This homework is due February 15, 2016, at Noon.

1. Homework process and study group

Who else did you work with on this homework? List names and student ID’s. (In case of hw party, you can also just describe the group.) How did you work on this homework?

Solution: I worked on this homework with...

I first worked by myself for 2 hours, but got stuck on Problem 5 so I went to office hours on...

Then I went to homework party for a few hours, where I finished the homework.

2. MIMO wireless signals

Ever wonder why newer wifi routers and cellular base stations have 4 or sometimes even more antennas on them? New wireless technologies actually use multiple antennas that each send their own signal on the same frequency band. The key here is not only do we encode signals in frequency bands, but also in spatial ones using a technique known as 'Spatial Multiplexing'.

We call this idea 'MIMO' wireless, which stands for "multiple input multiple output". This technique is used in many standards including 802.11n/ac, 4G LTE, and WiMAX.

In this problem, we will explore how signals are decoded on the output end.

Consider the following:

![Diagram of MIMO system](image-url)

We have 2 transmit antennas and 3 receive antennas, each receive antenna gets some signal from each of the transmit antennas. We can model the input output relation of the system as follows:

$$\begin{bmatrix} h_{11} & h_{21} \\ h_{12} & h_{22} \\ h_{13} & h_{23} \end{bmatrix} \cdot \begin{bmatrix} x_1(t) \\ x_2(t) \end{bmatrix} = \begin{bmatrix} y_1(t) \\ y_2(t) \\ y_3(t) \end{bmatrix}$$
or

\[ H \tilde{x}(t) = \tilde{y}(t) \]

Here, \( H \) is the spatial-response matrix and it acts on the signals instantaneously at each time. For the purpose of this problem, we are going to pretend that there are no echoes across time.

(a) With our new MIMO wireless system, we want to recover the original \( \tilde{x}(t) \) signal after receiving the \( \tilde{y}(t) \). In order to do this, we will left multiply \( \tilde{y}(t) \) by some matrix \( A \); ideally we should then exactly recover \( \tilde{x}(t) \) \((A\tilde{y}(t) = \tilde{x}(t))\). Using the SVD to decompose \( H \), analytically write down what this matrix \( A \) should be.

**Solution:** Using the SVD we can break down the \( H \) matrix:

\[ H = U\Sigma V^T \]

Here, \( U \) is a 3x3 square orthonormal matrix, \( \Sigma \) is a 3x2 diagonal matrix and \( V \) is a 2x2 square orthonormal matrix.

Because \( U \) and \( V \) are orthonormal matrices, we know that the conjugate transpose (in the real case, just the transpose. Although everything in this problem generalizes to the complex case, because we taught you the SVD in the context of real matrices, here we just use the transpose.) of each is the inverse. Thus, we can reverse the initial transformation.

Due to the nature of the \( \Sigma \) matrix, we can also introduce a "pseudo inverse" for the \( \Sigma \) matrix, call it \( \tilde{\Sigma} \), such that \( \tilde{\Sigma}\Sigma = I \). Because \( \Sigma \) is not a square matrix, it does not have an actual inverse, so why should we believe that we can invert it at all? The reason is the size of \( \Sigma \) — it maps two dimensions into three dimensions and as long as the singular values aren’t zero, this mapping looks to be reversible. We call the matrix that inverts this the "pseudo inverse". We can see this by the following:

\[ \Sigma = \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} \]

\[ \tilde{\Sigma} = \begin{bmatrix} \frac{1}{\sigma_1} & 0 & 0 \\ 0 & \frac{1}{\sigma_2} & 0 \end{bmatrix} \]

\[ \begin{bmatrix} \frac{1}{\sigma_1} & 0 & 0 \\ 0 & \frac{1}{\sigma_2} & 0 \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 \\ 0 & \sigma_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix} \]

We start with the relation of the channel,

\[ H\tilde{x}(t) = \tilde{y}(t) \]

We are trying to find the matrix \( A \), such that

\[ \tilde{x}(t) = A\tilde{y}t \]

which recovers the original \( \tilde{x} \) vector given the \( \tilde{y} \) vector. Normally, we would simply multiply the matrix by its inverse, but because this is a non-square matrix, it doesn’t have a precise inverse. However, the SVD allows us to get a good solution to the problem. Because \( \tilde{x} \) is a length 2 vector, and \( \tilde{y} \) is a length 3 vector, \( A \) needs to be a 2x3 matrix.
From here, we decompose via the SVD:

\[ U \Sigma V^T \vec{x}(t) = \vec{y}(t) \]

Because \( U \) and \( V \) are orthonormal matrices, their transpose is also their inverse, \((U^T U = I)\), so to reduce, we multiply by the transpose.

\[ U^T U \Sigma V^T \vec{x}(t) = U^T \vec{y}(t) \]

\[ \Sigma V^T \vec{x}(t) = U^T \vec{y}(t) \]

To remove the \( \Sigma \), we multiply it by its pseudo inverse.

\[ \tilde{\Sigma} \Sigma V^T \vec{x}(t) = \tilde{\Sigma} U^T \vec{y}(t) \]

\[ V^T \vec{x}(t) = \tilde{\Sigma} U^T \vec{y}(t) \]

\[ \vec{x}(t) = V \tilde{\Sigma} U^T \vec{y}(t) \]

\[ A = V \tilde{\Sigma} U^T \]

So, following algebra once we set up the problem allowed us to find an appropriate \( A \).

