EE263, Summer 2017
July 21-23, 2017

Midterm Exam Solutions

This is a 24 hour take-home exam with 5 problems. Please turn it in on Gradescope 24 hours after you pick it up.

- You may use any books, notes, or computer programs (*e.g.*, matlab), but you may not discuss the exam with others until Jul. 24, after everyone has taken the exam. The only exception is that you can ask the course staff for clarification, by emailing to the staff email address `{k2shah, bbartan, liyues} @stanford.edu`. We've tried pretty hard to make the exam unambiguous and clear, so we're unlikely to say much. Please do not post any exam related questions on Piazza.
- Since you have 24 hours, we expect your solutions to be legible, neat, and clear. Do not hand in your rough notes, and please try to simplify your solutions as much as you can. We will deduct points from solutions that are technically correct, but much more complicated than they need to be.
- Please check your email and canvas a few times during the exam, just in case we need to send out a clarification or other announcement. It's unlikely we'll need to do this, but you never know.
- Assemble your solutions to the problems in order, *i.e.*, problem 1, problem 2, ..., problem 5. Start each solution on a new page.
- Please make a copy of your exam before handing it in. We have never lost one, but it might occur.
- If a problem asks for some specific answers, make sure they are obvious in your solutions. You might put a box around the answers, so they stand out from the surrounding discussion, justification, plots, etc.
- When a problem involves some computation (say, using matlab), we do not want just the final answers. We want a clear discussion and justification of exactly what you did, the matlab source code that produces the result, and the final numerical result. Be sure to show us your verification that your computed solution satisfies whatever properties it is supposed to, at least up to numerical precision. For example, if you compute a vector x that is supposed to satisfy $Ax = b$ (say), show us the matlab code that checks this, and the result. (This might be done by the matlab code `norm(A*x-b)`; be sure to show us the result, which should be very small.) *We will not check your numerical solutions for you, in cases where there is more than one solution.*

- In the portion of your solutions where you explain the mathematical approach, you *cannot* refer to matlab operators, such as the backslash operator. (You can, of course, refer to inverses of matrices, or any other standard mathematical construct.)

- Some of the problems are described in a practical setting, such as alien power systems or computer vision. *You do not need to understand anything about the application area to solve these problems.* We've taken special care to make sure all the information and math needed to solve the problem is given in the problem description.

- Some of the problems require you to download and run a matlab/json file to generate the data needed. These files can be found at the URL

  `http://ee263.stanford.edu/exams/mt-data.zip`

  Some test.py scripts are given to show you how to import the data

- Please respect the honor code. Although we encourage you to work on homework assignments in small groups, *you cannot discuss the exam with anyone*, with the exception of EE263 course staff, until July 24, when everyone has taken it an the solutions are posted online.

1. *And Away We Go!*

   (a) Show that the eigenvalues of A and A^T are the same

   (b) Show that if the eigenvalues of A are $\lambda_1 \dots \lambda_n$ and A is invertible, then the eigenvalues of A^{-1} are $\lambda_1^{-1} \dots \lambda_n^{-1}$

   (c) Let $x, y \in \mathbf{R}^n$. Show that if $||x|| = ||y|| = 1$ and $x^T y = 1$ then $x = y$

   (d) Let $a, b \in \mathbf{R}^n$ and $H \in S_+$ where S_+ is the set of positive semidefinite matrices, that is $x^T H x \geq 0 \quad \forall x \in \mathbf{R}^n$. Show that if $Ha = b$ and $Hb = a$ then $a = b$. *Hint*: use a norm

Solution.

(a) **4 points**

Since $det(A) = det(A^T)$ we see that $det(I\lambda - A) = det(I\lambda - A)^T = det(I\lambda - A^T)$. Thus the roots of both the polynomials are the same, therefore the eigenvalues are the same.

(b) **5 points**

$A = T\Lambda T^{-1}$ so $A^{-1} = (T\Lambda T^{-1})^{-1} = T\Lambda^{-1}T^{-1}$

it follows that $\Lambda\Lambda^{-1} = I$ thus $\Lambda^{-1} = diag(\lambda_1^{-1} \dots \lambda_n^{-1})$

Some people did showed that for a arbitrary eigenvalue of A, it can be shown that the reciprocal is an eigenvalue of A^{-1}

(c) **5 points**

Consider $||x-y||^2 = (x-y)^T(x-y) = x^Tx - x^Ty - y^Tx + y^Ty = 1-1-1+1 = 0$
Thus $x - y = 0$ since by the definiteness of norms $||z|| = 0$ *iif* $z = 0$

(d) **6 points**

since $a-b \in reals^n$ it follows from H being positive semidefinite $(a-b)^T H(a-b) \geq 0$, but $(a-b)^T H(a-b) = (a-b)^T(Ha-Hb) = (a-b)^T(b-a) = -(a-b)^T(a-b) = -||a-b||$ which must be nonpositive as norms are nonnegative. Therefore

$$0 \leq (a-b)^T H(a-b) \leq 0$$

thus a-b=0 since any other value would violate one of the two statements above.

Some people noticed that H(a-b)=b-a = -(a-b) which implies that a-b is an eigenvector of H with eigenvalue -1, which would violate the premise, thus either a-b=0. Some claimed that this means $H = -I$ which is indeed not true.

Also some claimed that since $||a|| = ||b||$ or $a^T Ha = b^T Hb$ then $a = b$, but that is also not true, as you can take $H = I$ and a, b as any unit vector

2. *You Must Construct Additional Pylons*

You are the Hierarch of the Baelaam charged with maintaining the power levels of energizing pylons which power various structures in your base of operations. Consider m structures powered by n pylons. Each structure's energy level y_j for $j = 1...m$ is given by

$$y_j(p) = log(\sum_{i=1}^{n} \exp(\frac{p_i}{d_{j,i}^2}))$$

Where p_i are the power levels of the i'th pylon and $d_{j,i}$ are the distances between the j'th structure and the i'th pylon (we choose log-sum-exp as a smooth approximation of the max function). While each structure has some given target energy level R_j, they can handle some deviation (either over or under), however that will cause damage to the Nexus Crystals that act as energy conduits for the structure. Your goal as Hierarch is to find a set of pylon power levels $\mathbf{p} \in \mathbf{R}^n$ that minimizes the total square deviation, J, from the required energy levels.

$$J(p) = \sum_{j=1}^{m} (R_j - y_j(p))^2$$

Your chief engineer proposes that you could linearize the $y_j(p)$ function to find an update algorithm that starts with some initial pylon power level and changes the power each step by a small amount to reduce the total energy deviation J.

(a) Find an update expression for the approximate power level $\mathbf{y}(p + \delta p)$ as a linear dynamical system where $\mathbf{y} \in \mathbf{R}^m$ is the vector of structure energy levels. I.E find A and B such that

$$\mathbf{y}(p + \delta p) \approx A\mathbf{y}(p) + B\delta p$$

We want to relate the energy level at $p+\delta p$ to the energy level at p and the change in energy from a small change in power δp.

hint: B is not necessarily constant

(b) Derive an expression for the one step change in power levels that minimizes

$$J(p + \delta p) = \sum_{j=1}^{m} (R_j - y_j(p + \delta p))^2$$

as a function of $y(p), A, B, \delta p$. Use the result of this minimization problem, (the optimal δp) to determine an update expression for $p[k + 1] = p[k] + \alpha\delta p$, where α is a given step size, and k is the current iteration. If your method involves an inverse, explain what conditions must hold in order for the inverse to exist.

(c) Given the following list of required energy levels and locations of each structure and pylon, apply your algorithm for 200 iterations with an $\alpha = .01$ and in initial power level of $p[0]$ =[20, 40, 20].

Plot the pylon power levels and the structures energy levels for each iteration. as well as the the power deviation metric J. There should be 3 plots total. It should converge in roughly 150-200 iterations.

Also report the final cost and pylon power levels

```
Stucture_energy_goal=[10, 20 , 5 , 10 , 5]
Strcuture_location=[2 8; 4 5; 6 8; 2 2; 4 1]
Pylon_location=[2 5; 3 4 ; 5 4]
p0=[20 40 20]
```

For locations, each row is an x,y location.

Solution.

