CS215 Assignment 2 Report

Arpon Basu
Shashwat Garg

Autumn 2021

Contents

1 Introduction

Hello and welcome again to our report on CS215 Assignment 2. **This time we have used MATLAB instead of Python**. We have tried to make this report comprehensive and self-contained. We hope reading this would give you a proper flowing description of our work, methods used and the results obtained. Feel free to keep our code scripts alongside to know the exact implementation of our tasks. The pictures included in the graphs folder are a part of this report as well.

We have referred to some sites on the web for finding the MATLAB implementations (generic documentation pages) and general statistical knowledge needed for various parts of the assignment.

In many places, to better give context to the place from which the questions could have arisen, some theoretical discussions have been engaged in.

Hope you enjoy reading the report. Here we go!

2 Some Logistic Issues

Note that the version of MATLAB we used for developing our code has the `Statistics and Machine Learning Toolbox,v12.2` installed in it. **Note that we have not used any function which was explicitly forbidden, nor have we used any function which wasn't explicitly forbidden but had it been used would have defeated the spirit in which the question was posed**. Despite this, some very ordinary functions we needed were not available in the vanilla version of MATLAB and needed these toolboxes: Case in point, the `boxplot` utility is available only in the `Statistics and Machine Learning Toolbox`, not in the ordinary one.
Thus, when the graders will run our code in their machine, they should take care to install this toolbox to ensure the smooth execution of our code.
Note that some questions (especially Q2,6) may take 5-10 seconds to execute.
Also note that due to some MATLAB peculiarities, when you run our code for the *very first time*, the `saveas` functions in our code *may* throw an error (because you're trying to run our code in a "foreign" directory, it *may* throw a `handle not found error`). However, when you run it the second time, it will run smoothly without any glitches. Thus if your machine throws an error the first time, press the run button again.
In many cases, especially where many images were asked for, we have included only a few representative images for purposes of clarity, and the rest have been stored in the **Results directory** of our submission for the grader's perusal.

3 Problem 1

3.1 Our Approach

In this question we have been asked to propose implementable algorithms for generating random points (in 2D) distributed uniformly inside an Ellipse and a Triangle.

Further, in the actual implementation, we are asked to sample a bunch of points from the distribution and plot a 2d histogram of the same.

Thinking about it for a while, we actually came up with a fairly simple method. We define a bounding box “rectangle” which completely but minimally surrounds our region of interest. Then we sample uniform points in the rectangle using a pair of the rand() function. The catch is, we only keep those points which lie in our region and reject the rest, a method known as **rejection sampling**.

This gives us a simple algorithm to sample any number of points within any complex region uniformly.

The other approach could have been to partition the ellipse into different slices of variable width and write complex equations to ensure overall uniformity. We think that our method works just as well computationally and is much more elegant.

3.2 Code Flow

As overviewed in the previous subsection, the code for this part is fairly simple.

We specify the dimensions of the rectangle and also create zero matrices for storing the random variables.

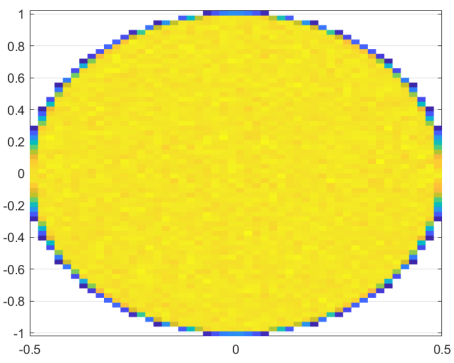
Next is a loop which generates a unique point (uniformly and randomly) inside the rectangle, followed by a simple condition check if the point lies within the ellipse or triangle.

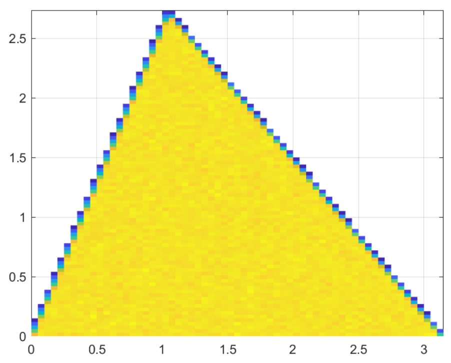
If lying inside the figure, we increment the count variable and repeat the process. Finally we create a 2-D histogram with the specified attributes.

This finishes this question’s coding part.

3.3 Graphs and the Results obtained

The results obtained are as follows-





3.4 Observations and their Rationalization

We discovered a simple method to create random variables coming from any well defined shape in the cartesian plane.

We also learnt about various parameters in the histogram2 function of MATLAB. THough only one sample is present in the report, we plotted 3d plots and adjusted the different attributes to better understand the capabilities of the software.

Our approach will struggle as the shapes get more complicated. To define whether the point lies inside or outside the region might require a series of if else condition. But this approach is computationally capable of any uniform plot, however complex, atleast in theory.

4 Problem 2

4.1 Introduction

This problem involves the generation of Gaussian samples and subsequently evaluate their Maximum Likelihood estimators and observe the convergence of those estimators as sample size grows larger and larger. The first two points (generation of samples and their likelihood estimators) have salient theoretical points to be talked about, and that is done in the following two subsections.

Then the results will be presented and some analysis will be done on those outcomes.

4.2 Generation of Samples with given mean and Covariance matrices

One of the major theoretical components in the question was the **generation of a sample of N multivariate Gaussians with given μ and Σ** from only a standard 2-D "unit" Gaussian, $\mathcal{N}(\mathbf{0}, \mathbf{I_2})$. This required the application of two crucial probability theory and linear algebra results applied in tandem, which we describe below:

- If $X \sim \mathcal{N}(\mu, \mathbf{\Sigma})$, then for $Y := AX + b$, where $A \in \mathbb{R}^{2\times 2}$ is any matrix and $b \in \mathbb{R}^{2\times 1}$ is any column vector, we have $Y \sim \mathcal{N}(\mathbf{A}\mu + \mathbf{b}, \mathbf{A\Sigma A^T})$
- Any **symmetric** matrix **A** can be expressed as $\mathbf{UDU^T}$ where $\mathbf{U} = \begin{bmatrix} u_1 & ... & u_n \end{bmatrix}$ where $u_1, u_2, ..., u_n \in \mathbb{R}^{n\times 1}$ are the **orthonormal** eigenvectors of A, and $\mathbf{D} = \text{diag}(\lambda_1, \lambda_2, ..., \lambda_n)$ is the diagonal matrix of the corresponding eigenvalues

Coming to the problem statement, we were first asked to generate a Gaussian with mean $\mu = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and covariance $C = \begin{bmatrix} 1.6250 & 1.9486 \\ -1.9486 & 3.8750 \end{bmatrix}$. Now, note that if we can express C as AA^T for some matrix A (such a decomposition is known as the **Cholesky decomposition of** C), then we have for $X := AG + \mu$; $G \sim \mathcal{N}(\mathbf{0}, \mathbf{I_2})$, $X \sim \mathcal{N}(\mu, \mathbf{AA^T}) = \mathcal{N}(\mu, \mathbf{C})$, ie- we would have generated a sample from our desired probability distribution function.

Then note that the decomposition of C into AA^T can be obtained from it's diagonalization. Indeed, let $A := US$, where $\mathbf{S} = \text{diag}(\sqrt{\lambda_1}, \sqrt{\lambda_2}, ..., \sqrt{\lambda_n})$. Then $AA^T = US \cdot S^T U^T = US \cdot SU^T = UDU^T = C$.

In our case, for C, we have $U = \begin{bmatrix} -0.8660 & -0.5000 \\ -0.5000 & 0.8660 \end{bmatrix}$ (the 2 column vectors are the eigenvectors of C), while the respective eigenvalues of C are $\frac{1}{2}$ and 5 respectively, making $\mathbf{S} = \text{diag}(\frac{1}{\sqrt{2}}, \sqrt{5})$ (note that any of $\text{diag}(\pm\frac{1}{\sqrt{2}}, \pm\sqrt{5})$ work).

