), and the functions $1/u$, $\sqrt{uv}$, $\sqrt{u}$, u^2, u^2/v, addition, subtraction, and scalar multiplication? (These functions have the obvious domains, and you can assume a sign-sensitive version of DCP, *e.g.*, u^2/v increasing in u for $u \geq 0$.) *Hint.* There are several ways to represent f using the atoms given above.

   Solution. Here is one solution. $\sqrt{xy}$ is concave for $x, y > 0$. $1/u$ is convex and decreasing for $x > 0$, so by the DCP rules, $1/\sqrt{xy}$ is convex for $x, y > 0$. u^2 is convex and increasing for $u > 0$, so

$$(1/\sqrt{xy})^2 = 1/(xy) = f(x, y)$$

   is convex.

   Here is an alternative solution. $\sqrt{x}$ is concave for $x > 0$. $1/u$ is convex and decreasing for $x > 0$, so by the DCP rules, $1/\sqrt{x}$ is convex for $x > 0$. Quadratic-over-linear function $g(x, y) = x^2/y$ is jointly convex for $x, y > 0$, increasing in x for $x > 0$, and decreasing for $y > 0$. So

$$g(1/\sqrt{x}, y) = 1/(xy)$$

   is convex by the general vector composition rule in additional exercise 2.2.

3. *Path planning with contingencies.* A vehicle path down a (straight, for simplicity) road is specified by a vector $p \in \mathbf{R}^N$, where p_i gives the position perpendicular to the centerline at the point ih meters down the road, where $h > 0$ is a given discretization size. (Throughout this problem, indexes on N-vectors will correspond to positions on the road.) We normalize p so $-1 \leq p_i \leq 1$ gives the road boundaries. (We are modeling the vehicle as a point, by adjusting for its width.) You are given the initial two positions $p_1 = a$ and $p_2 = b$ (which give the initial road position and angle), as well as the final two positions $p_{N-1} = c$ and $p_N = d$.

   You know there may be an obstruction at position $i = O$. This will require the path to either go around the obstruction on the left, which requires $p_O \geq 0.5$, or on the right, which requires $p_O \leq -0.5$, or possibly the obstruction will clear, and the obstruction does not place any additional constraint on the path. These are the three *contingencies* in the problem title, which we label as $k = 1, 2, 3$.

   You will plan three paths for these contingencies, $p^{(i)} \in \mathbf{R}^N$ for $i = 1, 2, 3$. They must each satisfy the given initial and final two road positions and the constraint of staying within the road boundaries. Paths $p^{(1)}$ and $p^{(2)}$ must satisfy the (different) obstacle avoidance constraints given above. Path $p^{(3)}$ does not need to satisfy an avoidance constraint.

   Now we add a twist: You will not learn which of the three contingencies will occur until the vehicle arrives at position $i = S$, when the sensors will determine which contingency holds. We model this with the *information constraints* (also called *causality constraints* or *non-anticipatory constraints*),

$$p_i^{(1)} = p_i^{(2)} = p_i^{(3)}, \quad i = 1, \ldots, S,$$

   which state that before you know which contingency holds, the three paths must be the same.

   The objective to be minimized is

$$\sum_{k=1}^{3} \sum_{i=2}^{N-1} (p_{i-1}^{(k)} - 2p_i^{(k)} + p_{i+1}^{(k)})^2,$$

   the sum of the squares of the second differences, which gives smooth paths.

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

   (b) Solve the problem with data given in `path_plan_contingencies_data.*`. The data files include code to plot the results, which you should use to plot (on one plot) the optimal paths. Report the optimal objective value. Give a *very brief* informal explanation for what you see happening for $i = 1, \ldots, S$.

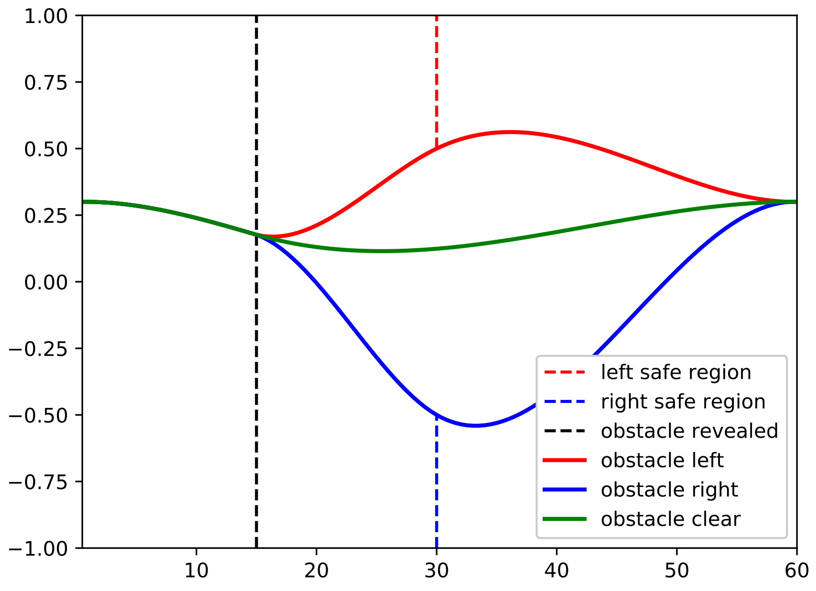
   Hint. In Python, use the (default) solver ECOS to avoid warnings about inaccurate solutions.

Solution.

(a) Note that the objective is a sum of squares of affine functions and therefore convex. The constraints are all affine. The following convex program solves the problem:

$$\begin{array}{ll}\text{minimize} & \sum_{k=1}^{3}\sum_{i=2}^{N-1}(p_{i-1}^{(k)} - 2p_i^{(k)} + p_{i+1}^{(k)})^2 \\ \text{subject to} & p_1^{(k)} = a, \quad p_2^{(k)} = b, \quad p_{N-1}^{(k)} = c, \quad p_N^{(k)} = d, \quad k = 1,2,3 \\ & -\mathbf{1} \preceq p^{(k)} \preceq \mathbf{1}, \quad k = 1,2,3 \\ & p_i^{(1)} = p_i^{(2)} = p_i^{(3)}, \quad i = 1,\dots,S \\ & p_O^{(1)} \geq 0.5, \quad p_O^{(2)} \leq -0.5. \end{array}$$

(b) The resulting plots are listed below:



The optimal objective value is 0.000210137.

The vehicle heads towards the middle of the road before the obstacle is revealed, in order to be ready for any of the contigencies. It seems strange at first to head *towards* the obstruction, as opposed to away from it, but once you think about it, it makes sense.

The following Matlab code solves the problem.

```
path_plan_contingencies_data;

% CVX Solve
cvx_begin
variable P(N, 3);
```

```
minimize(square_pos(norm((P(3:N, :)-2*P(2:N-1, :)+P(1:N-2, :)), 'fro')));
pos_start_end = [p1; p2; pN_1; pN];
% Constraints
subject to
% Same Start & End
for i = 1 : 3
    P([1, 2, N-1, N], i) == pos_start_end;
end
% Equality before Reveal
P(3 : S, 1) == P(3 : S, 2); P(3 : S, 2) == P(3 : S, 3);
% Avoid Obstruction
P(O, 1) >= 0.5; P(O, 2) <= -0.5;
% Boundary Constraints
P >= -1; P <= 1;
cvx_end
```

The following Julia code also solves the problem.

```
using Convex, ECOS, PyPlot
include("path_plan_contingencies_data.jl")

# Convex.jl Solve
P = Variable(N, 3);
p = minimize(square(vecnorm((P[3:N, :]-2*P[2:N-1, :]+P[1:N-2, :]), 2)));
pos_start_end = [p1; p2; pN_1; pN];
p.constraints += [P[[1, 2, N-1, N], i] == pos_start_end for i in 1:3];
p.constraints += [P[3:S, 1] == P[3:S, 2], P[3:S, 2] == P[3:S, 3]];
p.constraints += [P[O, 1] >= 0.5, P[O, 2] <= -0.5];
p.constraints += [P >= -1, P <= 1];
solve!(p, ECOSSolver()); println(p.optval); P = P.value;
```