(a) **6 points**

First we linearize the energy function by finding the Jacobian of the energy function. Clearly $A = I$, but B is a bit more involved

$$\mathbf{y}(p + \alpha\delta p) = \mathbf{y}(p) + B\delta p = \mathbf{y}(p) + \begin{bmatrix} \frac{\partial y_1}{\partial p_1} & \cdots & \frac{\partial y_1}{\partial p_n} \\ \vdots & \vdots & \vdots \\ \frac{\partial y_m}{\partial p_1} & \cdots & \frac{\partial y_m}{\partial p_n} \end{bmatrix} \delta p$$

now we find the partial derivatives

$$\frac{\partial y_j}{\partial p_i} = \frac{\exp(\frac{p_i}{d_{j,i}^2})}{\sum_{i=1}^{n} \exp(\frac{p_i}{d_{j,i}^2})} \frac{1}{d_{j,i}^2}$$

We see that B is now just a function of p. Partial Credit was given to those who attempted to find B

(b) **7 points**

We can substitute our expression for $\mathbf{y}[k+1]$ here

$$J[k+1] = ||(R - (\mathbf{y}(p) + B\delta p)||^2 = ||z - H\delta p||^2$$

Where z is the difference between $R - y[k]$ (the last error vector) and H is the Jacobian matrix of the energy function Which yields $\delta p(H^T H)^{-1} H^T z$ as the expression that minimizes the one step power update. Thus the update to the pylon power is given by

$$\mathbf{y}[k+1] = \mathbf{y}[k] + \alpha(H^T H)^{-1} H^T (R - \mathbf{y}[k])$$

(c) **7 points** Final pylon power levels after 200 iterations
[81.3, 35.2, 34.7]
Final cost 3.7
for 50 iterations
Final pylon power levels [52.7, 40.3, 35.7]
Final cost 24.1
Code (in Julia) that solves this problem

```
using Gadfly, ColorBrewer
#structures
R_loc=Float64[2 8; 4 5; 6 8; 2 2; 4 1]
R_goal=[10, 20 , 5 , 10 , 5 ]
nR=size(R_loc, 1)
#pylons
P_loc=Float64[2 5; 3 4 ; 5 4]
p0=Float64[20 40 20]
nP=size(P_loc, 1);

#show world
layers=Layer[]
append!(layers, layer(x=R_loc[:, 1], y=R_loc[:,2], Geom.point,
        Theme(default_color=colorant"red") ))
append!(layers, layer(x=P_loc[:, 1], y=P_loc[:,2], Geom.point,
        Theme(default_color=colorant"blue") ))
plot(layers..., )

#distances squared
d=[norm(R_loc[j,:]-P_loc[i,:])^2 for j=1:nR, i=1:nP]

function evalPower(p, d)
    return [log(sum([exp(p[j]/d[i,j]) for j in 1:size(d,2)]))
        for i in 1:size(d, 1)]
end
function makeGrad(p, d)
    return [sum(exp(p./d[i,:]))^-1 * exp(p[j]/d[i,j]) / d[i,j]
        for i in 1:size(d, 1), j in 1:size(d, 2) ]
end

t=200
p_hist=zeros(length(p0), t+1)
R_hist=zeros(nR, t)
j_hist=zeros(t)
p_hist[:, 1]= p0
```

```
for i=1:t
    p=p_hist[:,i]
    R_hist[:,i]=evalPower(p, d)
    y=R_goal-evalPower(p, d)
    j_hist[i]=norm(y)
    A=makeGrad(p, d)
    dp=pinv(A)*y
    p_hist[:,i+1]=p+.01*dp
end

@show p_hist[:,end]
norm(R_goal-evalPower(p_hist[:,end], d))^2 #last error


gcol=greens = palette("Set1", nP);
layers=[layer(x=0:t, y=p_hist[pylon, :], Geom.path,
    Theme(default_color=gcol[pylon])) for pylon=1:nP]
plot(layers...,
    Guide.xlabel("iter"), Guide.ylabel("Power"),
    Guide.title("Pylon Power"))

#structure
gcol=greens = palette("Set1", nR);
layers=[layer(x=1:t, y=R_hist[R, :], Geom.path,
    Theme(default_color=gcol[R])) for R=1:nR]
plot(layers...,
    Guide.xlabel("iter"), Guide.ylabel("Energy"),
    Guide.title("Structure Energy"))

#error
plot(x=1:t, y=j_hist, Geom.path,
    Guide.xlabel("iter"), Guide.ylabel("Error"),
    Guide.title("Energy Deviation"))
```