Thus, for $C = \begin{bmatrix} 1.6250 & 1.9486 \\ -1.9486 & 3.8750 \end{bmatrix}$, we have it's decomposition A to be equal to $US =$

$\begin{bmatrix} -0.6123 & -1.1180 \\ -0.3535 & 1.9365 \end{bmatrix}$.

Thus, finally, ($\begin{bmatrix} -0.6123 & -1.1180 \\ -0.3535 & 1.9365 \end{bmatrix} \cdot$ `randn(2,1)` $+ \begin{bmatrix} 1 \\ 2 \end{bmatrix}$) gives us a randomly chosen data-point from a Gaussian with mean $\begin{bmatrix} 1 \\ 2 \end{bmatrix}$ and covariance $\begin{bmatrix} 1.6250 & 1.9486 \\ -1.9486 & 3.8750 \end{bmatrix}$.

4.3 Maximum Likelihood Estimators for mean and Covariance

Given a sample of size N of points sampled from an unknown probability distribution, what are the **Maximum Likelihood Estimators** for their mean and covariance if the underlying probability distribution is assumed to be a Gaussian?
Let our points be $\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_N}$ (note that each of these points is a 2D column vector). Then the **log-likelihood function** (assuming independently sampled data-points) assuming a Gaussian $\mathcal{N}(\mu, \mathbf{\Sigma})$ probability distribution would be:-

$$l(\mu, \Sigma) = \log(\prod_{k=1}^{N} \frac{1}{(2\pi)^{2/2}\sqrt{|\Sigma|}} \exp(-\frac{1}{2}(\mathbf{x_k} - \mu)^T \Sigma^{-1} (\mathbf{x_k} - \mu)))$$

$$= -N \log(2\pi) - \frac{N}{2} \log(|\Sigma|) - \frac{1}{2} \sum_{k=1}^{N} (\mathbf{x_k} - \mu)^T \Sigma^{-1} (\mathbf{x_k} - \mu)$$

Differentiating this w.r.t μ (note that we are differentiating a scalar w.r.t a vector) to obtain the maximum likelihood estimator $\hat{\mu}$ for it, we have

$$\frac{\partial}{\partial \mu}(-N \log(2\pi) - \frac{N}{2} \log(|\Sigma|) - \frac{1}{2} \sum_{k=1}^{N} (\mathbf{x_k} - \mu)^T \Sigma^{-1} (\mathbf{x_k} - \mu))$$

$$= -\frac{1}{2} \sum_{k=1}^{N} \frac{\partial}{\partial \mu}((\mathbf{x_k} - \mu)^T \Sigma^{-1} (\mathbf{x_k} - \mu))$$

We now use a standard result from matrix calculus (click here) which says that $\frac{\partial \mathbf{x^T A x}}{\partial \mathbf{x}} = 2\mathbf{Ax}$. Thus, our differentiation from above simplifies into

$$= \frac{1}{2} \sum_{k=1}^{N} (\Sigma^{-1}(\mathbf{x_k} - \mu)) = \frac{1}{2} \Sigma^{-1} \sum_{k=1}^{N} (\mathbf{x_k} - \mu)$$

Thus,

$$\hat{\mu} = \arg(\frac{\partial l}{\partial \mu} = 0) = \arg(\frac{1}{2} \Sigma^{-1} \sum_{k=1}^{N} (\mathbf{x_k} - \mu) = 0) = \arg(\sum_{k=1}^{N} \mathbf{x_k} = N\mu)$$

Thus, the **Maximum Likelihood Estimator for mean** $\hat{\mu}$ is $\mathbf{\frac{1}{N} \sum_{k=1}^{N} x_k}$.
Coming to the **Maximum Likelihood Estimator for covariance** , we have

$$\frac{\partial l}{\partial \Sigma} = -\frac{N}{2} \frac{\partial \log(|\Sigma|)}{\partial \Sigma} - \frac{1}{2} \sum_{k=1}^{N} \frac{\partial((\mathbf{x_k} - \mu)^T \Sigma^{-1} (\mathbf{x_k} - \mu))}{\partial \Sigma}$$

Now, by the properties of matrix calculus $\frac{\partial \log(|A|)}{\partial A} = (A^{-1})^T$. Also, note that since $(\mathbf{x_k}-\mu)^T\Sigma^{-1}(\mathbf{x_k}-\mu)$ is a scalar, $(\mathbf{x_k}-\mu)^T\Sigma^{-1}(\mathbf{x_k}-\mu) = \mathrm{tr}((\mathbf{x_k}-\mu)^T\Sigma^{-1}(\mathbf{x_k}-\mu)) = \mathrm{tr}((\mathbf{x_k}-\mu)^T(\mathbf{x_k}-\mu)\Sigma^{-1})$ (since the trace of product of matrices is independent of order of multiplication). The final result from matrix calculus that we use is that $\frac{\partial(\mathrm{tr}(MX^{-1}N))}{\partial X} = -(X^{-1}NMX^{-1})^T$.
Thus, the differential can be simplified as (keeping in mind Σ is symmetric):-

$$\frac{\partial l}{\partial \Sigma} = -\frac{N}{2}\Sigma^{-1} - \frac{1}{2}\Sigma^{-1}(\sum_{k=1}^{N}(\mathbf{x_k}-\mu)(\mathbf{x_k}-\mu)^T)\Sigma^{-1}$$

Thus,

$$\hat{\Sigma} = \arg(-\frac{N}{2}\Sigma^{-1} - \frac{1}{2}\Sigma^{-1}(\sum_{k=1}^{N}(\mathbf{x_k}-\mu)(\mathbf{x_k}-\mu)^T)\Sigma^{-1} = 0) = \frac{1}{N}\sum_{k=1}^{N}(\mathbf{x_k}-\hat{\mu})(\mathbf{x_k}-\hat{\mu})^T$$

Thus, the **Maximum Likelihood Estimator for covariance** $\hat{\boldsymbol{\Sigma}}$ is $\mathbf{\frac{1}{N}\sum_{k=1}^{N}(x_k-\hat{\mu})(x_k-\hat{\mu})^T}$ (note that the mean variable was replaced at the last stage by it's estimator $\hat{\mu}$).