The following Python code also solves the problem.

```
from cvxpy import *
from path_plan_contingencies_data import *

P = Variable(N, 3)
obj = Minimize(square(norm((P[2:, :]-2*P[1:N-1, :]+P[:N-2, :]), "fro")))
pos_start_end = np.array([p1, p2, pN_1, pN])

# Constraints
constraints = []
# Same Start & End
constraints += [P[[0,1,N-2,N-1], i] == pos_start_end for i in range(0,3)]
# Equality before Reveal
```

```
constraints += [P[2 : S, 0] == P[2 : S, 1]]
constraints += [P[2 : S, 1] == P[2 : S, 2]]
# Avoid Obstruction
constraints += [P[O - 1, 0] >= 0.5]
constraints += [P[O - 1, 1] <= -0.5]
# Boundary Constraints
constraints += [P >= -1, P <= 1]

# Solve by CVXPY
prob = Problem(obj, constraints)
prob.solve(verbose=True)
```

4. *Total variation de-mosaicing.* A color image is represented by 3 $m \times n$ matrices R, G, and B that give the red, green, and blue pixel intensities. A camera sensor, however, measures only *one* of the color intensities at each pixel. The pattern of pixel sensor colors varies, but most of the patterns have twice as many green sensor pixels as red or blue. A common arrangement repeats the 2×2 block

$$\begin{array}{cc} \mathrm{R} & \mathrm{G} \\ \mathrm{G} & \mathrm{B} \end{array}$$

(assuming m and n are even).

De-mosaicing is the process of guessing, or interpolating, the missing color values at each pixel. The sensors give us mn entries in the matrices R, G, and B; in de-mosaicing, we guess the remaining $2mn$ entries in the matrices.

First we describe a very basic method of de-mosaicing. For each 2×2 block of pixels we have the 4 intensity values

$$\begin{array}{cc} R_{i,j} & G_{i,j+1} \\ G_{i+1,j} & B_{i+1,j+1} \end{array}.$$

We use the value $R_{i,j}$ as the red value for the other three pixels, and we do the same for the blue value $B_{i+1,j+1}$. For guessing the green values at i, j and $i+1, j+1$, we simply use the average of the two measured green values, $(G_{i,j+1} + G_{i+1,j})/2$.

A more sophisticated method relies on convex optimization. You choose the unknown pixel values in R, G, and B to minimize the total variation of the color image, defined as

$$\sum_{i=1}^{m-1} \sum_{j=1}^{n-1} \left\| \begin{bmatrix} R_{i,j} - R_{i,j+1} \\ G_{i,j} - G_{i,j+1} \\ B_{i,j} - B_{i,j+1} \\ R_{i+1,j} - R_{i,j} \\ G_{i+1,j} - G_{i,j} \\ B_{i+1,j} - B_{i,j} \end{bmatrix} \right\|_2 .$$

Note that the norms in the sum here are *not* squared. The argument of the norms is a vector in $\mathbf{R}^6$, an estimate of the spatial gradient of the RGB values.

We have provided you with several files in the data directory. Three images are given (in png format): `demosaic_raw.png`, which contains the raw or mosaic image to de-mosaic, `demosaic_original.png`, which contains the original image from which the raw image was constructed, and `demosaic_simple.png`, which is the image de-mosaiced by the simple method described above. Remember that the raw image, and any reconstructed de-mosaiced image, have only one third the information of the original, so we cannot expect them to look as good as the original. You don't need the original or basic de-mosaiced image files to solve the problem; they are given only so you can look at them to see what they are. You should zoom in while viewing the raw image and the basic de-mosaic version, so you can see the pattern of 2×2 blocks in the first, and the simple de-mosaic method in the second.

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.

The file `demosaic_data.*` constructs arrays `R_mask`, `G_mask`, and `B_mask`, which contain the indices of pixels whose values we know in the original image, the number of rows and columns in the image, m, n respectively, and arrays `R_raw`, `B_raw`, `G_raw`, which contain the known values of each color at each pixel, filled in with zeroes for the unknown values. So if `R` is an $m \times n$ matrix variable, the constraint `R[R_mask]==R_raw[R_mask]` in Julia and Python will impose the constraint that it agrees with the given red pixel values; in Matlab, the constraint can be expressed as `R(R_mask)==R_raw(R_mask)`. This file also contains a `save_image` method, which takes three arguments, `R`, `G`, `B` arrays (that you've reconstructed) and saves the file under the name `output_image.png`. To see the image in Matlab, use the `imshow` function.

Report the optimal value of total variation, and attach the de-mosaiced image. (If you don't have access to a color printer, you can submit a monochrome version. Print it large enough that we can see it, say, at least half the page width wide.)

Hint. Your solution code should take less than 10 seconds or so to run in Python and Matlab, but up to a minute or so in Julia. You might get a warning about an inaccurate solution, but you can ignore it.

Solution. Letting R, G, B denote the matrices for the de-mosaiced image, and letting $R^{\text{raw}}, B^{\text{raw}}, G^{\text{raw}}$ denote the matrices for the known color values, with $K_{\text{red}}, K_{\text{green}}, K_{\text{blue}}$ denoting the indices for the known red, green, and blue pixels, respectively, then demosaicing can be written in the following form:

$$\begin{array}{ll} \text{minimize} & \mathbf{tv}(R, G, B) \\ \text{subject to} & R_{ij} = R_{ij}^{\text{raw}}, \quad (i,j) \in K_{\text{red}} \\ & G_{ij} = G_{ij}^{\text{raw}}, \quad (i,j) \in K_{\text{green}} \\ & B_{ij} = B_{ij}^{\text{raw}}, \quad (i,j) \in K_{\text{blue}}. \end{array}$$

The following code solves the problem in Python.

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

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

```
tv_obj = Minimize(tv(R, G, B))
cons = []
cons += [R[R_mask] == R_raw[R_mask]]
cons += [G[G_mask] == G_raw[G_mask]]
cons += [B[B_mask] == B_raw[B_mask]]

prob = Problem(tv_obj, cons)
prob.solve()

print('optimal value : {}'.format(prob.value))

save_image(R.value, G.value, B.value)
```

In Julia:

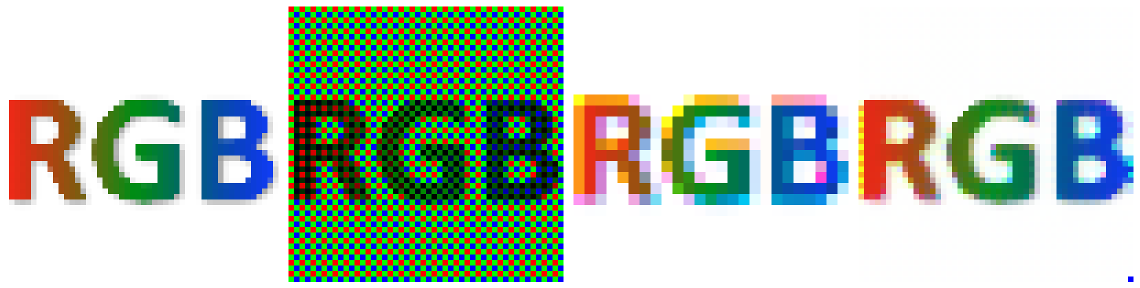
```
using Convex
include("demosaic_data.jl")
include("tv.jl")

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

constraints = [
    R[R_mask] == R_raw[R_mask];
    G[G_mask] == G_raw[G_mask];
    B[B_mask] == B_raw[B_mask];
];

print("everything is loaded");

prob = minimize(tv(R, G, B), constraints);
solve!(prob);

save_image(R.value, G.value, B.value);
```

In Matlab:

```
demosaic_data;

cvx_begin quiet
```

```
    variables R(m,n) G(m,n) B(m,n)
    minimize tv(R,G,B)
    subject to
        % Reconstruction matches the raw R, G, B values known
        R(R_mask) == R_raw(R_mask)
        G(G_mask) == G_raw(G_mask)
        B(B_mask) == B_raw(B_mask)
cvx_end

save_image(R,G,B);
```

With the results shown below:



The original image is shown on the left, with the raw image following. The image reconstructed with the simple blocking method for de-mosaicing described is third, while the last image is the TV-reconstructed image.