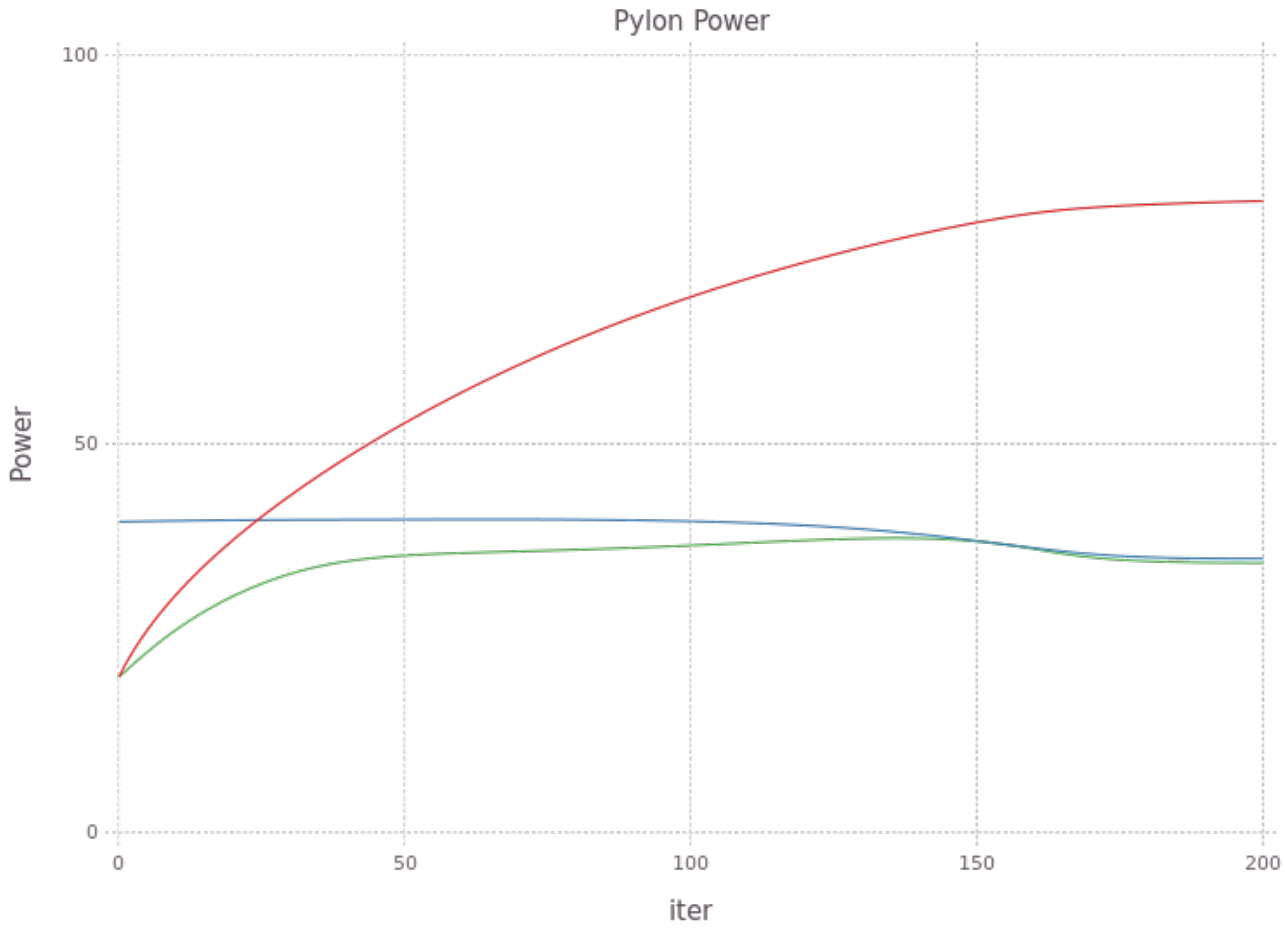


Figure 1: pylon power levels

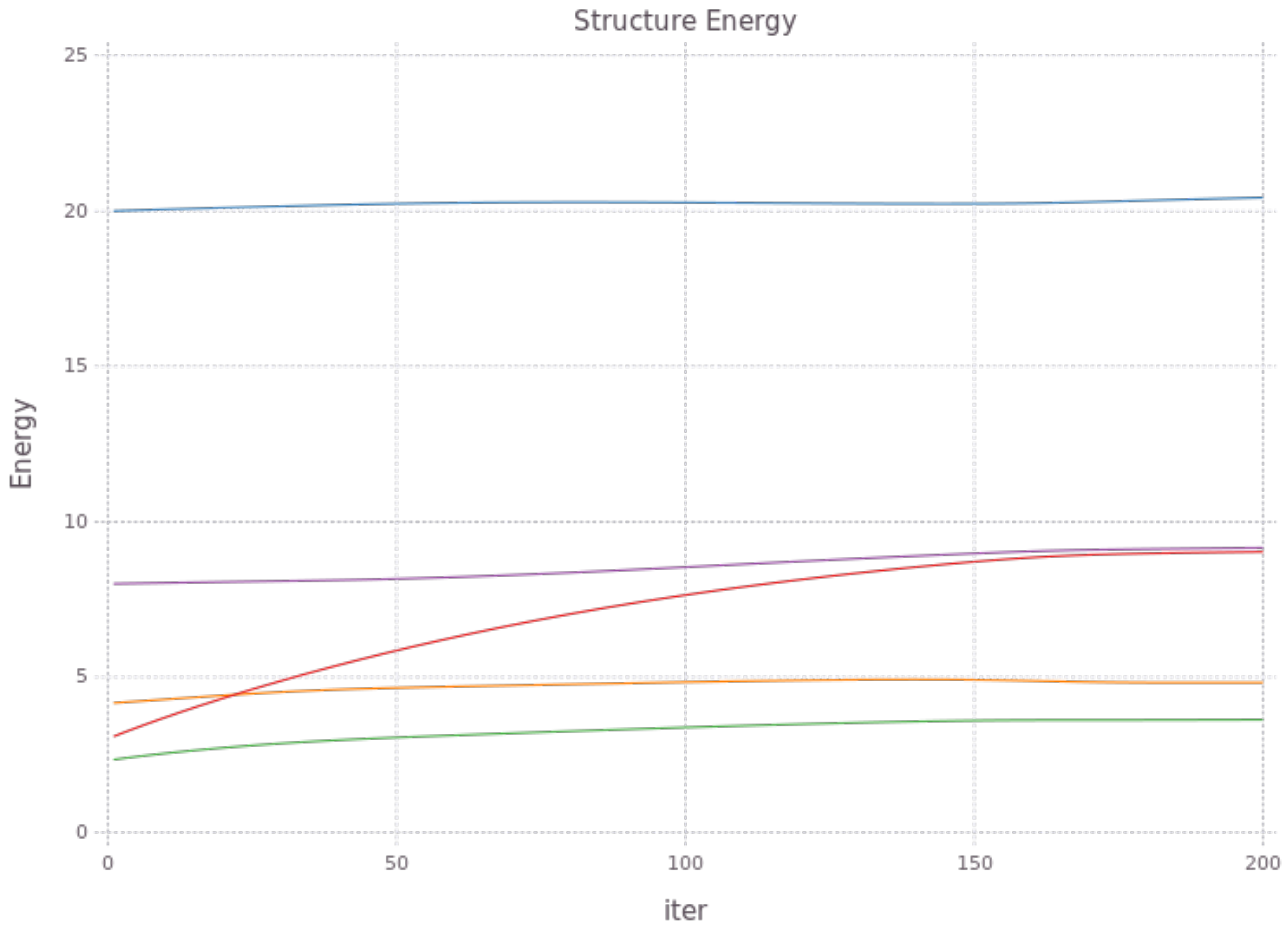


Figure 2: structure energy levels

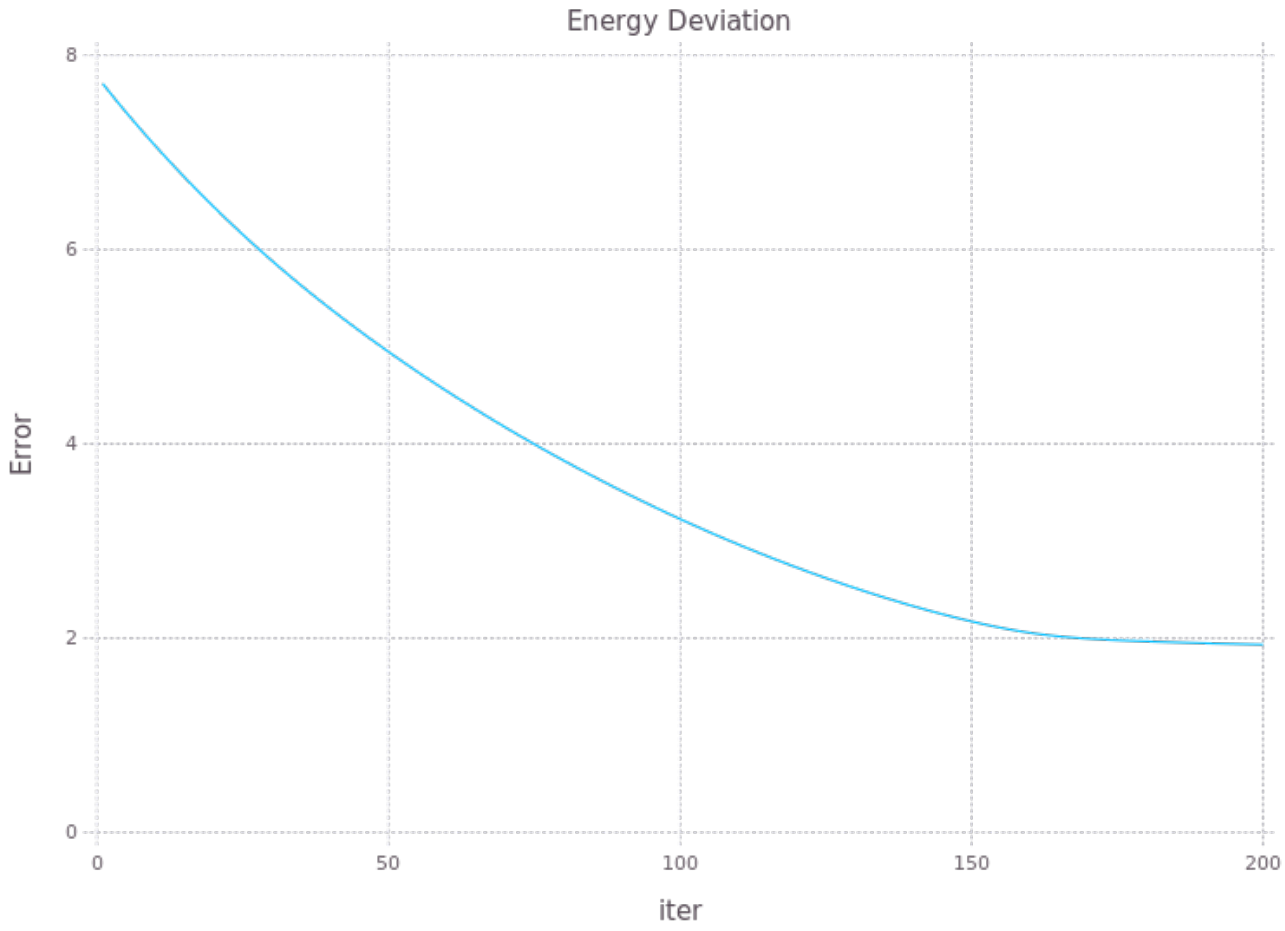


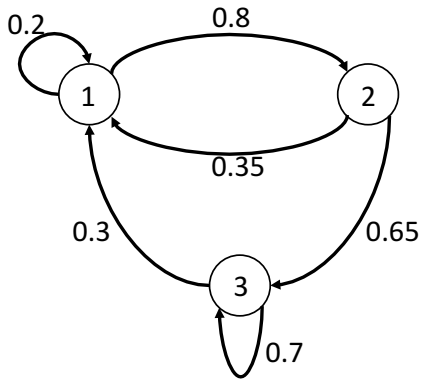
Figure 3: cost

3. *Oh The Places You Could Go!*

(Note: This problem deals with discrete-time Markov chains. But, you are not expected to have any prior knowledge of Markov chains. Everything you need to solve the problem is given in the problem statement.)