(b) There might have been some noise on each receive antenna. So how is the solution you found in part (a) related to the least squares solution of this problem?

**Solution:**

The solution given by the SVD is precisely the same as the least squares solution to the problem. This can be seen by the following:

From above, we have: \( \vec{x}(t) = V \tilde{\Sigma} U^T \vec{y}(t) \)

The linear-least-squares solution is:

\[ \vec{x}(t) = (H^T H)^{-1} H^T \vec{y}(t) \]

We need to show that \((H^T H)^{-1} H^T = V \tilde{\Sigma} U^T \)

\[(H^T H)^{-1} H^T = (V \Sigma U^T U \Sigma V^T)^{-1}(U \Sigma V^T)^T \]= \((V \Sigma U^T)^{-1}(V \Sigma U^T)^T \)

\[= ((V^T)^{-1}(\Sigma^T \Sigma)^{-1} V^{-1})(V \Sigma U^T) \]

\[= (V(\Sigma^T \Sigma)^{-1} V^T)(V \Sigma U^T) \]

\[= V(\Sigma^T \Sigma)^{-1} V^T \Sigma U^T \]

\[= V(\Sigma^T \Sigma)^{-1} \Sigma U^T \]

So far, all we’ve been doing is expanding out terms and cancelling out inverses that are next to each other. At this point, we have to deal with this \((\Sigma^T \Sigma)^{-1}\) term.

\[
\Sigma^T \Sigma = \begin{bmatrix}
\sigma_1 & 0 & 0 \\
0 & \sigma_2 & 0 \\
0 & 0 & 0
\end{bmatrix}
= \begin{bmatrix}
\sigma_1 & 0 \\
0 & \sigma_2 \\
0 & 0
\end{bmatrix}
= \begin{bmatrix}
\sigma_1^2 & 0 \\
0 & \sigma_2^2
\end{bmatrix}
\]

The inverse of this matrix can be seen by inspection to be:

\[
(\Sigma^T \Sigma)^{-1} = \begin{bmatrix}
\frac{1}{\sigma_1^2} & 0 \\
0 & \frac{1}{\sigma_2^2}
\end{bmatrix}
\]
The product \((\Sigma^T \Sigma)^{-1} \Sigma^T\) can consequently be reduced

\[
(\Sigma^T \Sigma)^{-1} \Sigma^T = \begin{bmatrix} \frac{1}{\sigma_1^2} & 0 \\ 0 & \frac{1}{\sigma_2^2} \end{bmatrix} \begin{bmatrix} \sigma_1 & 0 & 0 \\ 0 & \sigma_2 & 0 \end{bmatrix} = \begin{bmatrix} \frac{1}{\sigma_1} & 0 \\ 0 & \frac{1}{\sigma_2} \end{bmatrix} = \tilde{\Sigma}
\]

Plugging back into our original equation, we finally get:

\[
V(\Sigma^T \Sigma)^{-1} \Sigma^T U^T = V\tilde{\Sigma}U^T \checkmark
\]

And we have shown what needed to be shown. That indeed the solution we chose is the same as the linear-least-squares solution from 16A.

(c) What we just did is referred to as "post processing" or "post coding", and involves the receive end having more antennas than the send side. Many times this is not the case (eg. a wireless cell tower having many more antennas than a phone). What if we wanted to send 2 streams on 3 antennas and receive precisely those 2 streams back on the other end?

The channel is very similar to the one we had made in part (a). In fact, the original channel modelled with spatial response matrix \(H\) is precisely the transpose of this channel! Thus, we can say the spatial response matrix for this channel, let’s call it \(H'\), is simply the following:

\[
H' = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \end{bmatrix} = H^T
\]

Using the SVD of \(H\) and its relation to \(H'\), show how you can pre-process \(x_1(t)\) and \(x_2(t)\) so that you recover them precisely after they have been transmitted across the channel. To be more explicit, after the processing and transmission has been done \(y_1(t) = x_1(t), y_2(t) = x_2(t)\).

**Solution:** If we want \(\tilde{x}(t)\) to equal \(\tilde{y}(t)\) precisely, then we want to have the matrix \(H\) multiplied by its "inverse" during pre-processing. The new matrix representing the channel is \(H'\) as we are going from 3 antennas to 2. We will represent the processing matrix by \(A\).

In other words, we want a matrix \(A\) such that \(\tilde{y}(t) = H'^TA\tilde{x}(t)\).

\[
H'^TA = I
\]

\[
(U\Sigma V^T)^TA = I
\]

\[
(V\Sigma^T U^T)A = I
\]
Similarly as before, $U$ and $V$ are orthonormal matrices, so their transpose is also their inverse. We can also define another right-inverse for the $\Sigma^T$ matrix as follows. The reason it is a right inverse is because $A$ right-multiplies $H^T$, therefore $\tilde{\Sigma}^T$ must be the right inverse of $\Sigma^T$.

\[
\Sigma^T = \begin{bmatrix}
\sigma_1 & 0 & 0 \\
0 & \sigma_2 & 0 \\
\end{bmatrix}
\tilde{\Sigma}^T = \begin{bmatrix}
\frac{1}{\sigma_1} & 0 & 0 \\
0 & \frac{1}{\sigma_2} & 0 \\
\end{bmatrix}
\]

\[
\begin{bmatrix}
\sigma_1 & 0 & 0 \\
0 & \sigma_2 & 0 \\
\end{bmatrix} \begin{bmatrix}
\frac{1}{\sigma_1} & 0 & 0 \\
0 & \frac{1}{\sigma_2} & 0 \\
\end{bmatrix} = \begin{bmatrix}
1 & 0 \\
0 & 1 \\
\end{bmatrix}
\]

And hence:

\[A = U\tilde{\Sigma}^TV^T\]

is the natural choice to have the two receive antennas pick up the two desired signals without any interference between the two.