In Python and Matlab and Julia, we received an optimal value of 354.51.

5. *Maximum Sharpe ratio portfolio.* We consider a portfolio optimization problem with portfolio vector $x \in \mathbf{R}^n$, mean return $\mu \in \mathbf{R}^n$, and return covariance $\Sigma \in \mathbf{S}^n_{++}$. The ratio of portfolio mean return $\mu^T x$ to portfolio standard deviation $\|\Sigma^{1/2}x\|_2$ is called the *Sharpe ratio* of the portfolio. (It is often modified by subtracting a risk-free return from the mean return.) The Sharpe ratio measures how much return you get per risk taken on, and is a widely used single metric that combines return and risk. It is undefined for $\mu^T x \leq 0$.

   Consider the problem of choosing the portfolio to maximize the Sharpe ratio, subject to the constraint $\mathbf{1}^T x = 1$, and the leverage constraint $\|x\|_1 \leq L^{\max}$, where $L^{\max} \geq 1$ is a given leverage limit. You can assume there is a feasible x with $\mu^T x > 0$.

   (a) Show that the maximum Sharpe ratio problem is quasiconvex in the variable x.

   (b) Show how to solve the maximum Sharpe ratio problem by solving *one* convex optimization problem. You must fully justify any change of variables or problem transformation.

   Solution.

   Full disclosure. The problem statement above is very slightly modified from the original one, by replacing $\mathbf{S}_+$ with $\mathbf{S}_{++}$, and requiring that $\mu^T x > 0$. Only one student (and unfortunately, not an alpha or beta tester) discovered that these slightly stronger conditions are actually required. We thank him, and made these small changes so the problem will be air-tight as it continues its life as an exercise. It goes without saying that we did not penalize others (including the Professor and TAs) for missing these corner cases.

   (a) The constraints are evidently convex, so we only need to show that the Sharpe ratio is quasiconcave, since this is a maximization problem. This means that its superlevel sets, $S_\alpha = \{x \mid \mu^T x > 0,\ \mu^T x/\|\Sigma^{1/2}x\|_2 \geq \alpha\}$, are convex for any α. (The inequality $\mu^T x > 0$ defines the domain of the Sharpe ratio.) For $\alpha \leq 0$, $S_\alpha = \emptyset$, which is convex. So we can focus on the case $\alpha > 0$.

   From $\mu^T x > 0$ we know that $x \neq 0$. Its α-superlevel set is defined by

$$\frac{\mu^T x}{\|\Sigma^{1/2}x\|_2} \geq \alpha \Longleftrightarrow \mu^T x \geq \alpha \|\Sigma^{1/2}x\|_2,$$

   which is a convex constraint since $\alpha \geq 0$.

   (b) We will use the same trick used to solve a linear fractional program by solving one linear program.

   We first observe that the Sharpe ratio is (positively) homogeneous, *i.e.*, x and αx have the same Sharpe ratio, provided $\alpha > 0$. We rewrite the constraint $\|x\|_1 \leq L^{\max}$ as

$$\|x\|_1 \leq L^{\max}\mathbf{1}^T x,$$

which is also positively homogeneous. This is the homogeneous form of the leverage limit. We can also add the (redundant, positive homogeneous) constraint $\mu^T x > 0$. So the problem becomes

$$\begin{array}{ll} \text{maximize} & \mu^T x / \|\Sigma^{1/2} x\|_2 \\ \text{subject to} & \|x\|_1 \leq L^{\max} \mathbf{1}^T x, \quad \mu^T x > 0, \quad \mathbf{1}^T x = 1. \end{array}$$

The objective and constraints are all homogeneous except for $\mathbf{1}^T x = 1$.

We will change variables using z, defined as $z = x/\mu^T x$. (Note that this is a positive multiple of x.) The variable z satisfies $\mu^T z = 1$; we can recover x from z as $x = z/\mathbf{1}^T z$. Using z the problem becomes

$$\begin{array}{ll} \text{maximize} & 1/\|\Sigma^{1/2} z\|_2 \\ \text{subject to} & \|z\|_1 \leq L^{\max} \mathbf{1}^T z, \quad \mu^T z = 1. \end{array}$$

The objective can be replaced with minimizing $\|\Sigma^{1/2} z\|_2$, which results in the convex problem

$$\begin{array}{ll} \text{minimize} & \|\Sigma^{1/2} z\|_2 \\ \text{subject to} & \|z\|_1 \leq L^{\max} \mathbf{1}^T z, \quad \mu^T z = 1. \end{array}$$

We solve this, then recover the portfolio that maximizes Sharpe ratio as $x^\star = z^\star / \mathbf{1}^T z^\star$.

There are several other ways to use the same argument, and we accepted all of them. The critical part was to use a linear fractional or perspective change of variable, and to explain how to recover the original portfolio vector from the solution of the convex problem.

6. *Optimizing a set of disks.* A disk $D \subset \mathbf{R}^2$ is parametrized by its center $c \in \mathbf{R}^2$ and its radius $r \geq 0$, with the form $D = \{x \mid \|x - c\|_2 \leq r\}$. (We allow $r = 0$, in which case the disk reduces to a single point $\{c\}$.) The goal is to choose a set of n disks $D_1, \ldots, D_n$ (*i.e.*, specify their centers and radii), to minimize an objective subject to some constraints.

   One constraint is that the first k disks are fixed, *i.e.*,

$$c_i = c_i^{\text{fix}}, \quad r_i = r_i^{\text{fix}}, \quad i = 1, \ldots, k,$$

   where c_i^{fix} and r_i^{fix} are given.

   The second constraint is an overlap or intersection constraint, which requires some pairs of disks to intersect:

$$D_i \cap D_j \neq \emptyset, \quad (i,j) \in \mathcal{I},$$

   where $\mathcal{I} \subset \{1, \ldots, n\}^2$ is given. You can assume that for each $(i,j) \in \mathcal{I}$, $i < j$.

   We consider two objectives: The sum of the disk areas, and the sum of the disk perimeters. These two objectives result in two different problems.

   (a) Explain how to solve these two problems using convex optimization.

   (b) Solve both problems for the problem data given in `disks_data.*`. Give the optimal total area, and the optimal total perimeter. Plot the two optimal disk arrangements, using the code included in the data file. Give a *very brief* comment on the results, especially the distribution of disk radii each problem obtains.

   Solution.

   (a) We consider the first objective, minimizing the sum of the disk areas. The objective function $\sum_{i=1}^n \pi r_i^2$ is convex in the variables $c_1, \ldots, c_n$ and $r_1, \ldots, r_n$. The second objective, the sum of the disk perimeters $\sum_{i=1}^n 2\pi r_i$ is also convex (in fact, affine).

   Now we consider the constraints. The constraint that the first k disks are fixed are a set of linear constraints. The second constraint, the intersection constraint, can be written as $\|c_i - c_j\|_2 \leq r_i + r_j$ for each $(i,j) \in \mathcal{I}$. It is convex because the left hand side is convex, and the right hand side is linear.