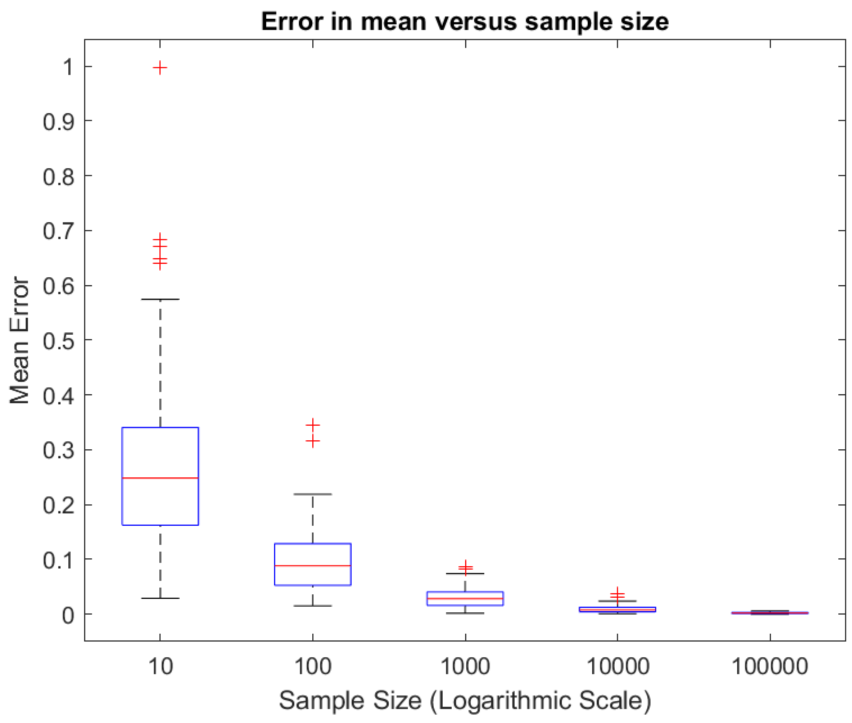
4.4 Results

Coming to the results, we first compile a set of $(\hat{\mu},\hat{\Sigma})$ for N varying from 10 to 10^5, We observer a convergence towards the "actual" values of $\mu = \begin{bmatrix}1\\2\end{bmatrix}$ and covariance $\Sigma = \begin{bmatrix}1.6250 & 1.9486\\ -1.9486 & 3.8750\end{bmatrix}$. Note that the component wise-convergence is oscillatory : Some components of $\hat{\Sigma}$ were greater than their counterparts for initial iterations, but then fell off, and approached the limit from below.
The magnitude of distance between the MLEs and their counterparts was measured through various norms (for means, the obvious vector-norms of their difference was taken, while for matrices, the **Frobenius norm** of the difference matrix was taken), and these error norms were collected over a 100 experiments.
The box plots of those errors (vector errors for $\hat{\mu}$ and Frobenius errors for $\hat{\Sigma}$) show a **clear trend towards zero**, and **their variance**, a measure of which is given by the height of the boxes **also decreases**. This can be explained at an intuitive level by the **Law of Large Numbers**, which can be used to prove why these **estimators converge towards their true values** and why the **variance of the errors goes to zero**.
Finally, coming to the scatter-plots, one can prove that the **direction of the principal empirical eigenvector** is actually **the direction of maximum variance** (a result from the PCA algorithm), a fact that is confirmed visually too from the diagrams. Also, the **length equal to the square root of the eigenvalue** on each side (we have purposefully **kept the line bidirectional** w.r.t the mean) gives us a rough estimate till what length is the sample densely populated, and after which the population get sparser, because, as we know the **standard deviation of a Gaussian pdf along an eigenvector is equal to the square root of the corresponding eigenvalue**. This is seen beautifully for lower N. For higher N, our eyes aren't able to distinguish between relatively sparse and dense samples due to the high numbers in both places, and hence the result is not so dramatic.

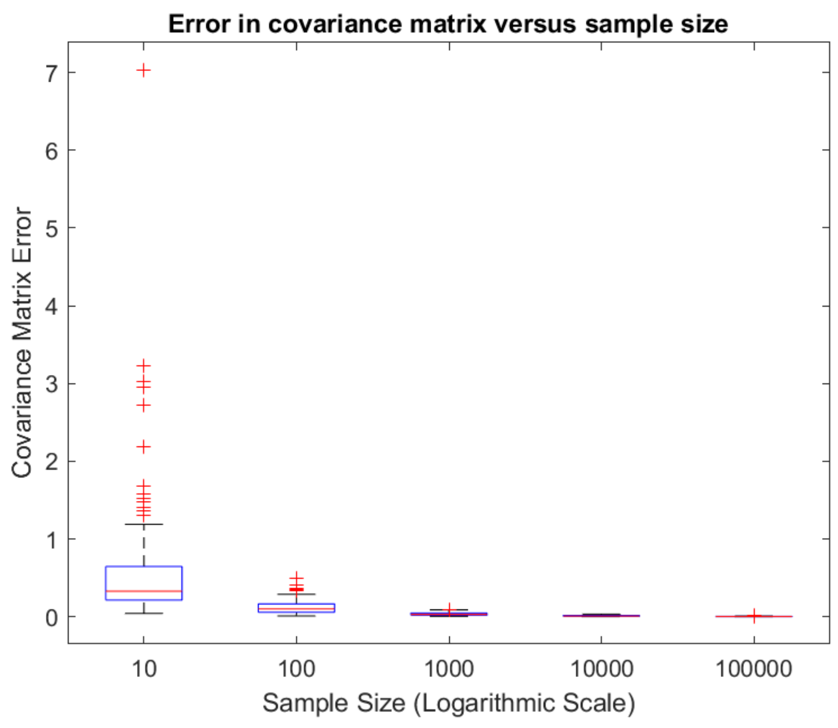
4.4.1 Compilation of MLEs of μ and Σ

Sample Size (N)	$\hat{\mu}$	$\hat{\Sigma}$
10	$\begin{bmatrix}1.0717 & 2.7813\end{bmatrix}^T$	$\begin{bmatrix}2.1312 & -2.3555 \\ -2.3555 & 3.0197\end{bmatrix}$
10^2	$\begin{bmatrix}1.0019 & 2.1488\end{bmatrix}^T$	$\begin{bmatrix}1.8227 & -2.2278 \\ -2.2278 & 4.0470\end{bmatrix}$
10^3	$\begin{bmatrix}0.9193 & 2.1861\end{bmatrix}^T$	$\begin{bmatrix}1.7112 & -2.0642 \\ -2.0642 & 4.0604\end{bmatrix}$
10^4	$\begin{bmatrix}0.9918 & 1.9831\end{bmatrix}^T$	$\begin{bmatrix}1.6415 & -1.9586 \\ -1.9586 & 3.8578\end{bmatrix}$
10^5	$\begin{bmatrix}0.9986 & 2.0029\end{bmatrix}^T$	$\begin{bmatrix}1.6322 & -1.9626 \\ -1.9626 & 3.9063\end{bmatrix}$

4.4.2 Box plot of errors in $\hat{\mu}$



4.4.3 Box plot of errors in $\hat{\Sigma}$



4.4.4 Plots of samples and their directions of maximum variance

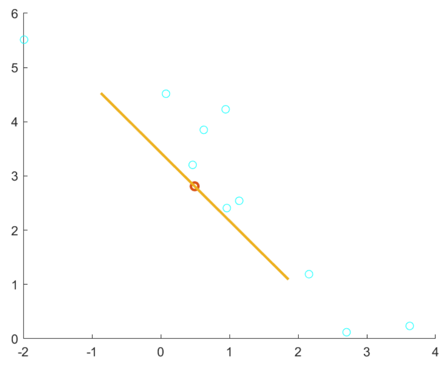


Figure 1: Scatter Plot for N = 10 Random Samples

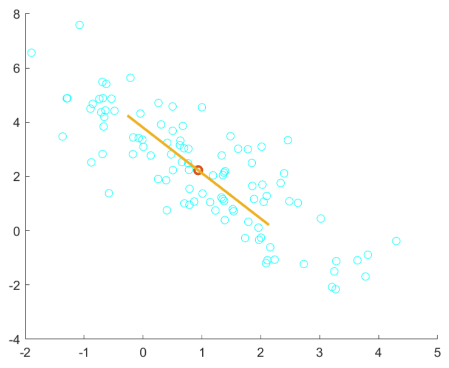


Figure 2: Scatter Plot for N = 100 Random Samples

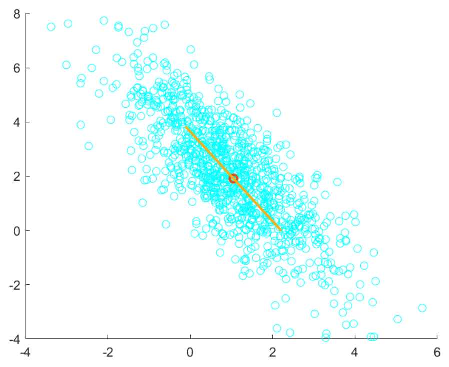


Figure 3: Scatter Plot for N = 1000 Random Samples

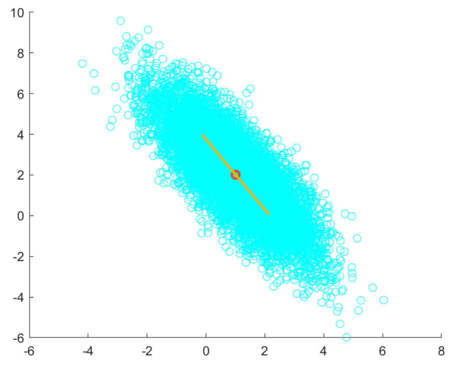


Figure 4: Scatter Plot for N = 10^4 Random Samples

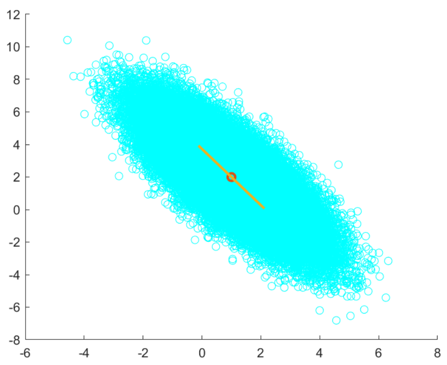


Figure 5: Scatter Plot for N = 10^5 Random Samples

5 Problem 3

5.1 Introduction

This problem primarily involves the estimation of a linear relationship between random variables X and Y, using given sample sets **points2D_Set1.mat** and **points2D_Set2.mat** assumed to be sampled from 2 unknown probability measures $P_1(X, Y)$ and $P_2(X, Y)$. This problem also warns us against lending too much credibility into linear approximations for data-sets which possess a patently non-linear behaviour, case in point : **points2D_Set2.mat**.
Thus, we'll first describe an algorithm which utilizes PCA to generate a linear relationship between our given random variables. We'll then move on to the presentation of it's results and their analysis thereof.