(d) (Optional) Why do you think that we are using the SVD here? Is there a unique solution of what to transmit that will achieve the desired goal in the previous part? Why choose this approach?

**Solution:** This solution isn’t unique, but it is the solution with lowest signal energy of all those that work. In fact, we can choose any vectors in the null space of $H^T$, add them to the components in our $A$ matrix, and we will still get the original signal back. Let $\vec{z}$ be a vector in the null space of $H^T$ and let $\vec{w}$ be any arbitrary vector of length two that represents how to weigh the components of $\vec{x}$. Writing this out:

\[
H^T(\vec{A} + \vec{z}\vec{w}^T)\vec{x}(t) = y(t)
\]

\[
(H^T\vec{A} + H^T\vec{z}\vec{w}^T)\vec{x}(t) = y(t)
\]

\[
H^T\vec{A}\vec{x}(t) = y(t)
\]

What we have just shown is that if we add vectors in the nullspace of $H^T$ to our solution $A$, we find another solution that still works. So, the solution is definitely not unique. So, which solution has the lowest signal energy? How would we even have thought of looking at the signal energy at all? The reason is that the SVD-based approach seems to have an elegance to it and a spirit of minimalism. So, it is natural to search for an appropriate basis in which to look at the signals being transmitted — hopefully, that basis will help us see more clearly in what way our approach is elegantly minimal.

Consider the SVD representation $U\Sigma V^T$ for $H$. The last vector of $U$ is the one that is the basis of the nullspace of $H^T$ and by the property of the SVD, it is orthogonal to the other vectors in $U$.

Consequently, if we look at the signal put out on the transmit antennas, it is most convenient to look at them in the $U$ basis. Because it is an orthonormal basis, it preserves all inner-products and norms. In this basis, all the valid solutions have the same first-two components (because that is what it takes to reproduce $\vec{x}$ at the receiver $\vec{y}$) but the third component is perfectly free. Since the energy in the signal is the sum of the squares of each component, it is minimized when the third component is zero, which means it has nothing in the nullspace of $H^T$. No transmitted energy is wasted. This is precisely what the SVD-based solution for $A$ in the previous part is like. It always puts zero weight on the last column of $U$ and hence has no energy in the nullspace of $H^T$. 

EECS 16B, Spring 2016, Homework 3 5
3. Hermitian Matrices  A Hermitian matrix \( T \) is a square matrix, which is equal to its conjugate transpose, i.e. \( T^* = T \). Hermitian matrices have nice properties! In this problem, we will walk through one of them. Let \( \lambda \) be an eigenvalue of \( T \) with associated eigenvector \( \vec{v} \). That is to say,

\[ T \vec{v} = \lambda \vec{v} \quad (1) \]

(a) Multiply both sides of Equation (1) with \( \vec{v}^* \). Call this Equation (2). (\textit{Hint: Where should we put} \( \vec{v}^* \) \textit{at each side of the equation? The right or left side of the original formula?})

\textbf{Solution: } Here we put \( \vec{v}^* \) at the left side because that way, we get a scalar result. This is convenient because scalars are simpler so it is the first choice of what to try. We get

\[ \vec{v}^* T \vec{v} = \lambda \vec{v}^* \vec{v} \]
\[ = \lambda \| \vec{v} \|^2 \quad (2) \]

(b) Compute the conjugate transpose of each side of Equation (1). What do you get? Simplify and call this Equation (3). (\textit{Hint: use the fact that} \( T^* = T \))

\textbf{Solution:}

\[ (T \vec{v})^* = (\lambda \vec{v})^* \]
\[ \vec{v}^* T^* = \lambda \vec{v}^* \]
\[ \vec{v}^* T = \lambda \vec{v}^* \quad (3) \]

(c) Multiply both sides of Equation (3) with \( \vec{v} \). Let that be Equation (4).

\textbf{Solution:}

Again, we want to get a scalar if possible, so we multiply by \( \vec{v} \) on the right.

\[ \vec{v}^* T \vec{v} = \lambda \vec{v}^* \vec{v} \]
\[ = \lambda \| \vec{v} \|^2 \quad (4) \]

(d) Compare Equations (2) and (4). What can you conclude about the eigenvalues of Hermitian matrices?

\textbf{Solution: } Note that the left sides of Equation (2) and (4) are the same, so we have

\[ \lambda \| \vec{v} \|^2 = \overline{\lambda} \| \vec{v} \|^2, \]

which implies

\[ \overline{\lambda} = \lambda \]

Therefore, the eigenvalues of a Hermitian matrix are real.