A Markov chain is a process with a set of states and probabilities of moving from one state to another. A Markov chain can be represented as a directed graph where edges have probability values assigned to them. Nodes in a Markov chain are called *states*. Remember that the edges are oriented in a directed graph.

We can specify a Markov chain of n states using its transition matrix $P \in \mathbb{R}^{n \times n}$, where P_{ij} is the probability that given the current state is j, the next state will be i. Note that if there is no edge from state j to state i, then we have $P_{ij} = 0$. See the simple example below for an illustration. Consider the following state diagram:



The transition matrix P of this Markov chain is

$$P = \begin{bmatrix} 0.2 & 0.35 & 0.3 \\ 0.8 & 0 & 0 \\ 0 & 0.65 & 0.7 \end{bmatrix}.$$

In this example, consider the transition from state 2 to state 3. In the state diagram, the transition from state 2 to state 3 has a probability of 0.65; thus, $P_{32} = 0.65$. Now, consider the edge from state 1 to again state 1. This means that given the current state is 1, with probability 0.2, the state will not change. Hence, $P_{11} = 0.2$. Lastly, note that there is no edge from state 1 to state 3. This means that if the current state is 1, the next state cannot be 3. Thus, $P_{31} = 0$.

Just as in graphs, we have *paths* in Markov chains too. A path of length l from state j to state i is a sequence of $l + 1$ states $s_0 = j, s_1, ..., s_l = i$, with $P_{s_{k+1}, s_k} \neq 0$ for $k = 0, 1, ..., l - 1$. Furthermore, the probability of a path is calculated by multiplying all the edge probabilities along the path:

$$\Pr(\text{path}) = \prod_{k=0}^{k=l-1} P_{s_{k+1}, s_k}.$$

For example, in the given state diagram, consider the sequence $3, 1, 1, 2, 1$. This is a path of length 4 starting from state 3 and ending at state 1. The probability of this particular path with states $3, 1, 1, 2, 1$ is calculated as: $P_{13} \times P_{11} \times P_{21} \times P_{12} = 0.3 \times 0.2 \times 0.8 \times 0.35 = 0.0168$.

Now, here is the problem. Consider a Markov chain of n states with transition matrix $P \in \mathbb{R}^{n \times n}$. Answer the following questions.

(a) Let $B = P^k$, where $k \in \mathbb{Z}, k \geq 1$. Give a simple interpretation of B_{ij} in terms of the original Markov chain. Explain why this interpretation is true. (*Hint: You might want to use the definition of a path in a Markov chain given above.*)

The remaining parts concern the specific Markov chain given in the file '*markov_chain.**'. For the following parts, please first briefly explain how to solve the question and then give your answer (enclosed in a box) and code.

(b) What is the length of the shortest possible path from state 7 to state 3?

(c) What is the sum of the probabilities of all such paths (7 to 3) with length equal to what you found in part b?

(d) Consider all possible paths of length 9 or less from state 2 to state 3. What is the weighted average of the lengths of all these paths with weights being the path probabilities; that is, find

$$\sum_{i=1}^{|S|} \Pr(\text{path}_i) l_{(\text{path}_i)},$$

where S is the set of all possible paths of length 9 or less from state 2 to state 3, and $l_{(\text{path}_i)}$ is the length of path_i? (The weighted average expression given above would be called *expected path length* for paths of length 9 or less, if we had scaled the probabilities such that they sum to 1.)

(e) Note that in the given transition matrix P, we have $P_{i5} = 0$ for all $i \in \{1, ..., n\} \backslash \{5\}$ (this is the set of $1..n$ with 5 removed) and $P_{55} = 1$. This means that state 5 is an *absorbing* state, meaning that once we reach this state, there is no escape because $P_{55} = 1$. Let a_j denote the probability of eventually reaching state 5 when the starting state is j. We have the following expression for a_j's:

$$a_j = \sum_{i=1}^{n} P_{ij} a_i.$$

For this part of the problem, we will compute a_j's for all $j = 1, ..., n$. Find a_j's using the given expression. Comment on the a_j values you found, using at most two sentences. (*Hint: It might be useful to form a vector from a_j's, excluding a_5 since it is already known: $a_5 = 1$.*)

Solution.

(a) **4 points** B_{ij} is the probability that we'll reach state i from state j in k steps. Or, it is the sum of probabilities of all paths of length k from state j to state i.

(b) **4 points** By checking when B_{37} becomes nonzero at each iteration over k where $B = P^k$, we can find the shortest path length. It's the smallest k for which B_{37} is nonzero. The answer is 3. The following code solves this problem:

```
%% part b
k = 1;
B = P^k;
while B(3,7) == 0
    k = k+1;
    B = P^k;
end
shortest_path_length = k
```

(c) **2 points** Let us denote the result from part b by l_{short}. Then, the sum of the probabilities of particular paths of length 3 is given by B_{37}, where $B = P^{l_{short}}$. The answer is 0.1674. The following code solves this:

```
%% part c
shortest_path_prob = B(3,7)
```

(d) **5 points** Note that the probability of a path of certain length is given by the corresponding index of the transition matrix raised to a power equal to the path length. Thus, we need all probabilities for all paths of lengths $1, .., 9$. Then, we'll multiply these probabilities with their corresponding lengths and sum them all up. The answer is 4.1302. The code is as follows:

```
%% part d
summ = 0;
for k = 1:9
    B = P^k;
    summ = summ + B(3,2)*k;
end
summ
```

(e) **5 points** We can express the given equation in this form: $a = P^T a$, where a is a column vector filled with a_j's. Because we know a_5, let us remove it from a and call it $\tilde{a}$. Then, we have $\tilde{a} = \tilde{P}\tilde{a} + b$, where $\tilde{P}$ is P with 5th row and column removed, and b is the 5th column of P^T with 5th entry removed. Moving $\tilde{P}\tilde{a}$ to the left-hand side of the equation, we get:

$$(I - \tilde{P})\tilde{a} = b.$$

Solving this linear system, we get the result: $\tilde{a} = [1, 1, 1, 1, 1, 1]^T$. Adding a_5, we have $a = [1, 1, 1, 1, 1, 1, 1]^T$. All a_j's are found to be 1. This makes sense because

there is only one absorbing state in the given Markov chain and in the end it is inevitable that whichever state we start, we'll end up in the absorbing state. It is important to notice that this is possible because there is only one absorbing state. If there was 2 or more, then we wouldn't get all a_j's equal to 1. The code for this part is as follows:

```
%% part e
n = size(P,1);
P_tilde = P;
P_tilde(5,:) = [];
P_tilde(:,5) = [];
a_tilde = inv(eye(n-1) - P_tilde') * P(5, [1:4,6:end])'
```

Note: It is possible to solve this problem in a couple of different ways. One way is to notice that a in the equation $a = P^T a$ is the eigenvector corresponding to the eigenvalue 1. Then you need to scale the vector a such that $a_5 = 1$. Another solution is to take a high power of P, say 100 and look at the 5'th row, which is all 1's. To understand this approach, note this: The 5'th row gives the probability of being in state 5 after 100 transitions/steps where the starting state is the column number. So, the 5'th row will give the vector a.

To get full credit from this problem, you could've used any of the 3 approaches above (or some other solution) as long as you support your solution with some math. The hint was to point you to the first solution above.

4. *Walk The Line*

Consider a toy vehicle on a 1-dimensional path. Let v_i denote the velocity of the vehicle in the time interval $t \in [i, i+1)$. The vehicle spends a kinetic energy of $E_i = bv_i^2$ during $t \in [i, i+1)$. $b \in \mathbf{R}$ here is a given constant. Furthermore, extra energy is lost due to extreme wind. Let the energy lost in $[i, i+1)$ be denoted by L_i. We model the energy loss as $L_i = k_i v_i$, where k_i is the loss coefficient in $t \in [i, i+1)$. Our purpose is to determine v_i, $i = 1, ..., n$ that minimize the total kinetic energy use. But, we are also given some constraints that v_i's need to satisfy:

- The first constraint is that the mean of all velocities must be equal to the given v_{mean}. That is,
$$\frac{1}{n}\sum_{i=1}^{n} v_i = v_{mean}$$
must be satisfied.
- The second set of constraints is the velocities v_j must satisfy $v_j = v_j^{given}$ for some subset of indices, that is for some velocities there is a strict constraint.
- The final constraint is that the sum of the total energy loss must be equal to the given L_{total}:
$$\sum_{i=1}^{n} L_i = L_{total}.$$
(Note: It would make more sense to model the last constraint as an inequality instead of an equality. However, since we want to restrict the problem to what we have seen in this class, we made this constraint an equality to make things simpler. You can think about this as a worse case)

The goal of the problem is to find the velocities that minimize the total kinetic energy usage E, where
$$E = \sum_{i=1}^{n} E_i,$$
subject to the given constraints.