5.2 Description of the PCA Algorithm

We now come on to the question of how exactly would one use PCA to obtain a linear fit: Note that our linear estimator must pass through the mean of the dataset as **PCA** anyways **centres a data-set first before working on it**.
Also note that PCA gives us the **directions of maximum variance** for any dataset. Thus, the direction of our line, in this case would be the direction along which the dataset had maximum variance.
Formulating above insights into a coherent algorithm,

- Suppose our dataset has N points of the form $\{(x_i, y_i)_{1 \le i \le n}\}$. For mathematical purposes below, let $\mathbf{x} = \begin{bmatrix} x_1 & \dots & x_n \end{bmatrix}^T \in \mathbb{R}^{n\times 1}$, $\mathbf{y} = \begin{bmatrix} y_1 & \dots & y_n \end{bmatrix}^T \in \mathbb{R}^{n\times 1}$ and let $\mathbf{z}$ be the concatenation of $\mathbf{x}$ and $\mathbf{y}$, ie:- $\mathbf{z} = \begin{bmatrix} x_1 & y_1 \\ x_2 & y_2 \\ \vdots & \vdots \\ x_n & y_n \end{bmatrix} \in \mathbb{R}^{n\times 2}$.

- In order to find the mean μ of the dataset, we construct the matrix $C = \begin{bmatrix} 1 & 1 & \dots & 1 \\ 1 & 1 & \dots & 1 \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 1 & \dots & 1 \end{bmatrix} \in \mathbb{R}^{n\times n}$ and pre-multiply it with $\mathbf{z}$, and then divide by N so that the resulting $n \times 2$ matrix has for each of it's rows the mean of the data-set μ, ie:- $\mathbf{Cz/N} = \begin{bmatrix} \mu_x & \mu_y \\ \mu_x & \mu_y \\ \vdots & \vdots \\ \mu_x & \mu_y \end{bmatrix} \in \mathbb{R}^{n\times 2}$.
- Subtracting $\mathbf{Cz/N}$ from $\mathbf{z}$ centres the dataset, ie:- we perform $\mathbf{z} -= \mathbf{Cz/N}$ to centre the dataset.
- After that, pre-multiplying $\mathbf{z}$ with it's transpose and dividing by N yields the 2×2 **covariance matrix $\boldsymbol{\Sigma}$ of the dataset**, a crucial step in our PCA analysis, ie:- $\boldsymbol{\Sigma} = \mathbf{z^T z/N}$.
- The primary eigenvector (ie:- the eigenvector corresponding to the largest eigenvalue) of $\boldsymbol{\Sigma}$), say $\mathbf{v}$, is the direction of our line, ie:- if $\mathbf{v} = \begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$, then $\frac{v_2}{v_1}$ is the slope of our line.
- Thus, the PCA linear estimator of our dataset can be given by $v_1(y - \mu_y) = v_2(x - \mu_x)$, where (μ_x, μ_y) is the **mean** of our dataset, while $\begin{bmatrix} v_1 \\ v_2 \end{bmatrix}$ is the **primary eigenvector of the covariance matrix of the dataset**.

5.3 Results and Analysis

The two results of our PCA estimates are attached below.

The results of this problem give us a classic demonstration of the fact that for datasets with a highly non-linear relationship between the random variables X and Y, linear regression, or it's PCA variant, aren't suitable models at the first place to use to analyse these datasets, because if $Y \sim f(X)$ for some highly non-linear function f, then linear fitting or reduction of dimensions through PCA can potentially lead to the loss of a lot of information.
Similar forces are at play in this problem too : In the first dataset, it can be quite easily seen from the dataset that the random variables indeed possess a linear relationship with each other and the deviation are just random noise. In such a case, PCA duly detects the "noise-direction" to be having a low weightage (mirrored by the amplitude of that eigenvalue), and since in the direction of maximum variance a linear relationship most probably does exist, a line passing through the mean along the principal eigenvector simulates the dataset well.
On the other hand, for the second dataset, the direction of maximum variance tells us little about the evolution of the Y random variable, as it goes significantly in both directions, clearly following a non-linear trajectory. As a result, the linear fit obtained is a bad approximation to the dataset and doesn't simulate it well because although it might not