   The first problem is

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^n \pi r_i^2 \\ \text{subject to} & c_i = c_i^{\text{fix}}, \quad r_i = r_i^{\text{fix}}, \quad i = 1, \ldots, k \\ & r_i \geq 0, \quad i = 1, \ldots, n \\ & \|c_i - c_j\|_2 \leq r_i + r_j, \quad (i,j) \in \mathcal{I}. \end{array}$$

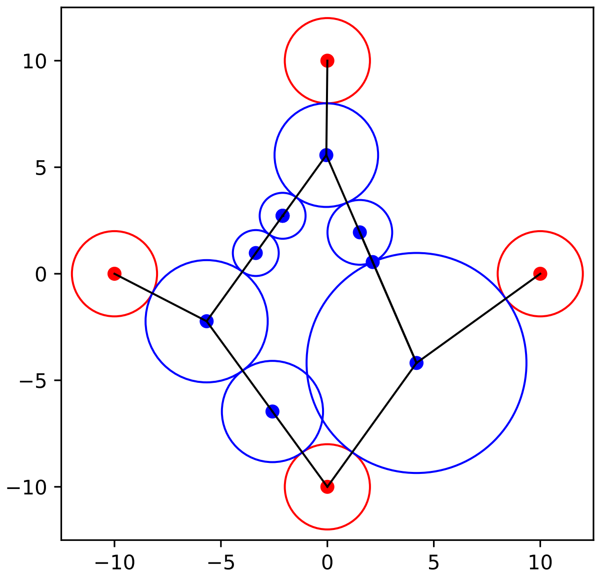
The second problem is

$$\begin{array}{ll} \text{minimize} & \sum_{i=1}^{n} 2\pi r_i \\ \text{subject to} & c_i = c_i^{\text{fix}}, \quad r_i = r_i^{\text{fix}}, \quad i = 1, \ldots, k \\ & r_i \geq 0, \quad i = 1, \ldots, n \\ & \|c_i - c_j\|_2 \leq r_i + r_j, \quad (i,j) \in \mathcal{I}. \end{array}$$

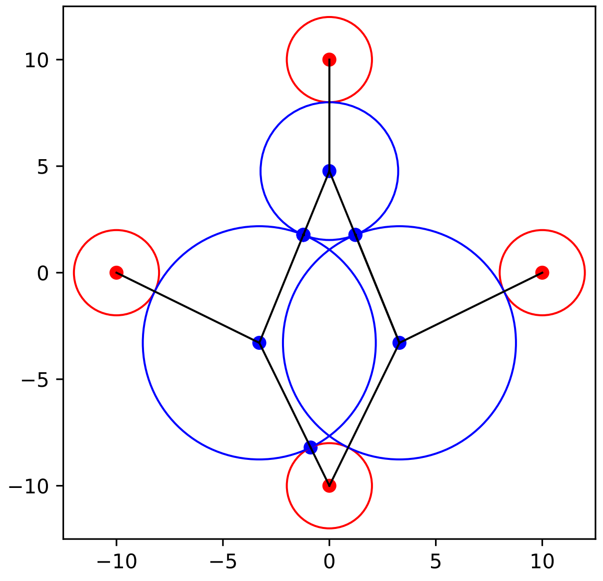
Alternative solution. Here is a nice alternative solution. Instead of explicitly working out the condition under which two disks intersect, we introduce a new variable $x_{ij} \in \mathbf{R}^2$ for each $(i,j) \in \mathcal{I}$, and we require that this point is in disks i and j. The latter is convex, since it has the form $\|x - c\|_2 \leq r$.

(b) The optimal total area is 210.77, and the optimal total perimeter is 139.35.

The resulting plot for optimal area disks is below.



The resulting plot for optimal perimeter disks is below.



We observe that when minimizing the perimeter, which is the sum, and therefore also the ℓ_1 norm, of the radii, leads to a solution where a few disks have large radii and several disks have radius zero. This is characteristic of ℓ_1 minimization. Minimizing the area, which is the ℓ_2 norm squared of the radii, leads to a solution with fewer large radii, and none with zero radii.

The following Matlab code solves the problem.

```
disks_data;

% minimum area
cvx_begin quiet
variables C(n, 2) R(n)
minimize(sum(R.^2))
subject to
R >= 0;
C(1:k,:) == Cgiven(1:k,:);
R(1:k) == Rgiven(1:k);
for i = 1 : size(Gindexes, 1)
    norm(C(Gindexes(i,1),:) - C(Gindexes(i,2),:)) ...
        <= (R(Gindexes(i,1)) + R(Gindexes(i,2)));
end
cvx_end
fprintf('optimal area: %3.4f\n', pi * cvx_optval);
```

```
% minimum perimeter
cvx_begin quiet
variables C(n, 2) R(n)
minimize(sum(R))
subject to
R >= 0;
C(1:k,:) == Cgiven(1:k,:);
R(1:k) == Rgiven(1:k);
for i = 1 : size(Gindexes, 1)
    norm(C(Gindexes(i,1),:) - C(Gindexes(i,2),:)) ...
        <= (R(Gindexes(i,1)) + R(Gindexes(i,2)));
end
cvx_end
fprintf('optimal perimeter: %3.4f\n', 2 * pi * cvx_optval);
```

The following Julia code also solves the problem.

```
using Convex, ECOS
include("disks_data.jl")

# Convex.jl Solve: Area
C = Variable(n, 2);
R = Variable(n);
p_area = minimize(sum(R.^2));
p_area.constraints += [R >= 0];
p_area.constraints += [C[1:k,:] == Cgiven[1:k,:]];
p_area.constraints += [R[1:k] == Rgiven[1:k]];
for i = 1 : size(Gindexes, 1)
    p_area.constraints +=
    [norm(C[Gindexes[i,1],:] - C[Gindexes[i,2],:])<=
        R[Gindexes[i,1]] + R[Gindexes[i,2]]];
end
solve!(p_area, ECOSSolver());
println("optimal area: ", pi * p_area.optval);
plot_disks(C.value, R.value, Gindexes, "disks_area.eps")

# Convex.jl Solve: Perimeter
C = Variable(n, 2);
R = Variable(n);
p_peri = minimize(sum(R));
p_peri.constraints += [R >= 0];
p_peri.constraints += [C[1:k,:] == Cgiven[1:k,:]];
p_peri.constraints += [R[1:k] == Rgiven[1:k]];
for i = 1 : size(Gindexes, 1)
```

```
    p_peri.constraints +=
    [norm(C[Gindexes[i,1],:] - C[Gindexes[i,2],:])<=
        R[Gindexes[i,1]] + R[Gindexes[i,2]]];
end
solve!(p_peri, ECOSSolver());
println("optimal perimeter: ", 2 * pi * p_peri.optval);
plot_disks(C.value, R.value, Gindexes, "disks_perimeter.eps")
```

The following Python code also solves the problem.

```
from cvxpy import *
from disks_data import *

C = Variable(n, 2)
R = Variable(n)
min_area_obj = Minimize(sum_squares(R))
min_perim_obj = Minimize(sum(R))
constraints = [R >= 0]
constraints += [C[:k,:] == Cgiven[:k,:]]
constraints += [R[:k] == Rgiven[:k]]
for i in range(0, len(Gindexes)):
    constraints += [norm(
        C[Gindexes[i, 0],:]-C[Gindexes[i, 1],:])
                    <= (R[Gindexes[i, 0]]+R[Gindexes[i, 1]])]
min_total_area = Problem(min_area_obj, constraints)
min_total_perim = Problem(min_perim_obj, constraints)

opt_area = min_total_area.solve(verbose=True)
print 'optimal area: ', np.pi*opt_area
plot_disks(C.value, R.value, Gindexes, name = 'areas.eps')

opt_peri = min_total_perim.solve(verbose=True)
print 'optimal perimeter: ', 2*np.pi*opt_peri
plot_disks(C.value, R.value, Gindexes, name = 'perimeters.eps')
```

7. *Decomposing a PV array output time series.* We are given a time series $p \in \mathbf{R}^T_+$ that gives the output power of a photo-voltaic (PV) array in 5-minute intervals, over $T = 2016$ periods (one week), given in `pv_output_data.*`. In this problem you will use convex optimization to decompose the time series into three components:

   - The *clear sky output* $c \in \mathbf{R}^T_+$, a smooth daily-periodic component, which gives what the PV output would have been without clouds. This signal is 24-hour-periodic, *i.e.*, $c_{t+288} = c_t$ for $t = 1, \ldots, T - 288$. (The clear sky output is zero at night, but we will not use this prior information in our decomposition method.)
   - A *weather shading loss* component $s \in \mathbf{R}^T_+$, which gives the loss of power due to clouds. This component satisfies $0 \preceq s \preceq c$, can change rapidly, and is not periodic.
   - A *residual* $r \in \mathbf{R}^T$, which accounts for measurement error, anomalies, and other errors.