(a) Set this problem up as an optimization problem with a quadratic cost and linear constraints. Then, find the solution of the problem; that is, find the expression for the velocities that minimize the cost, subject to the constraints.

(b) Solve the problem for the data given in '*minimizing_kinetic_energy.**'. Report the minimum total kinetic energy and the velocities. Plot the velocity as a function of time. In another figure, also plot the given loss coefficients k_i. Comment briefly (1-2 sentences) on the relationship between the velocities and k_i's. Submit the plots.

(c) We now modify our optimization problem by adding another term to the objective, in order to smooth the velocity and reduce acceleration. Let J be the new objective:

$$J = E + \mu \sum_{i=2}^{n} (v_i - v_{i-1})^2.$$

Explain how to minimize J subject to the same given constraints. Then, solve the problem for the given data. Take $\mu = 10000$. Report the minimum objective and the velocities. Plot the velocity as a function of time. Submit the plot and code.

(d) Comment briefly on the effect of the additional term in the objective J on the velocities. (no need for a long explanation, 1 or 2 sentences would be sufficient)

Solution.

(a) **6 points** Let us define the vector $v = \begin{bmatrix} v_1 & ... & v_n \end{bmatrix}^T$. We can rewrite the total kinetic energy as: $E = b\|v\|_2^2$. Now, we'll write all of the constraints in terms of v. First constraint can be rewritten as:

$$\frac{1}{n} \begin{bmatrix} 1 & ... & 1 \end{bmatrix} v = v_{mean}.$$

Second set of constraints:

$$e_j^T v = v_j^{given}$$

for $j \in J$, where e_j is the standard unit vector; that is, e_j is an n-dimensional vector full of zeros except for the j'th entry which is a 1.

Third constraint:

$$\begin{bmatrix} k_1 & ... & k_n \end{bmatrix} v = L_{total}.$$

We can express all of these constraints in a single matrix equation: $Av = y$, where

$$A = \begin{bmatrix} \frac{1}{n} 1^T \\ k^T \\ e_{j_1} \\ \vdots \\ e_{j_m} \end{bmatrix}.$$

Note that in the above expression, we take $J = \{j_1, ..., j_m\}$. Furthermore, y is defined as $y =$
$\begin{bmatrix} v_{mean} & L_{total} & v_{j_1}^{given} & ... & v_{j_m}^{given} \end{bmatrix}^T$. We now have the following constrained optimization problem:

$$\begin{aligned} \underset{v}{\text{minimize}} \quad & b\|v\|_2^2 \\ \text{subject to} \quad & Av = y. \end{aligned}$$

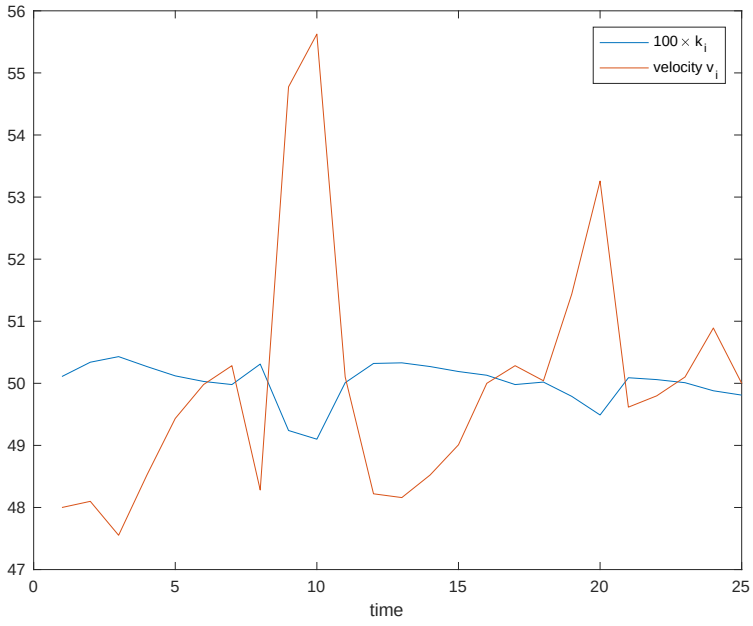
This is a least-norm problem. The solution is $v = A^T(AA^T)^{-1}y$.

Note: If you can convert a problem into an optimization problem you know the solution of (such as a least-norm problem as above), then you can simply apply the solution. You don't need to find the Lagrangian and optimality conditions. The solution $v = A^T(AA^T)^{-1}y$ was already obtained using Lagrangian. There is no need to repeat the steps. You can simply use the solution $v = A^T(AA^T)^{-1}y$, *if* you converted your problem into a least-norm problem. But, of course, you're free to use the Lagrangian approach (which is not really necessary for this part and requires more work) as long as you use it correctly.

Note 2: In an optimization problem with constraints, we always are looking for solutions that satisfy the constraints. If a solution doesn't satisfy the constraints, then we don't really have a solution. It might happen that there is no solution for the constraints. Then, the optimization problem has no solution. But, in this class, we select the constraints such that there always exist points that satisfy the constraints and we want to pick our solution from that set of points such that it minimizes the objective.

(b) **4 points** Minimum E is 62597. The velocities are: [48.0000 48.0990 47.5527 48.5238 49.4343 49.9806 50.2840 48.2810 54.7756 55.6253 50.1019 48.2203 48.1596 48.5238 49.0094 50.0000 50.2840 50.0412 51.4373 53.2582 49.6164 49.7985 50.1019 50.8910 50.0000].

The plot is as follows:



The relationship between k_i's and the velocities is that at times when k_i is small, the velocity is high.

The following Matlab code solves this problem:

```
%% part b
% A matrix
A = 1/n * ones(1, n);
A = [A; k'];
for j = 1:length(J)
```

```
    tmp = zeros(1, n);
    tmp(J(j)) = 1;
    A = [A; tmp];
end
% y vector
y = [v_mean; L_total; v_given'];

% solving for v
v = A'*inv(A*A')*y
total_kinetic_energy = b*norm(v,2)^2
% plot
figure; plot(1:n, 100*k);
hold on;
plot(1:n, v);
xlabel('time');
legend('100\times k_i', 'velocity v_i');
```

(c) **8 points** Let us define the following matrix:

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

Now, we can rewrite the objective as follows:

$$J = b\|v\|^2 + \mu\|Bv\|^2.$$

Moving the constants inside the norms, we get:

$$J = \|\sqrt{b}v\|^2 + \|\sqrt{\mu}Bv\|^2.$$

We can now write this as a single norm: $J = \|Dv\|^2$, where

$$D = \begin{bmatrix} \sqrt{b}I_n \\ \sqrt{\mu}B \end{bmatrix}$$

We have the following optimization problem:

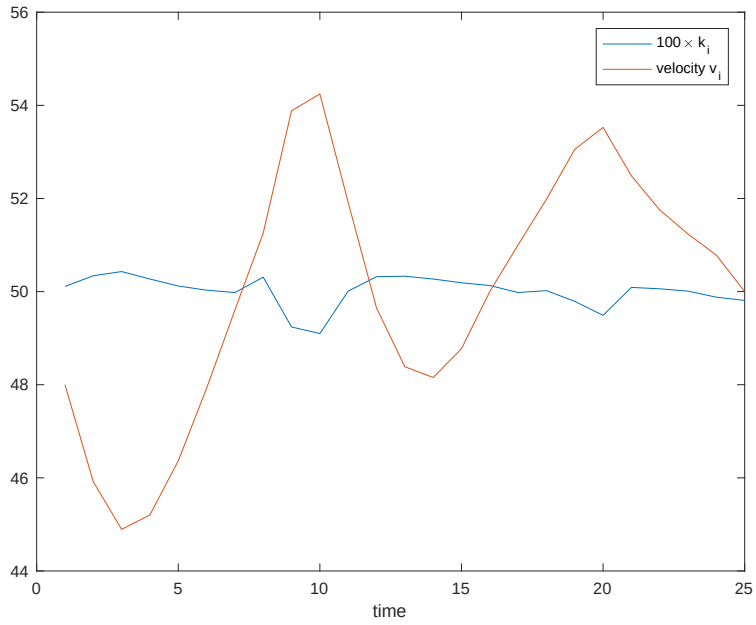
$$\begin{aligned} \underset{v}{\text{minimize}} \quad & \|Dv\|_2^2 \\ \text{subject to} \quad & Av = y. \end{aligned}$$