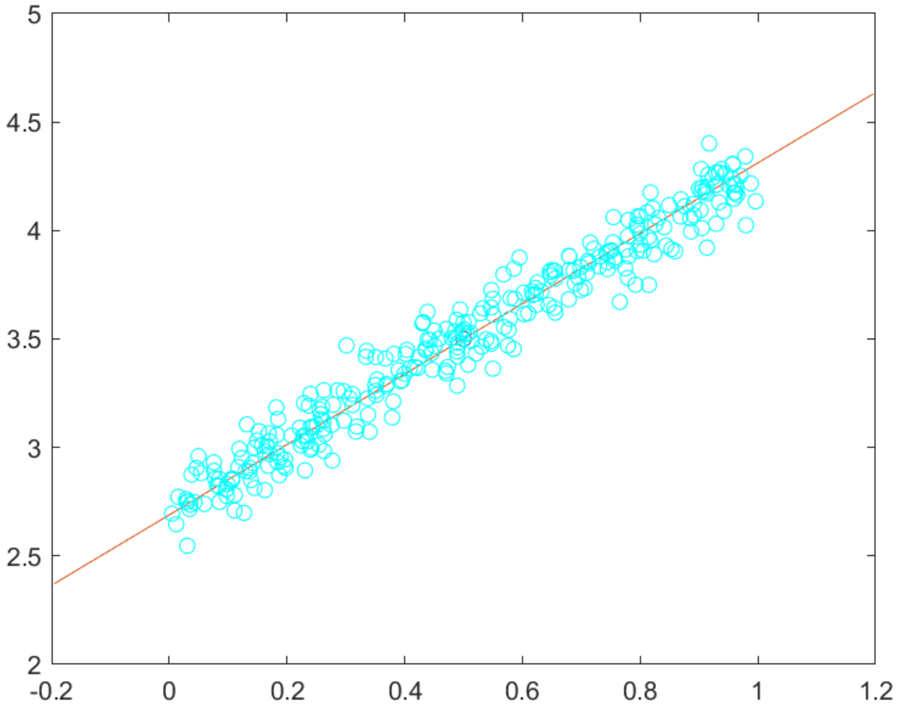


Figure 6: The PCA estimate of Data-set 1

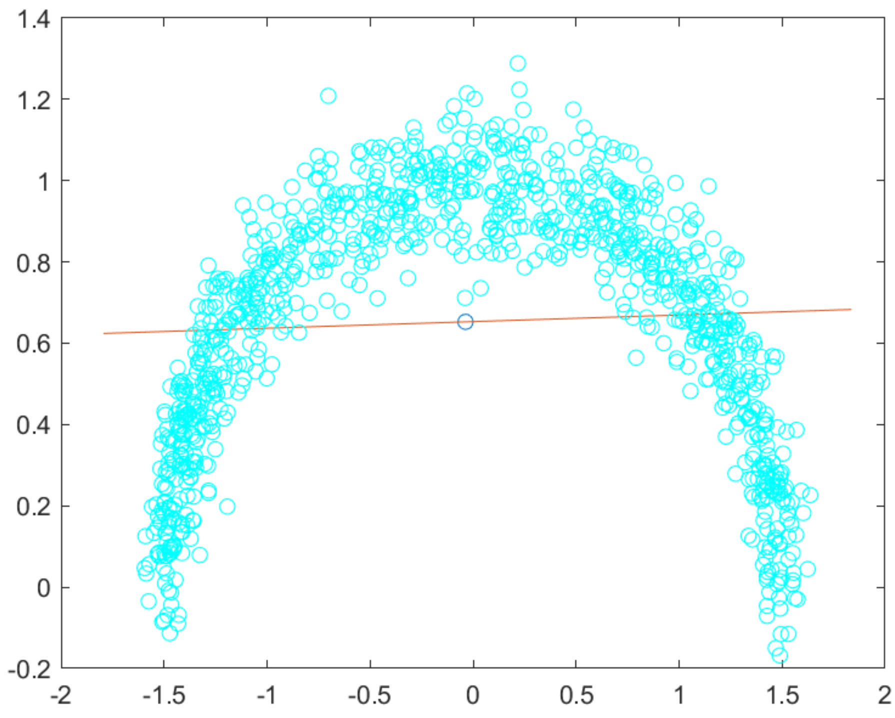


Figure 7: The PCA estimate of Data-set 2

have **two orthogonal directions of high information content (which is what PCA looks out for)**, it does have **2 independent degrees of freedom, which possess a non-linear relationship which PCA fails to capture properly, resulting in a poor simulation.**

6 Problem 4

6.1 Introduction

This problem required us to perform PCA over a popular database for handwritten digits, **MNIST**. While the procedure of PCA in this question is not particularly demanding, the observations it throws up are quite fascinating and worth analysis. Thus, after a brief section where we'll describe the code flow for this question, a significant time will be devoted to the results and analysing them.

6.2 Code Flow

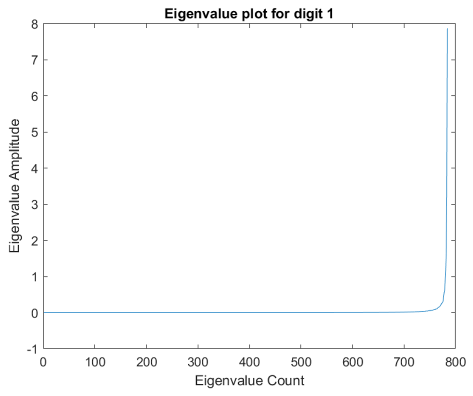
After the `mnist.mat` dataset is loaded onto the workspace, it's reshaped from a $28 \times 28 \times 60000$ array into a $28^2 \times 1 \times 60000$ array. After that, using the `labels_train` map, the mean vector for every digit is calculated. Subsequently, the mean vector is then utilised to center the original database, and once again all the centred column vectors for every digit are collated to form a matrix (of dimensions $28^2 \times 5923$, if say that digit occurred in that database 5923 times). The matrix when multiplied by it's transpose and then divided by the size of it's eliminated dimension (ie:- 5923/6742 etc.) gives us a $28^2 \times 28^2$ covariance matrix for that digit.
Consequently, many operations can be carried out using the covariance matrices, including calculating the principal eigenvalues and eigenvectors, plotting all the 28^2 eigenvalues to see their distribution (including using these to find out the number of **significant modes of variation in a digit: The number of eigenvalues which are at-least 1% of the principal eigenvalue**), and actually using the primary eigenvalues and eigenvectors to reconstruct the $(\mu - \sqrt{\lambda_1} \cdot v_1, \mu, \mu + \sqrt{\lambda_1} \cdot v_1)$ triple of images to see the results. All of these are readily doable using MATLAB's excellent array of utilities, and thus their coding aspect is unremarkable and not mentioned further here.

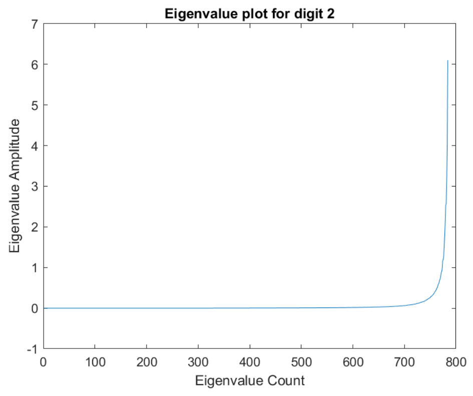
6.3 Results and Analysis

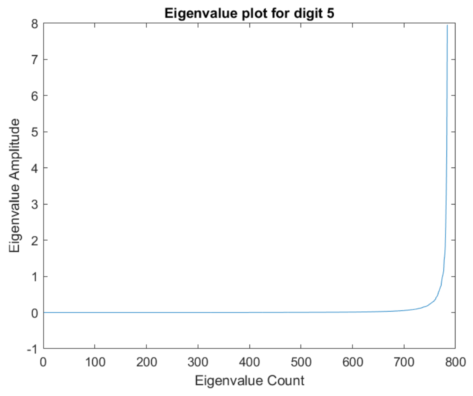
As was confirmed numerically through the code above, that the number of eigenvalues which are *significant* (we've set a threshold of **1% of the maximum to qualify as significant**) is often way lower than 784 (28^2) showing that the **most of the information carried in a 28 × 28 pixel image can be condensed into a much lesser subset of numbers than what the image itself requires, a classic dimension reducing hallmark of the Principal Component Analysis technique**. These conclusions are numerically backed up in the table and eigenvalue plots (notice a very sharp peak towards the right, showing there are very few "large" eigenvalues) below:

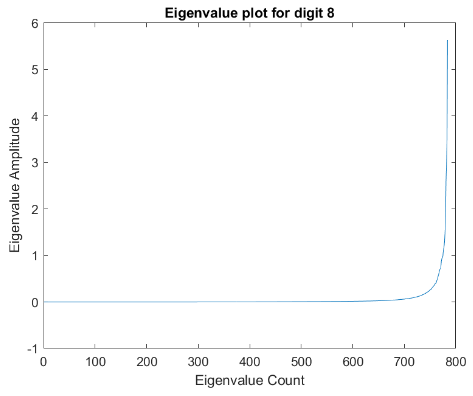
Digit	0	1	2	3	4	5	6	7	8	9
#Significant Variation	56	27	86	86	83	69	60	63	89	63

One may observe here that for relatively simple digits such as “1” (which is just a line), the number of modes of significant variation is very small (just 27: Also note that many of these eigenvalues represent slight **rotations** of the digits, ie:- the same digit 1 tilted by 5° will contribute to a different eigenvalue. Thus the actual number of eigenvalues representing handwriting differences and writing styles is even smaller), while for complicated digits like “8”, the number of significant differences goes higher.

