EA-1 Lab Project

This Lab Project serves as an introduction to Image Processing problems and describes how Linear Algebra concepts can be employed for their solution.

1 Introduction to Image Processing

Since computers only understand and operate with sequences of 0's and 1's, images are stored in their memory or hard-drive as arrays or matrices of numbers. What we perceive in the physical world as colors is no more than values which encode light intensity in the digital world. Since Linear Algebra is based on matrix and vector operations it consitutes one of the most powerful tools to process and analyze images. In this Lab Project we will explore some of its capabilities.

As mentioned above, images are represented as numbers. For black-and-white images (or grayscale images) the value of each pixel is most commonly represented in the computer by 8 bits of information (0's or 1's). As such, there is 2^8 combinations of sequences or else integer values in the range of $0-255$ can be represented. The value 0 corresponds to the absolute black color, the value 255 to the absolute white color, while values in between represent different shades of gray.

Obviously, for representing a color image we would need more bits to encode the color. This is indeed the case. Each color image is represented by a set of triplets of 8 bits, each one corresponding to a value of Red, Green or Blue colors. In other words, a color image of size $m \times n$ can be represented as a $3D$ matrix of size $m \times n \times 3$. Each one of the *depth* layers of the matrix corresponds to the intensity of a different color in the scene and the linear combination of their values is perceived by the human eye as a different color. The idea of this concept is depicted in Figure 1.

As you probably already know, MATLAB is a very useful tool for Linear Algebra problems. As such it is a very powerful environment for creating and testing Image Processing algorithms and applications.

As an introduction, we will list a set of basic MATLAB built-in functions that are useful for reading/writing and processing images.

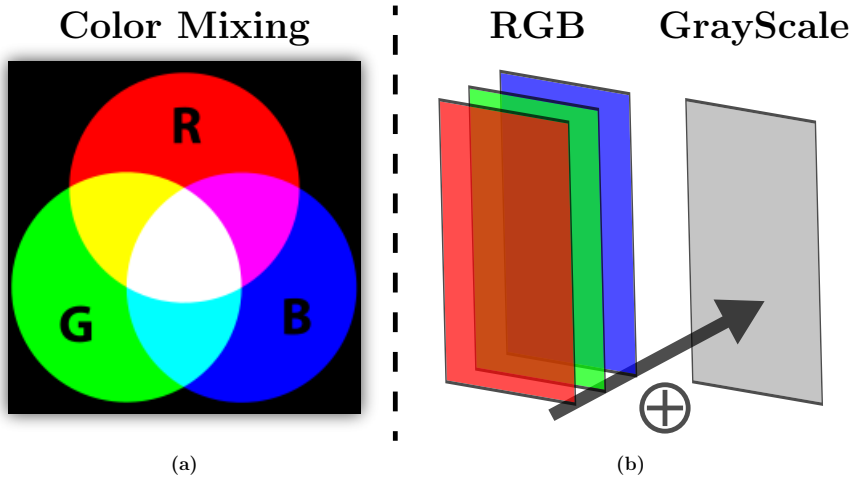

Figure 1: (a) Illustration of the Color Mixing concept in the RGB Color Space; (b) Illustration of transforming an RGB image (3 Channels) to Grayscale (1 Channel). In MATLAB this can be achieved using the function `rgb2gray()`.

- `imread(filename)`: Reads an image with name `filename` from the hard-drive. **Note**: `filename` must be a string that contains the full-path (filename) of the image in your computer. If your current working directory in MATLAB contains this image, then the format `name.ext` is sufficient. Here, `ext` denotes the extension of the image or the method that has been used in order to compress it so that it consumes as little memory as possible on the hard-drive. The are multiple image extensions with many different characteristics (e.g., `.bmp`, `.jpg`, `.tif`, `.png`). In this Lab Project we will use `.png` images which are rather low-quality and commonly used for web-applications. The `imread()` function reads an image in *unsigned 8-bit integer* format or `uint8`. As discussed above, these are numbers represented by 8 bits, hence obtaining integer values between 0 and 255.