We can solve this problem using the last expression given in page 15 of lecture slides 17:

$$v = (D^T D)^{-1} A^T (A(D^T D)^{-1} A^T)^{-1} y.$$

The minimum J is 482431. The velocities are: [48.0000 45.9157 44.8931 45.2043 46.3654 47.9227 49.6039 51.2581 53.8837 54.2438 51.9150 49.6501 48.3867 48.1550 48.7734 50.0000 51.0049 51.9828 53.0549 53.5255 52.4869 51.7542 51.2368 50.7831 50.0000].

The plot is given below:



The following Matlab code solves this problem:

%% *part c*

```
mu = 10000;
B = zeros(n-1, n);
for i = 1:(n-1)
    B(i,i:i+1) = [1 -1];
end
D = [sqrt(b)*eye(n); sqrt(mu)*B];
v = inv(D'*D)*A'*inv(A*inv(D'*D)*A')*y
objective_part_c = norm(D*v, 2)^2
% plot
figure; plot(1:n, 100*k);
hold on;
plot(1:n, v);
xlabel('time');
legend('100\times k_i', 'velocity v_i');
```

(d) **2 points** The additional term on the objective makes the velocities at each time interval close to each other. In other words, it makes the velocity transition smoother. This can either be noticed from the plots or from the expression, which is the sum of squares of differences of velocities.

5. *But First, Let Me Take a Selfie*

An image with n-pixel by n-pixels can be directly represented as a matrix $X \in \mathbb{R}^{n\times n}$. Most image processing problems are usually converted and solved as matrix problems. In this question, we will look deeper into an interesting topic about image reconstruction and deblurring using different filters (no, not instagram).

Given a two-dimensional image we can apply a image transformation process. In this case, the *Sliding-Window Spatial Filter* method, is equivalent to 2D discrete convolution. More concretely, given a 2D filter kernel (matrix) $W \in \mathbb{R}^{(n-m+1)\times(n-m+1)}$, we define the 2D convolution as the weighted summation of all image pixels in the sliding window according to the weight kernel's size W, *i.e.*

$$(f_W(X))_{ij} = \sum_{k=1}^{n-m+1}\sum_{l=1}^{n-m+1} W_{kl}X_{i+k-1,j+l-1}$$

where output image $Y \approx f_W(X) \in \mathbb{R}^{m\times m}$.

Additionally, like what we saw in the homework, it can be convenient to describe a matrix by a vector $x = \mathbf{vec}(X) \in \mathbb{R}^{n^2}$. For example, given an array $X \in \mathbb{R}^{n\times n}$, X can be conveniently vectorized by stacking all columns:

$$\mathbf{vec}(X) = \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

where $x_i \in \mathbb{R}^n$ is the column vector of X, *i.e.* $X = \begin{bmatrix} x_1, x_2 \dots x_n \end{bmatrix}$. Similarly, we can also represent output image Y by vector $y = \mathbf{vec}(Y) \in \mathbb{R}^{m^2}$.

Therefore the 2D convolution transformation is equivalent as following multiplication by a matrix $D \in \mathbb{R}^{(m^2)\times(n^2)}$.

$$\mathbf{vec}(Y) = D\mathbf{vec}(X)$$

Finally we get to the problem.

(a) **Sharpening Spatial Filter and Image Reconstruction**

One kind of simple filters is the *Sharpening Spatial Filter*, which calculates the horizontal, vertical and diagonal pixel difference for each position in image. For example, such a kernel W can look like:

$$W = \begin{bmatrix} 1 & 2 & 1 \\ 2 & -12 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Given the above kernel as a spatial filter, we can write the 2D convolution transformation process as $\mathbf{vec}(\mathrm{Y}) = \mathrm{D}\ \mathbf{vec}(\mathrm{X})$ as described above, where D is found as mentioned in the vectorization process.

We define the *roughness* of image array U as

$$R = \|D\mathbf{vec}(U)\|^2$$

This metric, R, is the sum of the squares of the 2D spatial differences for each elements in the image array U with its neighbors. A small R value corresponds to smoothly varying image U.

Now we come to the image reconstruction problem. Suppose some pixels are randomly missing in original image, and given the incomplete image, our goal is to interpolate these unknown pixels with the prior assumption that neighboring pixels usually have similar values.

To define the problem more precisely, we partition the set of indices $1, \ldots, n^2$ into two sets: I_{known} and $I_{unknown}$. Assume there are k known pixels in the image, *i.e.* k values in a vector $v_{known} \in \mathbb{R}^k$, and $n^2 - k$ unknown pixels in a vector $v_{unknown} \in \mathbb{R}^{n^2-k}$. Then we can describe the incomplete image as the linear combination of two vectors using two matrices $Z_{known} \in \mathbb{R}^{n^2 \times k}$ and $Z_{unknown} \in \mathbb{R}^{n^2 \times (n^2-k)}$, *i.e.*

$$\mathbf{vec}(X) = Z_{known} v_{known} + Z_{unknown} v_{unknown}$$

where each columns of Z_{known} and $Z_{unknown}$ is a unit vector mapping the entries in v vector to corresponding position in vector $\mathbf{vec}(X)$. In fact, the matrix $\begin{bmatrix} Z_{known} & Z_{unknown} \end{bmatrix}$ is a permutation matrix.

In the data file **ImageReconstructionData.***, you will find an incomplete image array `incomplete_img` and also problem data `v_known` and `idx_known` and `idx_unknown`. `v_known` is a column vector for all known pixel values, that is, the vector v_{known} as described above. `idx_known` is a column vector which maps the corresponding known pixel value in the vector v_{known} to the correct position in the vector of incomplete image $\mathbf{vec}$(`incomplete_img`). In other words, the *ith* known value in the vector `v_known` is actually located at the position `idx_known`(i) in the incomplete image vector $\mathbf{vec}$(`incomplete_img`) *i.e.* let's denote vector $z = \mathbf{vec}$(`incomplete_img`)

$$\texttt{v\_known}(i) = z(\texttt{idx\_known}(i))$$

Similarly, `idx_unknown` is a column vector which maps the corresponding unknown pixel value in the vector $v_{unknown}$ to the correct position in the vector of incomplete image $\mathbf{vec}$(`incomplete_img`); which also means that the value at the position `idx_unknown`(i) in the incomplete image vector is unknown.

Please note that you need to firstly construct matrix Z_{known} and $Z_{unknown}$ from vector `idx_known` and `idx_unknown` as described above. Then try to find the

unknown vector $v_{unknown}$ to reconstruct the original complete image by minimizing the image roughness measures R. Please give the value of roughness measure R for the reconstructed image.

(b) In the data file **ImageReconstructionData.m**, there is also the original image array `original_img`. Compare the reconstructed image and the original image. Hand in the complete source code and the images plots.

For matlab users you might fine following hints are useful.

You can use `"imagesc()"` function to show the figure.

In order to display grey image, add command `"colormap gray"` before `"imagesc()"`

If the image array data is `"double"`, you might want to convert to `"uint8"` to show right figure

For python users you can use the Images module in the PIL toolbox i.e.

```
from PIL import Image
import numpy as np
imgtest=img.fromarray(YourImgMatrix*255)
```

where *YourMatrix* is a matrix of pixel values (note the 255 to scale the pixel value from 0-1 to 0-255 (8bit))

For Julia users you need to install the Images package

```
Pkg.add("Images")
```

And image viewing is just

```
img=Gray.(YourImgMatrix)
```

For more info see

`http://juliaimages.github.io/latest/arrays_colors.html`

(c) **Smoothing Spatial Filter and Image Deblurring**

Another simple filter W is the *Smoothing Spatial Filter*, which calculate the weighted average of pixel values in the sliding window. For example the smoothing kernel is:

$$W = \frac{1}{16} \times \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

Note: you are not going to use above filter example to do more processing, it is just an example. In the following question, you will be required to work out a new spatial kernel W given a series of images.