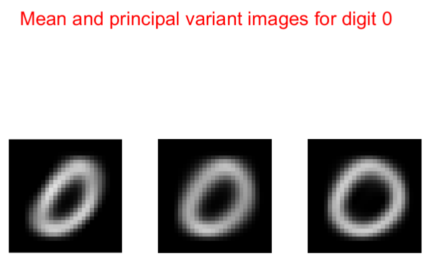


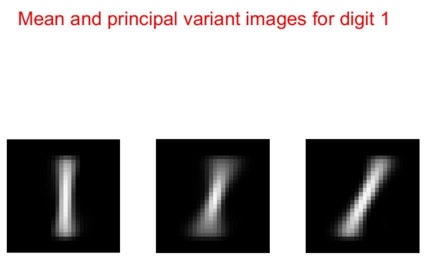


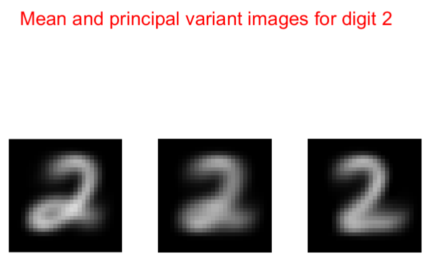


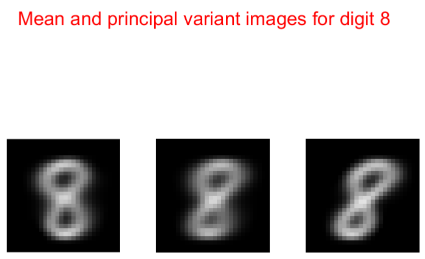


The eigenvalue plots (only a few representative images are shown, find the rest in our submissions folder) clearly show a sharp peak at the right, thus visually confirming what was numerically known: That most of the eigenvalues are insignificant. (While reading the plots, keep in mind that these are eigenvalues of a covariance matrix, which is by definition semi-positive-definite, and hence all the eigenvalues are ≥ 0. However, as is seen by the long tails, most are ≈ 0).
We now present the “$(\mu - \sqrt{\lambda_1} \cdot v_1, \mu, \mu + \sqrt{\lambda_1} \cdot v_1)$” images: What are these? The middle one of these images denotes the mean image, which is kind of representative of how the handwritten digit generally looks, having being averaged over such a large sample. The $\mu - \sqrt{\lambda_1} \cdot v_1$ and $\mu + \sqrt{\lambda_1} \cdot v_1$ components show what aspect of the digit shows maximum diversity in human handwriting. These points will become clearer with the images below (only a few representative images are shown, find the rest in our submissions folder):









The handwriting diversity point made above gets clearer through the images: For the digit “1”, we see that our assumptions of eigenvalues also taking into account affine transformations of the image are well-founded: Indeed the 3 variant images of “1” show a steady rotation to the right direction more so than any handwriting facet, similar seems to be the case with “8”. However, it’s not as though handwriting differences also haven’t been captured: Indeed in the digit “2”, one observes a “knot” near the base of 2 gradually grows smaller as we move from $\mu - \sqrt{\lambda_1} \cdot v_1$ image to the $\mu + \sqrt{\lambda_1} \cdot v_1$, showing that the “knot” of “2” is the place maximum people customize the digit. One also notes that the curve of the digit “2” is more or less same, once again exposing patterns in handwriting. Similar such analyses can be made for every digit, and more or less rotation and/or a variation in a single aspect of the digit are the major modes of variation in the handwriting of these digits.
One may like to reflect upon the role reshaping the 28×28 images into the $28^2 \times 1$ vector played: Indeed, since slices of the square matrix were stacked into a vector, they were able to modify independently from other stacks during the course of matrix manipulations. Thus, when the vector was reshaped back into the square matrix, the side by side overlaying allowed the eigenvectors, which can only bring about affine changes such as rotation and stretching, to actually bring about non-linear variations in the diagrams, as best exemplified in the case of the digit “0” (note that the diameter of “0” changes over the 3 copies) or “2”.

7 Problem 5

7.1 Introduction

This problem represents one of the most fundamental problems in data science, ie:- that of data compression. This problem demonstrates a method to do so via the versatile and wonderful method of **PCA**, and we shall elaborate on that. That will require a little bit theory from linear algebra, which we shall cover in the section below. Once however, all the algorithms are set and done, implementing it through code is quite simple and thus we shall not stress on it much. Thus, without further ado, let's begin the analysis.

7.2 Theoretical Foundations

We already know that the eigenvectors of a covariance matrix give us, in the order of the magnitude of their corresponding eigenvalues, the directions in which the (centred) dataset shows maximum variance.
Suppose now, we want to utilise this knowledge to our advantage and want to achieve data compression using this. How then do we proceed?
The crucial piece of information that gets us going is that for **real symmetric matrices, eigenvectors corresponding to distinct eigenvalues (which BTW are necessarily real) are always orthogonal**. Thus, **the set of unit eigenvectors of a real symmetric matrix forms an orthonormal basis for $\mathbb{R}^n$, where $n \times n$ are the dimensions of the matrix**.
Armed with this knowledge, we can say that for any vector $\mathbf{v} \in \mathbb{R}^n$ and for any real symmetric matrix C (assume all eigenvalues distinct) with the *orthonormal* set of eigenvectors $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, ..., \mathbf{u_n}\}$ we have

$$\mathbf{v} = \sum_{k=1}^{n} < \mathbf{v}, \mathbf{u_k} > \mathbf{u_k}$$

where $< \cdot, \cdot >$ represents the usual inner product of two vectors in $\mathbb{R}^n$.
We now come to the second part of our dimension reduction problem: So far, we have managed to express any vector (keep in mind these vectors are actually images) as linear combinations of our orthonormal eigenvectors. Since there are n eigenvectors, we need n units of information to uniquely store the image, and this is what we precisely don't want to do: We somehow want to store the the image by getting rid of the components which are deemed to be "less important", ie:- somehow we want to make do with $< n$ components.
Luckily, we have a measure of importance already available with us: Remember that the real symmetric matrix in question is our covariance matrix C, and we know from our experience that many of it's eigenvalues are ≈ 0, and we want to get rid of them.
Thus, we decide to keep only the top 84 eigenvectors, which have numerically significant corresponding eigenvalues, and we want to take **an orthogonal projection of our vector w.r.t only those eigenvectors, effectively reducing the dimension of our vector to just 84 dimensions**. Now, once again orthogonal projections are tricky affairs, but in this case, since we have a ready orthonormal basis available, the orthogonal projection of our vector **just truncates it's entire expansion to keep only the desired eigenvectors, ie:-**, if $\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, ..., \mathbf{u_{84}}\} \subset \{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, ..., \mathbf{u_n}\}$ is our desired set of principal eigenvectors, then the projection of any vector $\mathbf{v}$ onto the subspace spanned

by these vectors is just

$$\text{Proj}(\mathbf{v}, \text{span}(\{\mathbf{u_1}, \mathbf{u_2}, \mathbf{u_3}, ..., \mathbf{u_{84}}\})) = \sum_{k=1}^{84} < \mathbf{v}, \mathbf{u_k} > \mathbf{u_k}$$

7.3 Description of the PCA Algorithms

Armed with the theory developed in the section above, we give a description of the algorithms asked for in the question:
The first algorithm asks us to compute the 84 "coordinates" of the projection of our image vector onto the subspace spanned by the top 84 principal eigenvectors. This is done as follows:

- We first obviously have to generate the covariance matrix. Since this has been already done in the previous question, here we assume that we already have our covariance matrix C of size 784×784 available.
- Generate it's matrix of eigenvectors (784 eigenvectors of size 784×1, making a 784×784 matrix in total), keep only the top 84 (ranked on the basis of the magnitude of their corresponding eigenvalues), thus reducing our eigenvector matrix to size 784×84. This matrix is then normalized column by column to ensure that all the eigenvectors in it have norm 1.
- Now take the image of any digit of size 28×28 pixels, reshape it to a **row** vector of dimensions 1×784, and multiply it with the 784×84 eigenvector matrix corresponding to that digit (remember that we have 10 such eigenvector matrices generated from the 10 covariance matrices, one for each digit), to get a row vector of size 1×84. The matrix multiplication basically takes the inner product of our image vector with every column, which are unit eigenvectors, and collates them into the resultant row vector of size 1×84, just as desired.
- **The 84 elements of the above row vector are our desired "coordinates" along the top 84 principal eigenvectors**.

The above algorithm is implemented in the function `calculateCoordinates` in our code for this question.
We now move on to the algorithm for reconstructing the image of any digit below:

- Now that we have our 1×84 coordinate vector, to reshape it back to a 1×784 image vector, **we multiply the coordinate vector with the transpose of our 784×84 eigenvector matrix**
- The above matrix multiplication yields a 1×784 vector, which is reshaped back to a 28×28 matrix, which is nothing but **the matrix of our reconstructed image** !!