   These components satisfy $p = c - s + r$.

   We will assume that the average absolute value of the residual is no more than 4 (which is less than 1% of the average of p).

   Smoothness of c is measured by its Laplacian,

$$\mathcal{L}(c) = (c_1 - c_2)^2 + \cdots + (c_{287} - c_{288})^2 + (c_{288} - c_1)^2.$$

   (Note that the term involves c_1 and c_{288}.)

   We will choose c, s, and r by minimizing $\mathcal{L}(c) + \lambda \mathbf{1}^T s$ subject to the constraints described above, where λ is a positive parameter, that we take to be one.

   Solve this problem, and plot the resulting c, s, r, and p (which is given), on separate plots. Give the average values of c, s, and p, and the average absolute value of r (which should be 4).

   Solution. The following Matlab code solves the problem, obtaining 529.517, 4.480, and 529.038 as the average values of c, s, and p, respectively, and 4.000 as the average absolute value of r.

```
pv_modeling_data

N=24*12; %samples per day
lambda=1;

cvx_begin
    variable c(T);
    variable s(T);
    variable r(T);
    L = sum_square(c(1:N-1)-c(2:N)) + square(c(N) - c(1));
    minimize (L+lambda*sum(s));
    subject to
        0 <= s <= c
        p == c-s+r
        (1/T)*norm(r,1) <= 4
        for i = (N+1):T
            c(i) == c(i-N)
        end
cvx_end

fprintf('average optimal c value: %f\n', mean(c));
fprintf('average optimal s value: %f\n', mean(s));
fprintf('average optimal p value: %f\n', mean(p));
fprintf('average optimal absolute r value: %f\n', mean(abs(r)));

figure
subplot(4,1,1)
plot(1/N*(1:T),p)
title('p')
subplot(4,1,2)
plot(1/N*(1:T),c)
title('c')
subplot(4,1,3)
plot(1/N*(1:T),s)
title('s')
subplot(4,1,4)
plot(1/N*(1:T),r)
title('residual error')
xlabel('day')

figure
plot((5/60)*(1:N),c(1:N))
xlabel('hours')
title('c over one day')
```

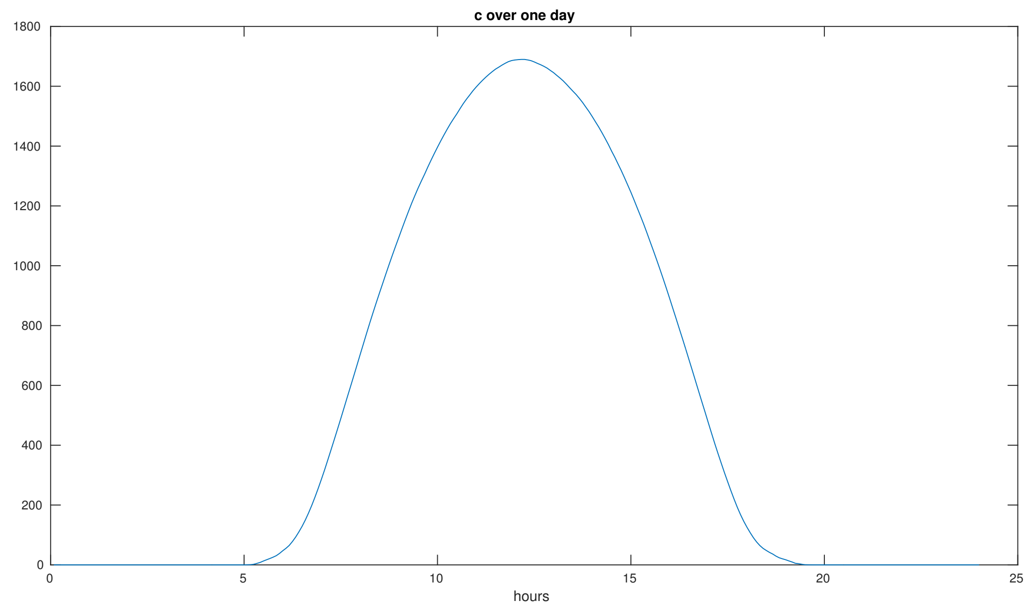
Our Python code obtains 529.520, 4.480, and 529.038 as the average values of c, s, and p, and 4.000 as the average absolute value of r.

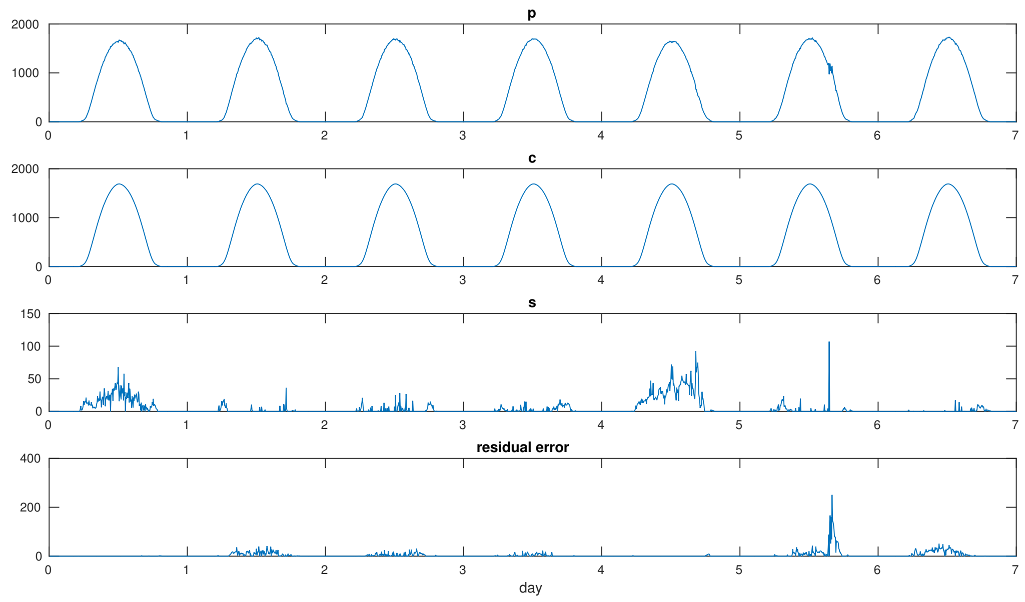
```
from cvxpy import *
import numpy as np
import numpy.matlib as mlib
import matplotlib.pyplot as ppt
from math import *
from pv_modeling_data import *

N=24*12 # samples per day
lambd = 1

c = Variable(T)
s = Variable(T)
r = Variable(T)
L = sum_squares(c[0:N-1]-c[1:N]) + square(c[N-1] - c[0]);
obj = Minimize(L + lambd*sum(s));
constraints = []
constraints += [0<=s]
constraints += [s<=c]
constraints += [p == c-s+r]
constraints += [(1.0/T)*norm(r,1) <= 4]
for i in range(N,T):
constraints += [c[i] == c[i - N]]

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

c = c.value
s = s.value
r = r.value

print ('average optimal c value: ', np.mean(c))
print ('average optimal s value: ', np.mean(s))
print ('average optimal p value: ', np.mean(p))
print ('average optimal absolute r value: ', np.mean(np.abs(r)))

ppt.figure()
ppt.subplot(411)
ppt.plot(1.0/N*np.arange(T),p)
ppt.title("p")
ppt.subplot(412)
ppt.plot(1.0/N*np.arange(T),c)
ppt.title("c")
ppt.subplot(413)
ppt.plot(1.0/N*np.arange(T),s)
ppt.title("s")
ppt.subplot(414)
ppt.plot(1.0/N*np.arange(T),r)
ppt.title("residual error")
ppt.xlabel("day")

ppt.figure()
ppt.plot((5.0/60)*np.arange(N),c[0:N])
ppt.xlabel("hours")
ppt.title("c over one day")
ppt.show()
```