4. Symmetric Matrices

A real symmetric matrix is an important special case of Hermitian matrices. Here we want to show every real symmetric matrix is diagonalized by a matrix of its orthonormal eigenvectors. In other words, a symmetric matrix has full complement of eigenvectors that are all orthogonal to each other.

In discussion section, you have seen a recursive derivation of this fact. Formally however, such recursive derivations are usually turned into proofs by using induction. This problem serves to both freshen your mind regarding induction as well as to give you a chance to prove for yourself this very important theorem.
(a) You will start by proving a basic lemma about real symmetric matrices under an orthonormal change of basis. Prove that if $S$ is a symmetric matrix $S = S^T$ that if $U$ is a matrix whose columns are orthonormal, then $S$ in the basis $U$, namely $U^T S U$ is also symmetric.

**Solution:** $U$ is a matrix with orthonormal column vectors. The transpose of $U^T S U$, $(U^T S U)^T$ is identical to $U^T S U$ as follows:

$$(U^T S U)^T = (SU)^T U = U^T S^T U = U^T S U$$

since $S^T = S$. Notice that it is not even necessary that $U$ is a square matrix. If $S$ is an $m \times m$ matrix, $U$ could be $m \times n$ while $n \neq m$.

(b) Another useful lemma is that real symmetric matrices have real eigenvalues. Argue why this is true.

In lecture, this was proved by leaning on the fact that real symmetric matrices can be diagonalized by an orthonormal basis. However, since that is what we are trying to prove here, it is good to have an alternative proof. *(Hint: Does another problem on the homework help us here?)*

**Solution:** Based on the previous problem about Hermitian matrices, we know the eigenvalues of a Hermitian matrix are real. A real symmetric matrix $S$ is also a Hermitian one, because under this condition (real and symmetric), $S^* = S^T$, while $S^* = S$ is the definition of Hermitian matrices. Consequently, $S$ must have all real eigenvalues.

(c) A third useful lemma is one about finding orthonormal bases. Show that given a nonzero vector $\vec{u}_0$ of length $n$, that it is possible to find an orthonormal set of $n$ vectors, $\vec{v}_0, \ldots, \vec{v}_{n-1}$ such that $\vec{v}_0 = \alpha \vec{u}_0$ for some scalar $\alpha$.

*(Hint: Use the Gram-Schmidt process somehow.)*

**Solution:** Finding $\vec{v}_0, \ldots, \vec{v}_{n-1}$ can be done with two steps: (1) based on $\vec{u}_0$, find $\vec{u}_1, \ldots, \vec{u}_{n-1}$, such that span($\vec{u}_0, \vec{u}_1, \ldots, \vec{u}_{n-1}$) = $\mathbb{R}^n$, and (2) apply the Gram-Schmidt process to orthonormalize that set of vectors.

For (1), one possible way is find a non-zero entry of $\vec{u}_0$, say $\vec{u}_0[k]$, then $\vec{u}_1, \ldots, \vec{u}_{n-1}$ could be the coordinate basis $\vec{e}_0, \ldots, \vec{e}_{n-1}$ but excluding $\vec{e}_k$. Why is this a basis? Because we can in this case, we can convert $\vec{u}_0$ into $\vec{e}_k$ (make other non-zero entries into zero) by subtracting appropriate multiples of the $\vec{e}_i$ basis vectors that we kept. We know something will be left because we chose $k$ so that $\vec{u}_0[k]$ is not zero. Because we have now recovered the standard basis, we know that span($\vec{u}_0, \vec{u}_1, \ldots, \vec{u}_{n-1}$) = span($\vec{e}_0, \vec{e}_1, \ldots, \vec{e}_{n-1}$) = $\mathbb{R}^n$.

For (2), we can simply apply the Gram-Schmidt process to find an orthonormal set of $n$ vectors. The Gram-Schmidt process takes a finite, linearly independent set $U = \{\vec{u}_0, \vec{u}_1, \ldots, \vec{u}_{n-1}\}$ and generates an orthonormal set $W = \{\vec{w}_0, \vec{w}_1, \ldots, \vec{w}_{n-1}\}$ that spans the same space as $U$.

For this problem, we compute the set $W$ based on $V$ first: let $\vec{w}_0 = \vec{v}_0$, $\vec{w}_1 = \vec{v}_1 - \frac{(\vec{w}_0, \vec{v}_1)}{\|\vec{w}_0\|} \vec{w}_0$, and keep computing other $\vec{w}_k$ in $W$ by subtracting $\vec{v}_k$ with its projections upon the prior existing $\vec{w}_0, \vec{w}_1, \ldots, \vec{w}_{k-1}$. After that, we normalize each vector in $W$ to make it an orthonormal basis $V$, where $\vec{v}_0 = \vec{w}_0 / \|\vec{w}_0\| = \vec{w}_1 / \|\vec{w}_1\| = \alpha \vec{u}_0$.

(d) For the main proof, we will proceed by formal induction. Recall that for a proof by induction, we have to start with a base case - this is also the base case in a recursive derivation.

Consider the trivial case of $S$ having dimensions $[1 \times 1]$ ($n = 1$). Does $S$ have an eigenvector? Can this eigenbasis be made orthonormal? Is the matrix diagonal in this basis? Are the entries real?