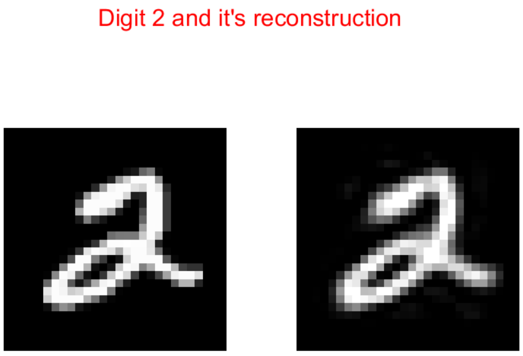
The above algorithm is implemented in the function `reconstructImage` in our code for this question.
In short, to summarize what we did,

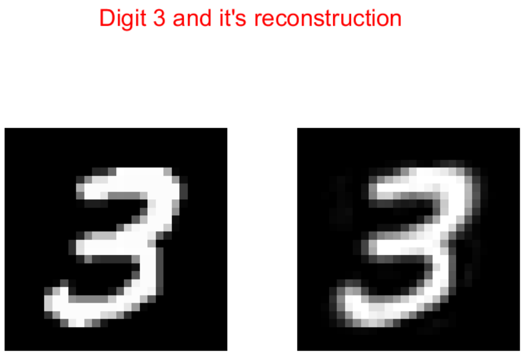
- The covariance matrix generated a 784×784 eigenvector matrix E, which had it's eigenvectors sorted in descending order.

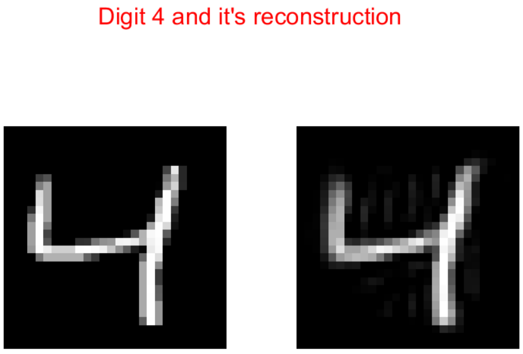
- It was then truncated, and normalized column-wise to yield the 784×84 eigenvector matrix $\widetilde{E}$.
- Let the **column vector** of our image be $\mathbf{v} \in \mathbb{R}^{784}$.
- $\mathbf{v}^T \cdot \widetilde{E}$ yields our coordinate vector, while $(\mathbf{v}^T \cdot \widetilde{E} \cdot \widetilde{E}^T)^T \in \mathbb{R}^{784\times1}$ yields the **column** vector of our reconstructed image.

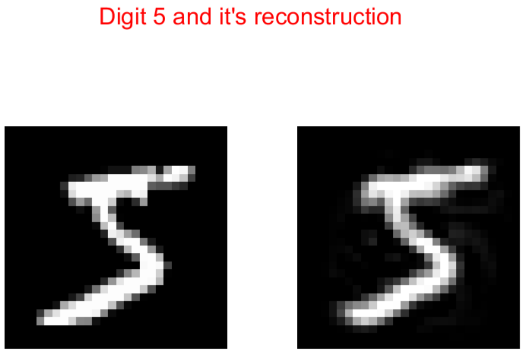
7.4 Results and Analysis

The above algorithms were implemented almost verbatim in our code, and they yielded the following reconstructions shown below (only a few representative images are shown, find the rest in our submissions folder):









One observes that an excellent reconstruction is obtained in all cases, which is only to be expected since the eigenvalues which are not in the top 84 are really small.
On close observation, one notes that the reconstructed images are slightly "blurrier" than the original ones: This naturally lends to the hypothesis that the large mass of ≈ 0 eigenvalues and eigenvectors don't play any roles in determining the structure of the digit itself, but rather give the image an almost imperceptible "micro-detail".
The role of the covariance matrix in smashing together the entire dataset to produce the

eigenvectors can be observed in the following way: If ones looks *very closely* at the digits shown above, one sees very faint white blobs disconnected from the larger digit. Those are the "remnants" produced from the eigenvectors which perhaps were generated from digits which had a bright presence in those areas (which the current digit doesn't have).

8 Problem 6

8.1 Introduction

This problem is quite similar to problems 4 and 5, and thus much of the theoretical background and implementation is similar. We shall thus just walk through the code and then present the results and analyse them.

8.2 Code Walkthrough

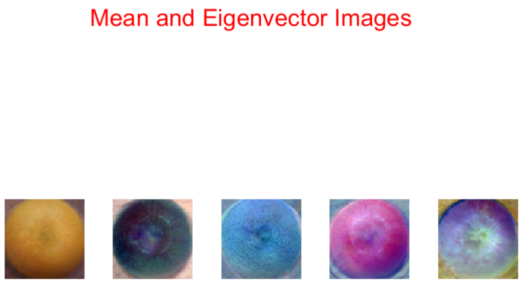
For evaluating the mean μ of the images, we simply reshape them into 19200×1 vectors as instructed in the question and take the mean of those 16 vectors.
The covariance matrix C is found similarly: By mean centering all the 16 vectors, stacking them to form a 19200×16 matrix, multiplying it by it's transpose and dividing the product by 16 to yield a 19200×19200 sized covariance matrix.
The eigenvalues and eigenvectors of C are evaluated using the `eigs` function in MATLAB: Any other method, such as `eig` or `svd` takes prohibitively large amounts of time owing to the large size of our covariance matrix.
Once the eigenvalues are obtained, they are plotted similarly to as in Q4, results of which are attached in the Results section.
Plotting the mean vectors and the top 4 principal eigenvectors is similarly a routine job using the `imshow` and `reshape` functions. One thing which has to be taken care of though, is the fact that since we're plotting RGB images, all entries in our image matrix should be in $[0, 1]$. This basically means that from every element in our image vector we subtract the minimum element of the vector, and then the vector is divided by `max(v) - min(v)`, so that every member of our normalized vector now belongs to $[0, 1]$. Fortunately, this entire normalization process can be carried out in a single step using the MATLAB function `rescale`.
Coming to the **closest approximation of the image using the 4 eigenvectors**, this is something that has already been explained at length in Q5 too: **Remember that the orthogonal projection of a vector v onto a set $\mathcal{B} \subseteq \mathbb{R}^n$ is the element in $\mathcal{B}$ closest to v, measured in terms of the Euclidean norm**. We have also seen that the orthogonal projection of the same vector $\mathbf{v}$ onto the subspace spanned by the 4 eigenvectors can be easily obtained using the truncated expression of it's linear combination in terms of all the eigenvectors.
Thus, our algorithm for finding out the **closest** representation remains the same: Let $\widetilde{E}$ be the 19200×4 eigenvector matrix. Then, as instructed in the question, we first find the orthogonal projection of the centred vector $\mathbf{v} - \mu$, and then add μ back again to restore the "mean". Hence our closest representation, expressed mathematically is $\widetilde{\mathbf{v}} := \mu + ((\mathbf{v} - \mu)^T \cdot \widetilde{E} \cdot \widetilde{E}^T)^T = \mu + \widetilde{E} \cdot \widetilde{E}^T(\mathbf{v} - \mu)$ (the transposes, as explained earlier, are to ensure the sanctity of matrix multiplication. The algorithm and it's logic has remained the same throughout Q5 and 6).