And our Julia code obtains 529.526, 4.481, and 529.038 as the average values of c, s, and p, and 3.999 as the average absolute value of r.

```
using Convex
using PyPlot
using ECOS

include("pv_modeling_data.jl")

N = 24*12 #samples per day
lambda = 1

c = Variable(T)
s = Variable(T)
r = Variable(T)
L = sumsquares(c[1:N-1]-c[2:N]) + square(c[N] - c[1])
prob = minimize(L + lambda*sum(s))
prob.constraints += 0 <= s
prob.constraints += s <= c
prob.constraints += p == c- s + r
prob.constraints += (1/T)*norm(r,1)  4

for i in N + 1 : T
  prob.constraints += c[i] == c[i - N]
end
solve!(prob, ECOSSolver())

c = c.value
s = s.value
r = r.value

println("average optimal c value: ", mean(c))
println("average optimal s value: ", mean(s))
println("average optimal p value: ", mean(p))
println("average optimal absolute r value: ", mean(abs(r)))

figure()
subplot(411)
plot(1/N*(1:T),p)
title("p")
subplot(412)
plot(1/N*(1:T),c)
title("c")
subplot(413)
plot(1/N*(1:T),s)
title("s")
subplot(414)
plot(1/N*(1:T),r)
title("residual error")
xlabel("day")

figure()
plot((5/60)*(1:N),c[1:N])
xlabel("hours")
title("c over one day")
```

The plots are shown below.




p
c
s
residual error
day
2000
1000
0
150
100
50
400
200
0
1
2
3
4
5
6
7

8. *Rank one nonnegative matrix approximation.* We are given *some* entries of an $m \times n$ matrix A with positive entries, and wish to approximate it as the outer product of vectors x and y with positive entries, *i.e.*, xy^T. We will use the average relative deviation between the entries of A and xy^T as our approximation criterion,

$$\frac{1}{mn} \sum_{i=1}^{m} \sum_{j=1}^{n} R(A_{ij}, x_i y_j),$$

where R is the relative deviation of two positive numbers, defined as

$$R(u, v) = \max\{u/v, v/u\} - 1.$$

If we scale x by the positive number α, and y by $1/\alpha$, the outer product $(\alpha x)(y/\alpha)^T$ is the same as xy^T, so we will normalize x as $\mathbf{1}^T x = 1$.

The data in the problem consists of *some* of the values of A. Specifically, we are given A_{ij} for $(i,j) \in \Omega \subseteq \{1, \ldots, m\} \times \{1, \ldots, n\}$. Thus, your goal is to find $x \in \mathbf{R}^m_{++}$ (which satisfies $\mathbf{1}^T x = 1$), $y \in \mathbf{R}^n_{++}$, and $A_{ij} > 0$ for $(i,j) \notin \Omega$, to minimize the average relative deviation between the entries of A and xy^T.

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

(b) Solve the problem for the data given in `rank_one_nmf_data.*`. This includes a matrix `A`, and a set of indexes `Omega` for the given entries. (The other entries of `A` are filled in with zeros.) Report the optimal average relative deviation between A and xy^T. Give your values for x_1, y_1, and $A_{11} = x_1 y_1$.

Solution.

(a) Observe that when we fix x and y, the optimal choice of A_{ij} for $(i,j) \notin \Omega$ is simply $A_{ij} = x_i y_j$. This results in zero relative deviation between between A_{ij} and $x_i y_j$. So we can assume the variables are only x and y, and the problem is

$$\begin{array}{ll} \text{minimize} & \sum_{(i,j)\in\Omega} R(A_{ij}, x_i y_j) \\ \text{subject to} & x \succ 0, \quad y \succ 0, \quad \mathbf{1}^T x = 1. \end{array}$$

It's easy to see that $R(A_{ij}, x_i y_j)$ is not a convex function, and the objective is not convex. This is not, in its current form, a convex problem.

We'll need to transform or change the variables. We'll use $u = \log x$, $v = \log y$, so $x = \exp u$, $y = \exp v$ (interpreted entrywise). We can write

$$\begin{aligned} R(A_{ij}, x_i y_j) &= \max\{\exp(-u_i - v_j + \log A_{ij}), \exp(u_i + v_j - \log A_{ij})\} - 1 \\ &= \exp|u_i + v_j - \log A_{ij}| - 1. \end{aligned}$$

We can drop the -1 from the objective terms, and obtain the objective

$$\sum_{(i,j)\in\Omega} \exp|u_i + v_j - \log A_{ij}|$$

which is convex in u, v. The constraints $u, v \succ 0$ come for free.

The constraint $\mathbf{1}^T x = 1$ however, is not convex in the new variables, since it is $\mathbf{1}^T \exp u = 1$. Now we we use a trick: In the original problem, we could scale x by any $\alpha > 0$ and y by $1/\alpha$, without changing xy^T. With the new variables, this corresponds to adding $\beta\mathbf{1}$ to u and subtracting $\beta\mathbf{1}$ from v. So we will normalize the problem with $\mathbf{1}^T u = 0$. This leads to the problem

$$\begin{array}{ll} \text{minimize} & \sum_{(i,j)\in\Omega} \exp|u_i + v_j - \log A_{ij}| \\ \text{subject to} & \mathbf{1}^T u = 0. \end{array}$$

This normalization *does not* give $\mathbf{1}^T x = 1$; it gives $\prod_i x_i = 1$. However, we can solve the problem above (which is convex) to get $\tilde{x}, \tilde{y}$, and then set

$$x^\star = \tilde{x}/(\mathbf{1}^T x), \quad y^\star = (\mathbf{1}^T \tilde{x})\tilde{y}.$$

These scalings do not affect the objective.

Connection to geometric programming. A number of students solved the problem by reducing it to a geometric program (GP). Of course that's very close to what we did above, since in both cases you work with the log of the variables, not the variables.

You keep the original variables A, x and y, and observe that $R(A_{ij}, x_i y_j)$ is the maximum of two monomials. This is not a posynomial, but it is a generalized posynomial. For those who claimed this was a GP, we took a few points off, because it's not a GP; it's a generalzied GP. By introducing an epigraph variable for each max, though, we can reduce the problem to a GP.

Among CVX*, CVX (but not yet CVXPY or Convex.jl) has a GP mode, invoked by `cvx_begin gp`. It actually handles generalized GPs, so you could type in the problem directly. As in the other solution above, you do have to normalize the result in the end so that $\mathbf{1}^T x = 1$.

(b) The following Matlab code solves the problem: The optimal average relative deviation between the entries of A and xy^T is 0.1390, and our value for A_{11} is 0.3639.

```
rank_one_nmf_data;

% converting zero elements of A to one to prevent log(0) in the objective!
A = A + ones(m,n) - Omega;

cvx_begin
variables u(m) v(n);
obj = sum(sum( Omega .* exp( abs( u*ones(1,n) + ones(m,1)*v' - log(A)))));
minimize(obj)
subject to
```

```
sum(u) == 0;
cvx_end

x_tilde = exp(u);
y_tilde = exp(v);
x = x_tilde / sum(x_tilde);
y = sum(x_tilde) * y_tilde;

rel_dev_opt = (cvx_optval - sum(sum(Omega))) / m / n;
A_11 = x(1) * y(1);
disp(['The optimal average relative deviation = ', num2str(rel_dev_opt)]);
disp(['A_11 = ', num2str(A_11)]);
```