**Solution:** Yes, it has an eigenvector, because $S$ would be a scalar. Let $S = [s]$ and $\vec{u} = 1$. Then note that $S \vec{u} = s \vec{u}$. This implies that $s$ is an eigenvalue and $\vec{u} = 1$ is an eigenvector. This eigenbasis is orthonormal, because there is no other vector which is not orthogonal to $\vec{u}$. We can think of $S$ as a $1 \times 1$ matrix, such that the only entry is real. Also, $S$ is diagonal in that basis because there is only one element.
(e) After the base case, we do an inductive stage. The first step in the inductive stage is to write down the induction hypothesis. Assume this property holds for all symmetric matrices with size \((n - 1) \times (n - 1)\). Write down the statement of this fact in your own words using mathematical notation. *(Hint: In general for proofs by induction, you want to start with the strongest version of what you want to prove. This gives you the most powerful inductive hypothesis.)*

**Solution:** Any \((n - 1) \times (n - 1)\) real symmetric matrix can be diagonalized by a matrix of its orthonormal real eigenvectors.

Mathematical notation: an \((n - 1) \times (n - 1)\) real symmetric matrix \(Q\) can be diagonalized by its orthonormal real eigenvectors, \(\vec{u}_0, \ldots, \vec{u}_{n-1}\). Let \(U\) be an \(n - 1 \times n - 1\) matrix, where the column vectors are those orthonormal real eigenvectors of \(Q\). Then we can express \(Q\) as \(U \Lambda_Q U^T\), where \(\Lambda_Q\) is a diagonal matrix with real eigenvalues of \(Q\) along its diagonal.

(f) Now think about a symmetric matrix \(S\) with size \([n \times n]\): consider a real eigenvalue \(\lambda_0\) of \(S\), and the corresponding eigenvector \(\vec{v}_0\) (a column vector with size \(n\)). Use an appropriate orthonormal change of basis \(V\) to show that \(S = V \begin{bmatrix} \lambda_0 & \cdots & \cdot \\ \vec{0} & \ddots & \cdot \\ \cdot & \cdots & \cdot \end{bmatrix} V^T\).

**Solution:** According to (c), we could derive an orthonormal set of \(n\) vectors, \(\vec{v}_0, \ldots, \vec{v}_{n-1}\), based on \(\vec{u}_0\). Let \(\vec{v}_0, \ldots, \vec{v}_{n-1}\) be the columns of matrix \(V\). Suppose \(X = V^T S V\), such that \(S = V X V^T\) (because \(V\) is an orthonormal basis, \(V^T V = I\)).

Hence we have

\[
X = \begin{bmatrix} \vec{v}_0^T \\ \vec{v}_1^T \\ \vdots \\ \vec{v}_{n-1}^T \end{bmatrix} S \begin{bmatrix} \vec{v}_0 & \vec{v}_1 & \cdots & \vec{v}_{n-1} \end{bmatrix},
\]

where \(\vec{v}_0\) is the normalized eigenvector corresponding to the eigenvalue \(\lambda_0\) of \(S\), which means \(S \vec{v}_0 = \lambda_0 \vec{v}_0\). The first column of \(X\) will be

\[
S \vec{v}_0 = \begin{bmatrix} \vec{v}_0^T \\ \vec{v}_1^T \\ \vdots \\ \vec{v}_{n-1}^T \end{bmatrix} \begin{bmatrix} \vec{v}_0^T \\ \vec{v}_1^T \\ \vdots \\ \vec{v}_{n-1}^T \end{bmatrix} \lambda_0 \vec{v}_0.
\]

Since \(V\) is an orthonormal matrix, \(\vec{v}_i^T \vec{v}_j = 1\) when \(i = j\); otherwise it is zero. Therefore, the first column of \(X\) is

\[
\begin{bmatrix} \lambda_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix}
\]

which is what we wanted to show.

(g) Continue the previous part to show that actually

\[
S = V \begin{bmatrix} \lambda_0 & \vec{0}^T \\ \vec{0} & \Omega \end{bmatrix} V^T
\]

where \(\Omega\) is an \(n - 1\) by \(n - 1\) symmetric matrix.
Solution: Recall the lemma we proved in (a), $X$ must be a symmetric matrix, because $X = V^T SV$ where $S$ is a symmetric matrix and $V$ is orthonormal. Since we have proved the first column of this matrix is

$$
\begin{bmatrix}
\lambda_0 \\
0 \\
\vdots \\
0
\end{bmatrix},
$$

hence $X$ can be written as

$$
\begin{bmatrix}
\lambda_0 & 0 \\
0 & \tilde{Q}
\end{bmatrix},
$$

where $Q$ is an $n-1$ by $n-1$ symmetric matrix. This is a short direct argument that gets us what we want.