Suppose we are taking pictures using a camera but carelessly get a bunch of blurry images Y. In order to deblur these images, we need the blurring model of the camera. To simplify, we assume the blurring process can be approximately described by 2D convolution and a constant offset, *i.e.*

$$\phi_{W,b}(X^{(k)})_{ij} = f_W(X^{(k)})_{ij} + b$$

where f_W is the 2D convolution transformation described above, and $\phi_{W,b}(X)$ is the predicted blurry image. $b \in \mathbb{R}$ is the constant offset value.

Given the original image series $X^{(1)}, \cdots, X^{(N)} \in \mathbb{R}^{n \times n}$, and corresponding blurry images $Y^{(1)}, \cdots, Y^{(N)} \in \mathbb{R}^{m \times m}$, you need to estimate the model parameters W and b by minimizing the RMS fitting error.

$$J = \sqrt{\frac{1}{N} \sum_{k=1}^{N} \|Y^{(k)} - \phi_{W,b}(X^{(k)})\|_F^2}$$

where the squared Frobenius norm of a matrix $M \in \mathbb{R}^{m \times m}$ is given by

$$\|M\|_F^2 = \sum_{i=1}^{m} \sum_{j=1}^{m} M_{ij}^2$$

Explain how to choose the blurring model parameters $W \in \mathbb{R}^{(n-m+1)\times(n-m+1)}$ and $b \in \mathbb{R}$ in order to minimize the RMS fitting error above.

(d) Apply your method to the data given in **ImageDeblurData.m**. Now each input image is a matrix $X \in \mathbb{R}^{28 \times 28}$, each camera image is a matrix $Y \in \mathbb{R}^{24 \times 24}$, and the filter you wish to estimate has weight matrix $A \in \mathbb{R}^{5 \times 5}$ and scalar bias $b \in \mathbb{R}$. Use `X` and `Y` in the **ImageDeblurData.m** file.

In each of the data matrices, each column is a different image. The images have been reshaped from matrices to column vectors by reshaping the matrices in column-major order. That is, we represent every input image $X^{(i)}$ by vector $x^{(i)} = \mathbf{vec}(X^{(i)}) \in \mathbb{R}^{n^2}$, which is the ith columns in matrix `X`. Similarly, we represent every output image $Y^{(i)}$ by vector $y^{(i)} = \mathbf{vec}(Y^{(i)}) \in \mathbb{R}^{m^2}$, which is the ith columns in matrix `Y`.

Thus, `X` is a matrix in $\mathbb{R}^{784 \times 100}$, so there are 100 input images of 784 pixels each, and `Y` is a matrix in $\mathbb{R}^{576 \times 100}$ with the corresponding 100 camera images of 576 pixels each. Report to **four decimal** places your estimates of the parameters W and b, and the corresponding value of J.

(e) Given a image $\hat{Y}$ which is blurred under the same camera setting, can you always find the deblurred image $\hat{X}$ such that $\phi_{W,b}(\hat{X}) = \hat{Y}$ by using the above model? Justify your answer. Please feel free to use Matlab/Phython to compute or check your condition. Please also submit your code.

Solution.

(a) **4 points**

Firstly we can express our roughness measure directly in terms of the vector of known values vector v_{known} and unknown values vector $v_{unknown}$ as

$$\begin{aligned} R &= \|D(Z_{known} v_{known} + Z_{unknown} v_{unknown})\|^2 \\ &= \|DZ_{known} v_{known} + DZ_{unknown} v_{unknown}\|^2 \end{aligned}$$

Defining

$$A = DZ_{unknown} \quad b = -DZ_{known}v_{known}$$

Provided that A is skinny and full rank, the least-square solution is

$$v_{unknown} = (A^T A)^{-1} A^T b$$

In order to construct matrix D, we can use the kernel W to shift across each rows and columns on the image array X as a sliding window, and then vertorize the image array X and corresponding coefficient matrix with nonzero weights value only located within the sliding window.

Repeat the above process and stack all coefficient vector to get matrix D. Please refer to following code to see how to implement this.

(b) **4 points**

The following code solves this problem

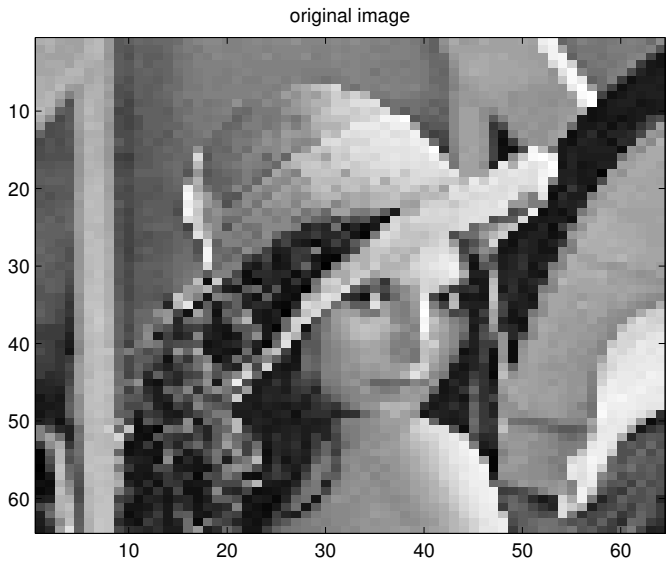
```
clear all; close all; clc
%% load data from data file
ImageReconstructionData;
%% construct Z_known and Z_unknown
[m, n] = size(incomplete_img); % compute the number of rows and columns for
    image X
num = m*n; % compute the total number of pixels in image X
num_known = length(idx_known);
num_unknown = length(idx_unknown);

Z_known = zeros(num, num_known);
Z_unknown = zeros(num, num_unknown);

for i = 1:num_known
    Z_known(idx_known(i), i) = 1;
end

for j = 1:num_unknown
    Z_unknown(idx_unknown(j), j) = 1;
end

%% construct matrix D
W = [1, 2, 1;
    2, -12, 2;
     1, 2, 1];
D = zeros((m-2)*(n-2), m*n);
ctr = 1;
for j = 1:(n-2)
    for i = 1:(m-2)
        tmp = zeros(m, n);
        tmp(i:(i+2), j:(j+2)) = W;
        tmp = tmp(:);
```

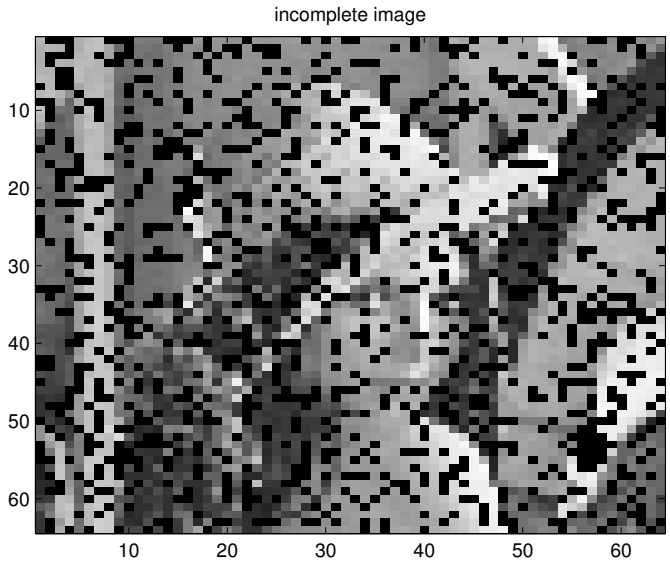
```
        D(ctr, :) = tmp';
        assert(ctr == i+(j-1)*(m-2));
        ctr = ctr + 1;
    end
end
%% solve least square problem for unknown values
A = D * Z_unknown;
assert(size(A,2) == rank(A)); % check if A is skinny and full-rank
b = - D * Z_known * v_known;
v_unknown = pinv(A) * b;
%% display original image and incompletement image
figure(1);
colormap gray;
imagesc(uint8(original_img));
imwrite(uint8(original_img), 'original_img.png');
title('original image');

incomplete_img_vector = Z_known * v_known + Z_unknown * zeros(num_unknown,1);
incomplete_img = reshape(incomplete_img_vector, [m, n]);
figure(2);
colormap gray
imagesc(uint8(incomplete_img));
imwrite(uint8(incomplete_img), 'incomplete_img.png');
title('incomplete image');
%% display reconstructed image
reconstructed_img_vector = Z_known * v_known + Z_unknown * v_unknown;
reconstructed_img = reshape(reconstructed_img_vector, [m, n]);
figure;
colormap gray
imagesc(uint8(reconstructed_img));
imwrite(uint8(reconstructed_img), 'reconstructed_img.png');
title('reconstructed image');
%% compute roughness of reconstructed image
R = norm(D * reconstructed_img_vector)^2
```