Also, clearly, the reconstructed images will bear some errors vis-a-vis the original ones (in fact we can exactly calculate the error borne: since $\widetilde{\mathbf{v}} = \sum_{k=1}^{4} < \mathbf{v}, \mathbf{u_k} > \mathbf{u_k}$ is the reconstructed image, while the original one was $\mathbf{v} = \sum_{k=1}^{19200} < \mathbf{v}, \mathbf{u_k} > \mathbf{u_k}$, the `error_in_reconstruction` $= \left\| \sum_{k=5}^{19200} < \mathbf{v}, \mathbf{u_k} > \mathbf{u_k} \right\|_{\text{Fro}} = \sqrt{\sum_{k=5}^{19200} < \mathbf{v}, \mathbf{u_k} >^2}$). Thus, for each reconstruction, we have also calculated the Frobenius norm of the difference between the image and it's reconstruction, divided it by the norm of the original image and presented the percentage error on top of every image for the grader's perusal.
Finally, coming to the "sampling" of random fruits to "artificially" generate new fruit images, that was accomplished as follows: We first generate the 4 coordinates ("$< \mathbf{v}, \mathbf{u_k} >, 1 \le k \le 4$") that any of our dataset images (centred) make with the 4 principal eigenvectors. We then take the 1×4 row vector $\mathbf{r}$ of the coordinates and apply to it a **Gaussian distortion of standard deviation $\sqrt{\lambda_k}$, for each k in the row vector**. What does that do? $\forall x \in \mathbf{r}$, x **+=** η, where η is sampled independently for each element from $\mathcal{N}(0, \sqrt{\lambda_k}^2)$. Note that the **standard deviation of each element in the distorted row vector has been kept equal to the the square root of the corresponding eigenvalue**, which allows for fair sampling as eigenvectors corresponding to larger eigenvalues can allow for more variance, and similarly eigenvectors with smaller eigenvalues will have a more concentrated component, after all the eigenvalues along that eigenvector are small precisely because the variation in that direction was lesser, according to PCA theory.
Thus, for any row vector of image coordinates, we are able to generate a closely resembling but not identical set of distorted coordinates $\widetilde{r}$. These coordinates are then fed further into the food-chain to generate an artificial image $\mathbf{v}'$ as $\mu + (\widetilde{\mathbf{r}} \cdot \widetilde{E^T})^T = \mu + E \cdot \widetilde{\mathbf{r}}^T$.
The 3 artificially generated images are also presented in the results section. They have an excellent resemblance to the fruits in the dataset, yet are different from any of them.

8.3 Results

The results and their interpretations (if necessary) are given below.

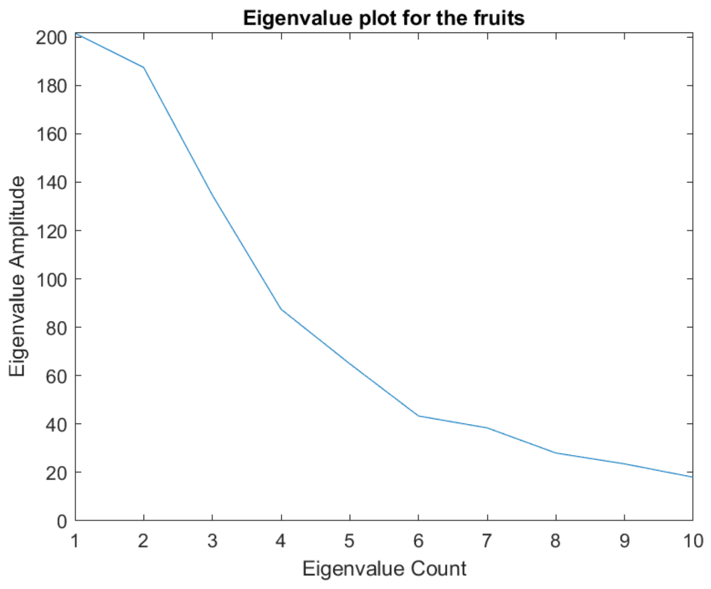
8.3.1 Mean and Eigenvector Images



Note that the leftmost image is the mean vector, rest are the four principal eigenvectors.

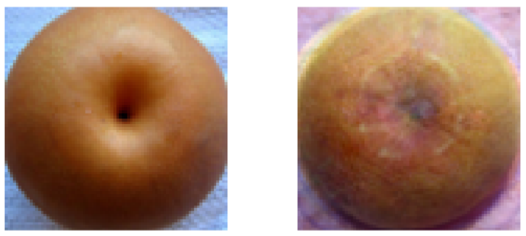
We observe that all of the principal eigenvectors look like round fruits, in accordance with the dataset they were fed on. In fact, one sees that **the eigenvectors' majorly differ in their colour: This tells us that the shape of fruits in our dataset is mostly the same, it's their colour which shows maximum variation, with red apples, orange oranges and green guavas.**

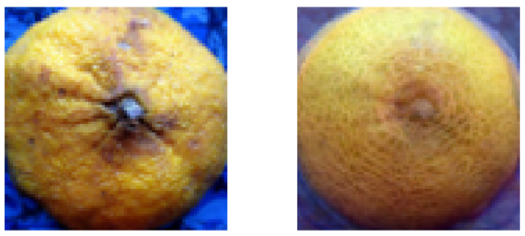
8.3.2 Eigenvalue Plot (Descending Order, Top 10)

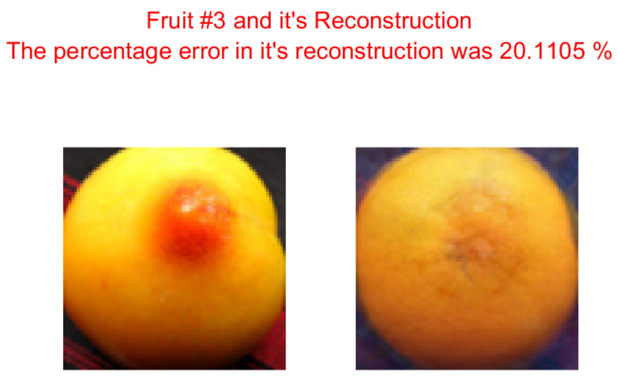


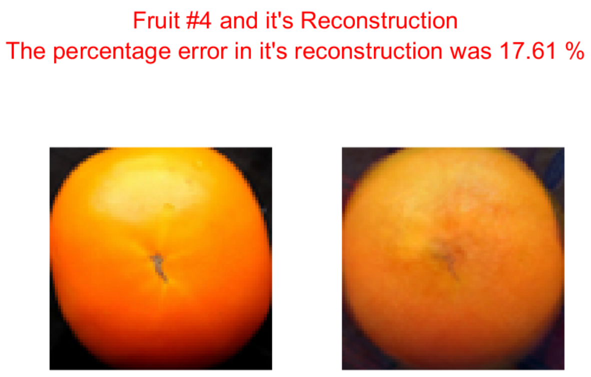
8.3.3 Fruits and their Reconstruction

Only a few representative images are attached here, find the rest in our results folder.


Fruit #1 and it's Reconstruction
The percentage error in it's reconstruction was 23.3197 %

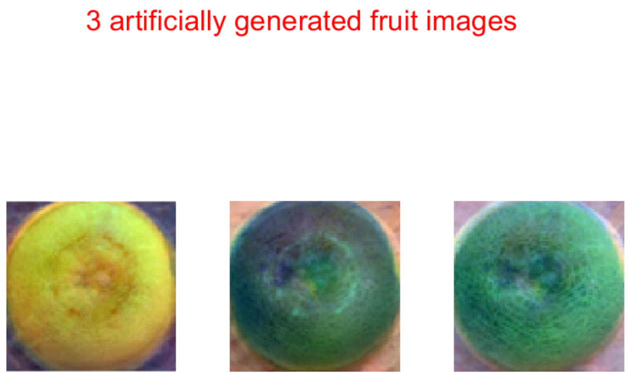

Fruit #2 and it's Reconstruction
The percentage error in it's reconstruction was 24.4393 %


Fruit #3 and it's Reconstruction
The percentage error in it's reconstruction was 20.1105 %


Fruit #4 and it's Reconstruction
The percentage error in it's reconstruction was 17.61 %

One sees that due to the fact that 16 very different types of fruits were used to "train" our PCA model, the reconstructed image kind of "homogenises" quite distinct looking fruits into a generic mean structure it has learnt. One observes that since a majority of the fruits had their top views photographed, with the "cavity" of the fruit stalks figuring prominently in the centre of the image, the eigenvector reconstructions insert a stalk cavity even when there is none: case in point, Fruit 3. This also alerts us to the danger of using a very heterogeneous dataset, as in this case.

8.3.4 Artificially Generated Fruit Images



Note how close a resemblance these 3 fruits bear to actual fruits, although their colouration seems a bit off. This thus demonstrates the power of the PCA technique in successfully generating convincing samples from pre-existing data. This is perhaps the linear algebra, pre-stochastic analogue of **GANs**, which achieve the same through deep learning.