An alternative approach was shown in discussion 3A where we computed $X$ directly. Let $R$ be a matrix with columns $\vec{v}_1, \ldots, \vec{v}_{n-1}$, which are orthogonal unit vectors. Recall that $\lambda_0$ is an eigenvalue of $S$, with the corresponding eigenvector $\vec{v}_0$. From here, we can compute $X$ step by step:

$$
X = V^T SV = [\vec{v}_0 \ R]^T S [\vec{v}_0 \ R] = [\vec{v}_0^T \ R^T] S [\vec{v}_0 \ R] = [\vec{v}_0^T \ R^T] [SV_0 \ SR] = [\vec{v}_0^T \ R^T] [\lambda_0 \vec{v}_0 \ SR]
$$

Now we could write $X$ as

$$
\begin{bmatrix}
\lambda_0 \vec{v}_0^T & \vec{v}_0^T SR \\
\lambda_0 R^T \vec{v}_0 & R^T SR
\end{bmatrix},
$$

Because $\vec{v}_0$ is a unit vector, $\vec{v}_0^T \vec{v}_0 = 1$. Since $V$ is an orthonormal matrix, the inner product of $\vec{v}_0$ with column vectors in $R$ must be 0, so $R^T \vec{v}_0 = \vec{0}$. Also, $\vec{v}_0^T SR = (S\vec{v}_0)^T R = \lambda_0 \vec{v}_0^T R = \vec{0}^T$.

Then we get the desired form of $X$, where $Q = R^T SR$. Recall the lemma we proved in (a), because $S$ is a symmetric matrix, while the columns of $R$ are orthonormal, $Q$ must be symmetric.

Either one of these approaches is fine.

(h) According to our induction hypothesis, we can write $Q$ as $U\Lambda U^T$ where $U$ is an orthonormal $n-1$ sized square matrix and $\Lambda$ is a diagonal matrix filled with real entries. Use this fact to show that indeed there must exist an orthonormal $n$-sized square matrix $W$ so

$$
S = W \begin{bmatrix} \lambda_0 & \vec{0}^T \\ \vec{0} & \Lambda \end{bmatrix} W^T
$$

(\text{Hint: What is the product of orthonormal matrices?})

Solution:

From the previous part, we know that, $S$ can be written as follows.

$$
S = [\vec{v}_0 \ R] \begin{bmatrix} \lambda_0 & \vec{0}^T \\ \vec{0} & \Lambda \end{bmatrix} \begin{bmatrix} \vec{0}^T \\ R^T \end{bmatrix}
$$

Thanks to our induction hypothesis, we can compute $S$ as follows:

$$
S = [\vec{v}_0 \ R] \begin{bmatrix} \lambda_0 & \vec{0}^T \\ \vec{0} & U\Lambda U^T \end{bmatrix} \begin{bmatrix} \vec{0}^T \\ R^T \end{bmatrix}
$$

At this point, what we would like to do is to pull the $U$s out of the inner matrix and stick them with the $R$s in the outer matrices. Then we would be done. This could be done immediately because of the properties of block matrix multiplication, or we could further justify it.
If you wanted to further justify it (this was not required for full credit), we could calculate as follows:

\[
\begin{bmatrix}
\vec{v}_0 & R \\
\lambda_0 & U \Lambda U^T
\end{bmatrix}
\begin{bmatrix}
\vec{v}_0^T \\
R^T
\end{bmatrix} = \begin{bmatrix}
\lambda_0 \vec{v}_0 & RU \Lambda U^T \\
0 & \Lambda
\end{bmatrix}
\begin{bmatrix}
\vec{v}_0^T \\
R^T
\end{bmatrix} = (\begin{bmatrix}
\lambda_0 \vec{v}_0 & O_{(n \times n-1)} \\
0 & RU \Lambda U^T
\end{bmatrix})
\begin{bmatrix}
\vec{v}_0^T \\
R^T
\end{bmatrix},
\]

where \( O_{(n \times n-1)} \) is a matrix filled with zeros.

Then

\[
S = \begin{bmatrix}
\lambda_0 \vec{v}_0 & \vec{v}_0^T \\
0 & \Lambda
\end{bmatrix} = \begin{bmatrix}
\lambda_0 \vec{v}_0 & O_{(n \times n-1)} \\
0 & RU \Lambda
\end{bmatrix}
\begin{bmatrix}
\vec{v}_0^T \\
R^T
\end{bmatrix}.
\]

Then we can place \( S \) into the orthonormal basis \( W = [\vec{v}_0 \ RU] \) to verify:

\[
S = \begin{bmatrix}
\vec{v}_0 & RU \\
\lambda_0 & U \Lambda
\end{bmatrix}
\begin{bmatrix}
\vec{v}_0^T \\
U^T \Lambda
\end{bmatrix} = \begin{bmatrix}
\lambda_0 \vec{v}_0 & O_{(n \times n-1)} \\
0 & RU \Lambda
\end{bmatrix}
\begin{bmatrix}
\vec{v}_0^T \\
U^T \Lambda
\end{bmatrix},
\]

which also becomes

\[
\lambda_0 \vec{v}_0 \vec{v}_0^T + [RU \Lambda U^T R^T].
\]

Regardless of whether we felt required to justify pulling the \( U \)'s out, we know that the multiplication of two orthonomal matrices (in this case, \( R \) and \( U \)) are orthonormal as well.