- `rgb2gray(img)`: Converts the matrix stored in the variable `img` to a grayscale image, i.e., accepts a $3D$ matrix and returns a $2D$ matrix of values that correspond to different shades of gray. Each value in the grayscale image is determined as a linear combination of the values in the three channels of the image, (Red, Green and Blue), as seen in Figure 1. The coefficients of this linear combination are predefined and decided such that the perceived final shades of gray are properly balanced. In MATLAB the order of color Channels is Red - Green - Blue. Therefore, if you have a $3D$ matrix of size $m \times n \times 3$ the three Channels can be obtained as: `x(:,:,1)`, `x(:,:,2)`, `x(:,:,3)`, respectively.

- `imshow(img)`: Shows the matrix stored in the variable `img` as an image in the current Figure Window. **Note**: As explained above, for the image to be shown properly it needs to be represented by values between 0 and 255. So, unless your data is already in `uint8` format (you can see that in the MATLAB workspace) consider rescaling the values of the image to the range $[0 - 255]$ and apply the `uint8()` function on it to convert it to `uint8` format before showing it. `imshow()` also works with images that have values between 0 - 1. In either case, if values end up being outside of these limits they are going to be truncated to this maximum/minimum values.

  You can test the `imread()` and `imshow()` functions using the commands:

  ```
  x = readImage('cameraman.tif');
  figure, imshow(x);
  ```

  This should open up a new figure window and display the "Cameraman" image which is stored internally in MATLAB and commonly used in Image Processing research area.

- `im2double(img)`: Converts a `uint8` matrix (or image) `img` into a matrix with `double` values or values of double precision, meaning that they can represent any real number, up to the computer's precision ability. At the same time, this command rescales the values in the image in the range between 0 and 1. The double precision as well as rescaling makes the image ready for processing since most built-in MATLAB commands require double precision variables. At the same time, double precision limits the issue of round-off errors when one performs calculations.

  Now test the command: `im2double(x)`. Note the change in variable type from `uint8` to `double` in the MATLAB Workspace.

- `imwrite(img,filename)`: Saves the matrix stored in the variable `img` as an image in the file-path defined by the sting `filename`. The image should first be rescaled to values between 0 and 255 and transformed into `uint8` format before saving.

Now test the commands:

```
y = uint8(255*x);
imwrite(y, 'myimage.png');
```

This should save the image in your current MATLAB directory and assign it the name `myimage.png`. You can view this image outside of MATLAB by looking in the folder that has been saved and opening it with any Image Viewing application you might have installed on your computer.

As you realize, MATLAB is not only a dull working environment for solving Linear Systems of Equations but rather a platform that could allow you to manipulate and process your images just like Photoshop does.

2 Problem Description

In this Lab Project, you will implement a set of MATLAB functions in order to perform a very simple version of Face Recognition. As the face database, we will use the faces of 100 NBA players.

The main idea behind Face Recognition is finding the face image that looks most similar to the query image. Assume that we have access to a database of labeled images, i.e., images that depict people whose identity is known and these images have been appropriately labeled. Now, if one sends a query face image for recognition, the most straightforward way to recognize this face is to look through the images in the database and find the one which is most similar to our query image. Since the images in the database are labeled, finding the image with the most similar appearance implies that the identity of the person in the image can be found, assuming that the face of this person exists in the database. Obviously, the problem is not that simple and there is ongoing research on the topic for images that have been corrupted by noise, occlusions, facial expressions, lighting variations, etc. However, in this Lab Project we want to introduce the main concepts behind the topic and we will present them in a simple and straightforward way.

Assume that we have access to N RGB images (color images) of players. Also, assume that the size of each one of the R, G and B Channels in each image is $m \times n = M$. One can read these images, convert them to grayscale, vectorize them and store them in the columns of a matrix (database) D of size $M \times N$, i.e., each column contains an image of $m \times n = M$ pixels and there is N such columns. Furthermore, assume that we also have access to a Labeling vector $\mathbf{p}$ of size $N \times 1$ that holds the name of the k-th player (Name-k) at location k, where $k = 1 \dots N$.