The following Python code solves the problem: The optimal average relative deviation between the entries of A and xy^T is 0.1839, and our value for A_{11} is 0.2728.

```
import numpy as np
import cvxpy as cp
from rank_one_nmf_data import *

# converting zero elements of A to one to prevent log(0) in the objective!
A += np.ones((m,n)) - Omega

u = cp.Variable(m)
v = cp.Variable(n)
constraints = [ cp.sum_entries(u) == 0 ]
B = cp.exp( cp.abs( u*np.ones((1,n)) + np.ones((m,1))*v.T - cp.log(A) ) )
f = 0
for i in range(m):
    for j in range(n):
        f += Omega[i,j]*B[i,j]
obj = cp.Minimize(f)
prob = cp.Problem(obj, constraints)
prob.solve()

x_tilde = np.exp(u.value)
y_tilde = np.exp(v.value)
x = x_tilde / sum(x_tilde)
y = np.sum(x_tilde) * y_tilde

rel_dev_opt = (prob.value - sum(sum(Omega))) / m / n
print('The optimal average relative deviation = ', rel_dev_opt)
```

```
A_11 = x[0,0] * y[0,0]
print('A_11 = ', A_11)
```

The following Julia code solves the problem: The optimal average relative deviation between the entries of A and xy^T is 0.2987, and our value for A_{11} is 0.1719.

```
using Convex, SCS

include("rank_one_nmf_data.jl")

# converting zero elements of A to one to prevent log(0) in the objective!
A = A + ones(m,n) - Omega;

u = Variable(m);
v = Variable(n);
obj = sum( Omega .* exp( abs( u*ones(1,n) + ones(m,1)*v' - log(A) ) ) );
constraints = [sum(u) == 0];
prob = minimize(obj, constraints);
solve!(prob)

x_tilde = exp(u.value)
y_tilde = exp(v.value)
x = x_tilde / sum(x_tilde)
y = sum(x_tilde) * y_tilde

rel_dev_opt = ( prob.optval - sum(Omega) ) / m / n;
A_11 = x[1]*y[1]
println("The optimal average relative devation = $(rel_dev_opt)")
println("A_11 = $(A_11)")
```

9. *Post-modern portfolio optimization metrics.* Let $r \in \mathbf{R}^T$ denote a time series (say, daily) of investment returns, *i.e.*, the increase in value divided by initial value. The value of the investment (typically, a portfolio) is the time series vector $v \in \mathbf{R}^T$ defined by the recursion

$$v_{t+1} = v_t(1 + r_t), \quad t = 0, \ldots, T-1,$$

with v_0 a given positive initial value. Here we are compounding the investment returns. We will assume that all returns satisfy $r_t > -1$, which implies that $v \succ 0$. We define the *high-water value* or *last high value* as

$$h_t = \max_{\tau \leq t} v_\tau, \quad t = 1, \ldots, T.$$

The value and high-water value are functions of r.

Portfolio theory as originally developed by Markowitz in the 1950s takes into account the mean return $\mu = \mathbf{1}^T r/T$ and variance (risk) $\sigma^2 = \|r - \mu\mathbf{1}\|_2^2/T$. The idea of using a mathematical approach to choose a portfolio to maximize return and minimize risk came to be called *modern portfolio theory.* Of course, it's not so modern nowadays.

Researchers later suggested various alternative metrics that are (supposedly) closer to what we really care about than the mean return and risk. The use of these metrics was dubbed (or marketed as) *post-modern portfolio theory.* Some of these so-called post-modern portfolio metrics are described below, along with a parenthetical note about whether we'd like to minimize or maximize the metric.

For each metric we wish to minimize, determine if it is a convex or quasiconvex function of r, or neither. For each metric we wish to maximize, determine if it is a concave or quasiconcave function of r, or neither. For example, the mean return (which we wish to maximize) is a concave function of r, and the risk (variance, which we wish to minimize) is a convex function of r. When the metric is convex or quasiconvex (or concave or quasiconcave), justify your answer. When it is neither, you can simply state this; you do not need to produce a counterexample. We will deduct some points if your answer is not strong enough, *e.g.*, if you assert that a metric is quasiconvex, but it is in fact convex.

(a) *Logarithmic or Kelly growth rate.* (Maximize.) $(1/T)\sum_t \log(1 + r_t)$. This is the average growth rate of v_t.

(b) *Downside variance.* (Minimize.) The downside variance is $(1/T)\sum_t (r_t - \mu)_-^2$, where $(u)_- = \max\{-u, 0\}$, and μ is the mean return. This assesses a penalty for a return below the average (the 'downside'), but not for a return above the average.

(c) *Maximum drawdown.* (Minimize.) The *drawdown* at period t is defined as $d_t = (h_t - v_t)/h_t$. The maximum drawdown is defined as $\max_t d_t$.

(d) *Maximum consecutive days under water.* (Minimize.) A time period t is called *under water* if $v_t < h_t$, *i.e.*, the current value is less than the last high. Maximum consecutive days under water means just that, *i.e.*, the maximum number of consecutive days under water.

Remark. Many other post-modern metrics can derived be from, or are related to, the ones described above. Examples include the Sortino, Calmar, and Information ratios. You can thank the EE364a staff for refraining from asking about these.

Solution.

(a) *Logarithmic or Kelly growth rate.* $f(r) = (1/T)\sum_t \log(1+r_t)$ is evidently concave, since it is a sum of the log (a concave function) of affine functions of r.

(b) *Downside variance.* This is convex, since $r_t - \mu$ is affine, $(r_t - \mu)_-$ is convex and nonnegative, and therefore its square is too, since the square function is convex and increasing for nonnegative arguments. The sum gives the downside variance.

(c) *Maximum drawdown.* This is quasiconvex. Let's look at its sublevel set, *i.e.*, the set of returns that satisfy $d_t = (h_t - v_t)/h_t \leq \alpha$ for all t, where $1 > \alpha > 0$. We write this as

$$(1-\alpha)h_t \leq v_t, \quad t = 1, \ldots, T.$$

which is equivalent to

$$v_t \geq (1-\alpha) \max_{1 \leq s \leq t} v_s, \quad t = 1, \ldots, T.$$

Using $v_t/v_s = \prod_{\tau=s+1}^{t-1}(1+r_\tau)$ we can express this as

$$\sum_{\tau=s+1}^{t-1} \log(1+r_\tau) \geq \log(1-\alpha) \text{ for } 1 \leq s \leq t \leq T.$$

Since the lefthand side is a concave function of r, for any values of s and t, this describes a convex set as the inequality above describes the $1-\alpha$ superlevel set of the lefthand side. Whew!

(d) *Maximum consecutive days under water.* This function is neither convex nor quasi-convex.

It's not that easy to show this, *i.e.*, to produce a counter-example. We didn't ask you to do this, but we've done it for completeness.

We consider $T = 4$, and the following two return vectors:

$$r = (0.01, -0.9, -0.9, 100), \qquad \tilde{r} = (-0.5, -0.5, 3.1, 0).$$

With $v_0 = 1$, the corresponding value sequences are

$$v = (1.01, 0.101, 0.0101, 1.0204), \qquad \tilde{v} = (0.5, 0.25, 1.025, 1.025).$$

We can see that each of these returns has maximum days under water two.

Now consider the return sequence

$$\hat{r} = 0.5r + 0.5\tilde{r} = (-0.245, -0.7, 1.1, 50).$$

If maximum days onder water is quasiconvex, then this return sequence should have maximum days under water of no more than two. But its corresponding value sequence is

$$\hat{v} = (1, 0.755, 0.2265, 0.47565, 24.258),$$

which has three maximum days under water.

10. *Blending overlapping covariance matrices.* We consider the problem of constructing a covariance matrix $R \in \mathbf{S}_+^n$ from two (not necessarily consistent) estimates of submatrices S and T. We order the indices in the underlying random variable so that the first n_1 entries correspond to those in the first submatrix but not the second, the next n_2 entries correspond to the entries in both submatrices, and the last n_3 entries are those in the second submatrix but not the first. We have $n_1 + n_2 + n_3 = n$, and we assume all three are positive. We partition the matrix R as

$$R = \begin{bmatrix} R_{11} & R_{12} & R_{13} \\ R_{12}^T & R_{22} & R_{23} \\ R_{13}^T & R_{23}^T & R_{33} \end{bmatrix}.$$

We wish to choose $R \in \mathbf{S}_+^n$ so that

$$R^{(1)} = \begin{bmatrix} R_{11} & R_{12} \\ R_{12}^T & R_{22} \end{bmatrix} \approx S = \begin{bmatrix} S_{11} & S_{12} \\ S_{12}^T & S_{22} \end{bmatrix}$$

and

$$R^{(2)} = \begin{bmatrix} R_{22} & R_{23} \\ R_{23}^T & R_{33} \end{bmatrix} \approx T = \begin{bmatrix} T_{22} & T_{23} \\ T_{23}^T & T_{33} \end{bmatrix}.$$

(Note the non-standard labeling of the block indices in T.) You can assume that $S \in \mathbf{S}_+^{n_1+n_2}$ and $T \in \mathbf{S}_+^{n_2+n_3}$ are given.