It is easy to further verify that \( \vec{v}_0 \) is orthonormal to all columns of \( RU \) by seeing what happens to \( \vec{v}_0^T RU \):

\[
\vec{v}_0^T RU = (\vec{v}_0 R) U = \vec{v}_0 U = \vec{0}^T,
\]

because \( \vec{v}_0 \) is orthogonal to column vectors of \( R \). Therefore, we can now define the orthonormal basis \( W \) as,

\[
W = [\vec{v}_0 \ RU]
\]

This shows what we wanted to prove:

\[
S = W \begin{bmatrix}
\lambda_0 & \vec{0}^T \\
0 & \Lambda
\end{bmatrix} W^T
\]

and we are done.

By induction, we are now done since we have proved that having the desired property for \( n - 1 \) implies that we have the property for \( n \) and we also have a valid base case at \( n = 1 \).

According to the base case and inductive steps we just proved, the statement, “every real symmetric matrix is diagonalized by a matrix of its real orthonormal eigenvectors” is proved by induction.

**Solution:** This is called the spectral theorem for real symmetric matrices. After circulant matrices, it is perhaps the easiest spectral theorem to prove. A similar theorem holds for complex matrices that are Hermitian. There is also a general spectral theorem for all matrices, but it is more involved to state and prove, and this is done in Math 110. Because of the engineering value of the SVD however, we find ourselves more often leaning on the spectral theorem for real symmetric matrices, so it is good for you all to understand this proof. The proof here also has this nice recursive character to it that goes along well with what you have learned in our sister courses 61ab.

**5. Eigenfaces**

In this problem, we will be exploring the use of PCA to compress and visualize pictures of human faces. We use the images from the data set Labeled Faces in the Wild. Specifically, we use the set with all images.
aligned using deep funneling to ensure that the faces are centered in each photo. Each image is a 250x250 image with the face aligned in the center. To turn the image into a vector, we stack each column of pixels in the image on top of each other, and we normalize each pixel value to be between 0 and 1. Thus, a single image of a face is represented by a 625,000 dimensional vector, but a vector this size is a bit challenging to work with directly. We combine the vectors from each image into a single matrix and run PCA on it. For this problem, we will provide you with the first 250 principal components, but you can explore how well the images are compressed with fewer components. Please refer to the IPython notebook to answer the following questions.

(a) We provide you with a set of faces from the training set and compress them using the first 100 principal components. You can adjust the number of principal components used to do the compression. What changes do you see in the compressed images when you used a small number of components and what changes do you see when you use a large number?

**Solution:** When fewer principal components are used, the images appear to have a general “head” shape, but do not contain the distinguishing features that would identify the face. When more principal components are used, the images more closely resemble the originals.

(b) You can visualize each principal component to see what each dimension “adds” to the high-dimensional image. What visual differences do you see in the first few components compared to the last few components?

**Solution:** The first few principal components are blurry images, which can be interpreted as the general shape of the head that appears in all of the training images. Because the training images are aligned with the nose at the center, the first few principal components also pick up on the general shape of the facial features (eyes, nose, and mouth) because they appear in the same location in each image. The last few components look like they contain a lot of noise, but this is probably for distinguishing features of a small set of images in the training set and capturing things like the specific background in the photo.

(c) By using PCA on the face images, we obtain orthogonal vectors that point in directions of high variance in the original images. We can use these vectors to transform the data into a lower dimensional space and plot the data points. In the notebook, we provide you with code to plot a subset of 400 images using the first two principal components. Try plotting other components of the data, and see how the shape of the points change. What difference do you see in the plot when you use the first two principal components compared with the last two principal components? What do you think is the cause of this difference?

**Solution:** The variance of the points in the plot is larger for the first two components compared to the last two components. We can also confirm that the variance is larger for the first few components because the singular values are large while the singular values for the last few components are small. This happens because PCA orders the principal components by the singular values, which can be used to measure the variability in the data for each component.

(d) We can use the principal components to generate new faces randomly. We accomplish this by picking a random point in the low-dimensional space and then multiplying it by the matrix of principal components. In the notebook, we provide you with code to generate faces using the first 250 principal components. You can adjust the number of components used. How does this affect the resulting images?

**Solution:** When fewer components are used, the faces appear more similar and when a very small number of principal components are used, they are almost indistinguishable. When we use more principal components, the synthesized faces appear more distinct. This happens because we are adding
more degrees of freedom to our “face” vector when we add more principal components. This allows us to generate faces with more variety because we have more parameters that control how the face looks.

6. Image Processing by Clustering  
In this homework problem, you will learn how to use the k-means algorithm to solve two image processing problems: (1) color quantization and (2) image segmentation.

Digital images are composed of pixels (you could think of pixels as small points shown on your screen.) Each pixel is a data point (sample) of an original image. The intensity of each pixel is its feature. If we use 8-bits integer $Var_i$ to present the intensity of one pixel, $(Var_i = 0)$ means black, while $(Var_i = 255)$ means white. Images expressed only by pixel intensities are called grayscale images.

In color image systems, the color of a pixel is typically represented by three component intensities (features) such as Red, Green, and Blue. The features of one pixel is a list of three 8-bit integers: $[Var_r, Var_g, Var_b]$. Here $[0, 0, 0]$ means black, while $[255, 255, 255]$ presents white. You could play with this website, http://www.colorschemer.com/online.html, to see how values of $[Var_r, Var_g, Var_b]$ influence the color.

Now think about your own experience: Have you needed to delete some photos on your cell phones because you ran out of the memory? Have you felt "why does it take forever to upload my photos”? We need ways to reduce the memory size of each photo by image compression.