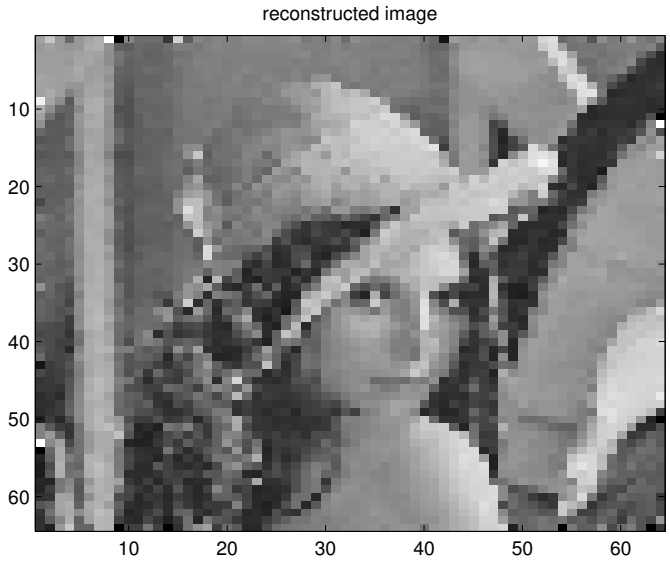
As a result, the roughness measure of reconstructed image is

```
R =  2.0041e+8
```

Display the original image and reconstructed image here.


original image
10
20
30
40
50
60
10
20
30
40
50
60


incomplete image
10
20
30
40
50
60
10
20
30
40
50
60



(c) **5 points**

Although in our problem data file, $n = 28$ and $m = 24$, here we consider $n = 5$ and $m = 3$ for illustrative purposes. The results of this section can be generalized to any value of n and m.

One straightforward way to solve this problem is to regard weight kernel W and constant scalar b as unknown vector. Then work with original image series $X^{(1)}, \cdots, X^{(N)} \in \mathbb{R}^{n \times n}$ given in the problem statement to figure out coefficient matrix A. In the other hand, reshape the blurry images $Y^{(1)}, \cdots, Y^{(N)} \in \mathbb{R}^{m \times m}$ into one vector shown in the least square problem.

We can rewrite the expression for Y_{ij} given in the problem statement as

$$Y_{ij} = \begin{bmatrix} X_{ij} & X_{i+1,j} & X_{i+2,j} & X_{i,j+1} & \cdots & X_{i+2,j+2} & 1 \end{bmatrix} \begin{bmatrix} W_{11} \\ W_{21} \\ W_{31} \\ W_{21} \\ \vdots \\ W_{33} \\ b \end{bmatrix}$$

We can concatenate the pixels of all of our camera images as a column vector.

$$\begin{bmatrix} Y_{11}^{(1)} \\ Y_{21}^{(1)} \\ \vdots \\ Y_{33}^{(1)} \\ Y_{11}^{(2)} \\ Y_{21}^{(2)} \\ \vdots \end{bmatrix} = \begin{bmatrix} X_{11}^{(1)} & X_{21}^{(1)} & X_{31}^{(1)} & X_{12}^{(1)} & \dots & X_{33}^{(1)} & 1 \\ X_{21}^{(1)} & X_{31}^{(1)} & X_{41}^{(1)} & X_{22}^{(1)} & \dots & X_{43}^{(1)} & 1 \\ \vdots & & & & \ddots & & \vdots \\ X_{33}^{(1)} & X_{43}^{(1)} & X_{53}^{(1)} & X_{34}^{(1)} & \dots & X_{55}^{(1)} & 1 \\ X_{11}^{(2)} & X_{21}^{(2)} & X_{31}^{(2)} & X_{12}^{(2)} & \dots & X_{33}^{(2)} & 1 \\ X_{21}^{(2)} & X_{31}^{(2)} & X_{41}^{(2)} & X_{22}^{(2)} & \dots & X_{43}^{(2)} & 1 \\ \vdots & & & & \ddots & & \vdots \end{bmatrix} \begin{bmatrix} W_{11} \\ W_{12} \\ W_{13} \\ W_{21} \\ \vdots \\ W_{33} \\ b \end{bmatrix}$$

We now use least squares to find W and b.

Note that we have effectively taken each input image $X^{(k)}$ and generated 9 images in $\mathbb{R}^{3\times3}$. Likewise, we have split each output camera image $Y^{(k)}$ into 9 images in $\mathbb{R}$ (i.e., each new camera image has only one pixel). This is equivalent to the original problem from the filter's point of view.

(d) **5 points**

The estimated filter parameters are

$$W = \begin{bmatrix} 0 & 0.0165 & 0.0205 & 0.0154 & 0.0026 \\ 0.0138 & 0.0572 & 0.0974 & 0.0575 & 0.0127 \\ 0.0231 & 0.1004 & 0.1598 & 0.1021 & 0.0217 \\ 0.0126 & 0.0591 & 0.1030 & 0.0547 & 0.0157 \\ 0.0035 & 0.0120 & 0.0212 & 0.0149 & 0.0015 \end{bmatrix}$$

$$b = 0.5001$$

The RMS error J is `1.2019`.

The following code solves this problem

```
clear all; close all; clc
%% load data from data file
ImageDeblurData;

%% construct Y vector
Y_vec = Y(:);
assert(length(Y_vec) == m^2 * N);
assert(n-m+1 == 5)

%% construct coefficient matrix
C = zeros(m^2 * N, (n-m+1)^2);
ctr = 1;
for num = 1:N
    x_img = X(:, num);
    x_img = reshape(x_img, n, n);

    for j = 1:m
```

```
        for i = 1:m
            coef = x_img(i:i+n-m, j:j+n-m);
            C(ctr, :) = coef(:)';
            assert(ctr == i + (j-1)*m + (num-1)*m^2);
            ctr = ctr + 1;
        end
    end
end

%% solve least square problem for unknown values
A = [C, ones(m^2 * N, 1)];
assert(size(A,2) == rank(A)); % check if A is skinny and full-rank
W = pinv(A) * Y_vec;
%% display filter kernel W and constant b
b = W(end)
W = reshape(W(1:end-1), n-m+1, n-m+1)
%% compute J value
J = sqrt(norm( Y_vec - A*[W(:);b] )^2 / N)
```

(e) **2 points**

Recall from last question that

$$\hat{M}\hat{X} + b1 = \hat{Y}$$

We can rewrite this as

$$\hat{M}\hat{X} = \hat{Y} - b1$$

Note that since $\hat{M}$ is strictly fat, and a quick computation check shows that it is also **full-rank**. We know that $\hat{M}$ is **onto**. This means that for any $\hat{Y}$ there always exists some $\hat{X}$ that will satisfy this equation. But since the dimension of the nullspace of $\hat{M}$ is nonzero, the solution of above equation is not unique.

Therefore, given a image $\hat{Y}$ which is blurred under the same camera setting, we can always find some images $\hat{X}$ such that $\phi_{W,b}(\hat{X}) = \hat{Y}$ by using the above model. The following code construct matrix M and check full-rank condition

```
%% construct matrix M and check if M is full-rank
M = zeros(m^2, n^2);
ctr = 1;
for j = 1:(n-4)
    for i = 1:(n-4)
        tmp = zeros(n, n);
        tmp(i:(i+4), j:(j+4)) = W;
        tmp = tmp(:);
        M(ctr, :) = tmp';
        assert(ctr == i+(j-1)*(n-4));
        ctr = ctr + 1;
    end
end
assert(min(size(M)) == rank(M))
```