Roughly speaking, your job is to guess the six submatrices R_{ij} for $i \leq j$. For four of these, R_{11}, R_{12}, R_{23}, and R_{33}, you have only one piece of data to work with, *i.e.*, S_{11}, S_{12}, T_{23}, and T_{33}, respectively. For one of them, R_{22}, you have two pieces of data to work with, *i.e.*, S_{22} and T_{22}. For one submatrix, R_{13}, you have no pieces of data to work with.

(a) *A simple method.* Based on the given data S and T, our guess of R is

$$\begin{array}{lll} R_{11} = S_{11}, & R_{12} = S_{12}, & R_{13} = 0, \\ R_{22} = (1/2)(S_{22} + T_{22}), & R_{23} = T_{23}, & R_{33} = T_{33}. \end{array}$$

For the four submatrices for which you have only one piece of data, we simply use that data as our guess. For the one submatrix for which we have two pieces of data, we average the two values. For the one submatrix for which we have no data, we guess the zero matrix.

Show by a specific numerical example that this simple method can yield an *unacceptable* value of R. (No, we will not be more specific about what we mean by this; part of the problem is to figure out what we mean. Also, we will deduct points from examples that are more complicated than they need to be.)

(b) *Convex optimization to the rescue.* Suppose we choose R by solving the convex optimization problem

$$\begin{array}{ll} \text{minimize} & \|R^{(1)} - S\|_F^2 + \|R^{(2)} - T\|_F^2 + \|R_{13}\|_F^2 \\ \text{subject to} & R \succeq 0. \end{array}$$

Here the variable is $R \in \mathbf{S}^n$, and $\|U\|_F = (\mathbf{Tr}(U^T U))^{1/2}$ is the Frobenius norm of a matrix.

Let $R^{\rm sim}$ be the estimate of R obtained using the simple method in part (a). Show that if $R^{\rm sim} \succeq 0$, then it is the solution of this problem.

(c) Apply the method described in part (b) to the specific numerical example you provided in part (a), and check (numerically) that the result $R^\star$ is now acceptable.

Solution.

(a) The simple method can yield a matrix $R^{\rm sim}$ that is not positive semidefinite; it is in this sense that the method is unacceptable. For example, if $n_1 = n_2 = n_3 = 1$ and S and T are chosen such that

$$S = T = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix},$$

then the simple method yields the guess

$$R^{\rm sim} = \begin{bmatrix} 1 & 1 & 0 \\ 1 & 1 & 1 \\ 0 & 1 & 1 \end{bmatrix},$$

which is not positive semidefinite. (You can verify this by computing its eigenvalues, or simply noticing that $\det R^{\rm sim} = -1$.)

(b) Consider the unconstrained problem

$$\text{minimize} \quad \|R^{(1)} - S\|_F^2 + \|R^{(2)} - T\|_F^2 + \|R_{13}\|_F^2,$$

with variable $R \in \mathbf{S}$. We can write out the objective in terms of the blocks of R as

$$\begin{array}{l} \|R_{11} - S_{11}\|_F^2 + 2\|R_{12} - S_{12}\|_F^2 + \|R_{22} - S_{22}\|_F^2 + \\ \|R_{22} - T_{22}\|_F^2 + 2\|R_{23} - T_{23}\|_F^2 + \|R_{33} - T_{33}\|_F^2 + \|R_{13}\|_F^2 \end{array}$$

Now let's optimize over the subblocks of R separately. First, R_{13} only appears in the last term, and its optimal value is clearly $R_{13} = 0$, which agrees with the simple method of part (a). The four blocks of R for which we have only one data piece occur in only one term above each, so we simply set them to equal the associated given block, which also agrees with the simple method. Finally R_{22} appears in two terms above. Evidently the solution is to average the two matrices S_{22} and T_{22} to get R_{22}. This agrees with the simple method. So the simple method solves this unconstrained problem. If the result is positive semidefinite, then it solves the problem with the constraint too.

(c) The covariance matrix that is recovered by applying the method from part (b) to the example furnished in part (a) is shown below, with a floating point precision of 3 digits:

$$R^\star = \begin{bmatrix} 1.10 & 0.850 & 0.197 \\ 0.850 & 1.11 & 0.850 \\ 0.197 & 0.850 & 1.10 \end{bmatrix}.$$

You can check that this matrix is positive semidefinite.

The following Matlab code was used to obtain the above estimate.

```
n_1 = 1
n_2 = 1
n_3 = 1
n = n_1 + n_2 + n_3

S = ones(n_1+n_2, n_1+n_2)
T = ones(n_2+n_3, n_2+n_3)

% Recover R as per the method described in part (b)
cvx_begin
    variable R(n, n) semidefinite
    R1 = R(1:2, 1:2)
    R2 = R(2:3, 2:3)
    R_13 = R(1, 3)

    minimize(sum(sum_square(R1 - S)) + ...
        sum(sum_square(R2 - T)) +  sum(sum_square(R_13)));
cvx_end

display('Recovered covariance matrix R')
display(R)
display('Eigenvalues')
display(eig(R))

% For reference, output the matrix that the simple method would return
R_simple = [1, 1, 0; 1, 1, 1; 0, 1, 1]
display('Matrix recovered from the simple method')
display(R_simple)
```

The same estimate can be obtained using Python:

```
import numpy as np
import cvxpy as cp

n_1 = 1
```

```
n_2 = 1
n_3 = 1
n = n_1 + n_2 + n_3

S = np.ones((n_1+n_2, n_1+n_2))
T = np.ones((n_2+n_3, n_2+n_3))

R = cp.Semidef(n)
R1 = R[0:2, 0:2]
R2 = R[1:3, 1:3]
R_13 = R[0, 2]

# Recover R as per the method described in part (b)
objective = cp.Minimize(
    cp.norm(R1 - S, "fro")**2 + cp.norm(R2 - T, "fro")**2 +
    cp.norm(R_13, "fro")**2)
p = cp.Problem(objective, [])
p.solve()

print "Optimal value: %f" % p.value
print "Recovered covariance matrix R\n%s" % str(R.value)
print "Eigenvalues\n%s" % str(np.linalg.eig(R.value)[0])

# For reference, output the matrix that the simple method would return
R_simple = np.array([[1, 1, 0], [1, 1, 1], [0, 1, 1]])
print "Matrix recovered from simple method\n%s" % str(R_simple)
```

The estimate can also be obtained using Julia:

```
using Convex

n_1 = 1
n_2 = 1
n_3 = 1
n = n_1 + n_2 + n_3

S = ones(n_1+n_2, n_1+n_2)
T = ones(n_2+n_3, n_2+n_3)

# Recover R as per the method described in part (b)
R = Semidefinite(n)
R1 = R[1:2, 1:2]
R2 = R[2:3, 2:3]
R_13 = R[1, 3]
```

```
p = minimize(vecnorm(R1 - S, 2)^2 + vecnorm(R2 - T, 2)^2 + vecnorm(R_13, 2)^2)
solve!(p)

println("Optimal value ", p.optval)
println("Recovered covariance matrix R\n", R.value)
println("Eigenvalues\n", eig(R.value)[1])

# For reference, output the matrix that the simple method would return
R_simple = [1 1 0; 1 1 1; 0 1 1]
println("Matrix recovered from simple method\n", R_simple)
```