In computer graphics, there are two types of image compression techniques: lossless and lossy image compression. The former one is preferred for archival purposes, which aim at maintaining all features but reducing the required memory of one image, while the latter one tries to remove some irrelevant and redundant features without destroying the main features of this image. In this problem, you will learn how to use the k-means algorithm to perform lossy image compression by color quantization.

Color quantization is one way to reduce the memory size of an image. In real schemes, it complements the DCT/DFT/Haar-based techniques you saw on the last homework, and is applied there to the coefficients in that basis, but here we do it directly to pixels for simplicity. The target of color quantization is to reduce the number of colors used in an image, while trying to make the new image visually similar to the original image. An image stored in Graphics Interchange Format (GIF) is an example.

For example, consider an image of the size $800 \times 600$, where each pixel takes 24 bits (3 bytes) to store its color intensities. This raw image takes $800 \times 600 \times 3$ bytes, about 1.4MB to store. If we use only 8 colors to represent all pixels in this image, we could include a color map, which stores the RBG values for these representation colors, and then use 3 bits for each pixel to indicate which one is its representation color. In that case, the compressed image will take $8 \times 24 + 800 \times 600 \times 3$ bits, about 0.2MB, to store it. We save memory.

There are two main tasks in color quantization: (1) decide the representation colors, and (2) determine which representation color each pixel should be assigned. Both tasks can be done by the k-means algorithm: It groups data points (pixels) into $k$ different clusters (representation colors), and the centroids of the clusters will be the representation colors for those pixels inside the cluster.

Here is another thing we can do with this flow: image segmentation. Image segmentation partitions a digital image into multiple segments (sets of pixels.) It is typically used to locate boundaries of objects in images. Once we isolate objects from images, we can perform object detection and recognition, which play essential roles in artificial intelligence. This can be done by clustering pixels with similar features and labeling them by an indicating color of each cluster (object).

(a) Please look at the ipython notebook file, there is a 4 by 4 grayscale image. Perform the k-means algorithm on the 16 data points with $k = 3$. What are the representation colors (centroids)? Show the image after color quantization.
**Solution:** See sol3.ipynb. There are multiple local optima, so we have code for you to check your own answers. Here we choose [0, 135, 175] as the initial centroids, while the initial clusters are {0, 63, 27} for 0, {111, 113, 115, 89, 135, 122, 123, 134} for 135, and {178, 175, 234, 235, 169} for 175. Then centroids become [30, 117.75, 198.2]. The clusters are still the same. Hence we have {0, 63, 27} for 30, {111, 113, 115, 89, 135, 122, 123, 134} for 117.75, and {178, 175, 234, 235, 169} for 198.2.

(b) See ipython notebook. Apply the k-means algorithm to the grayscale image with different values of k. Observe the distortion. Choose an ideal value for k, which should be the minimum value for keeping the compressed image visually similar to the original image. Calculate the memory we need for the compressed image.

**Solution:** When you adjust the value of $k_{gray}$, the higher $k_{gray}$ will make the image more similar to the original image, which means the value of $distortion_{gray}$ is smaller. For this image, 8 different colors might be enough, but some might feel we need more. This image we use is of size $597 \times 640$. If you set $k_{gray} = 8$, we need $8 \times 8 + 597 \times 640 \times 3$ bits to store it, where we need 8 bits for each representation color in grayscale for the color map, and 3 bits for each pixel to indicate its representation color.

(c) See ipython notebook. Apply the k-means algorithm to the color image with different values of k. Observe the distortion. Choose an ideal value for k, calculate the memory we need for the compressed image.

**Solution:** When you adjust the value of $k_{color}$, the higher $k_{color}$ will make the image more similar to the original image, which means the value of $distortion_{color}$ is smaller. It is difficult to express this image properly by only 8 or 16 colors, because there are color blends (gradients) in the sky of this image. This image we use is of size $384 \times 256$. If you set $k_{color} = 64$, we need $24 \times 64 + 384 \times 256 \times 6$ bits to store it, where we need 24 bits for the RGB values of each representation color, and 6 bits for each pixel to indicate its representation color.

(d) See ipython notebook. Here we combine three features of each pixel into one feature as the intensity in grayscale images. Use the k-means algorithm to locate the boundaries of objects. How many objects do you expect from this image? Adjust the value of k and describe your observations. If you are interested in this, you could learn more algorithms for image segmentation from EECS courses in image processing and computer vision.

**Solution:** See sol3.ipynb. There are 3 objects in this image: the fish, sea and coral. If you run the k-means algorithm with $k_{segment\_intensity} = 3$, you will see the boundary between the sea and the two other objects is clear. However, we cannot extract the fish and coral perfectly because pixels for each object do not have similar intensities. This issue cannot be resolved by increasing the value of $k_{segment\_intensity}$. In advanced courses, you will learn how to properly incorporate information from neighboring pixels into features of one pixel.

7. **Your Own Problem**

Write your own problem related to this week’s material and solve it. You may still work in groups to brainstorm problems, but each student must submit a unique problem. What is the problem? How to formulate it? How to solve it? What is the solution?

**Contributors:**

- Saavan Patel.
- Siddharth Iyer.
• Yu-Yun Dai.
• Stephen Bailey.