Since we know the (Index - Player Name) correspondence in vector $\mathbf{p}$, the easiest way to perform Face Recognition would be to reorder the columns of the database D such that the image of the k-th player is at the k-th column of the matrix D. Then, by sending a query image, we can find the column of the matrix D that holds an image which is most similar to our query image and then identify the player as player-k with name Name-k.

2.1 How do we find images that look similar?

Similarity between images can be defined using specific cost functions that measure the difference (*distance*) between two images. Assuming that we have two vectorized images in vectors $\mathbf{x}$ and $\mathbf{y}$ which are both of size $M \times 1$, the most commonly used metrics that measure their difference are the Mean Squared Error (MSE) and the Peak Signal-to-Noise Ratio (PSNR) which are directly related to each other. Their definitions are,

$$\text{MSE} = \frac{1}{M}\sum_{i=1}^{M}(\mathbf{x}_i - \mathbf{y}_i)^2\,, \tag{1}$$

$$\text{PSNR} = 10\log_{10}\left(\frac{\text{MAX}_{\mathbf{x}}^2}{\text{MSE}}\right), \tag{2}$$

where $\text{MAX}_{\mathbf{x}}$ is the maximum possible value for an element of the vector $\mathbf{x}$ or $\mathbf{y}$. In Image Processing, this value is usually 255 when the images are of type `uint8` or 1 when the images are of type `double` and normalized. (read Section 1).

For the MSE metric, lower values mean higher similarity while for the PSNR metric, higher values mean higher similarity. Therefore, if one finds an image at the k-th column of the database D that gives the minimum MSE or, equivalently, the maximum PSNR, when compared to a query image, this means that the image that resides on this k-th column of D is the one with the closest *distance* to our query image.

2.2 Goal of this Project

The goal of this Lab Project is to create an algorithm that reorders the columns of a database of images, as described above, so that Face Recognition would take place. We provide you with the following:

- A set of RGB images for all the players. (in the folder `Player_Images`).
- Function `createImageDatabase(imagePath)` which reads all the RGB images in the folder `imagePath`, converts them to grayscale, vectorizes them and places them randomly in the columns of the database matrix D.
- Script `LabProject.m` which contains the main script you would need to run for this Lab Project. Except for selecting the player number, you **should not** make any other changes to this script. The script has been designed to run up to different sections once you have written the appropriate functions, as described in Section 3.
- The definition of the function `unScrambleDatabase.m` is provided. However, you will need to implement the code for this function.
- Finally, the function `identifyPlayer()` is provided. This function holds the labeling vector $\mathbf{p}$ and shows the results presented in Figures 4 and 5, using the appropriate input arguments.

Note: Functions `createImageDatabase()` and `identifyPlayers()` are provided in `.p` file format. This format is essentially the same as a MATLAB `.m` file but cannot be accessed for editing or viewing.

The main steps of the algorithm are:

1. Construct the Image Database D.
2. Unscramble the Database by reoordering its columns. The image of the player with filename `player1.png` should end up on the 1-st column of the database, the image of the player with filename `player2.png` should end up on the 2-nd column of the database, etc.
3. Verify that the database has been properly reordered. To do this you use the same unscrambling function as above but providing the result of the previous unscrambling as an input. If the columns of the database do not change and they are sorted (from image of the player with filename `player1.png` to the image of the player with filename `player100.png`), this means that the reordering was successful. The reordering procedure can take some time to finish. However, once your function is implemented correctly and the reordering is properly done, the script will automatically store your correctly reordered matrix and will not perform reordering the next time you try to run your code.
4. Plot the order of the indices before and after reordering.
5. Read the image of player-k (your selected image). This k should correspond to the number in the file that holds your selected image, e.g., if the filename of the selected image in the folder `Player_Images` has filename `Player34.png` then k, or else the variable `PlayerNumber` at the `Initialization` section of the `LabProject.m` script should be selected as 34.
6. Calculate the column in both databases (before and after reordering) that is mostly similar to your picture.
7. Calculate the PSNR between your picture and all images in both databases (before and after reordering).
8. Pass these results as input arguments to the function `identifyPlayer()` using, first, the randomly ordered database and then the properly reordered database in order to see the wrong and correct identification of your image, respectively.

These main skeleton of the program that performs these steps has already been implemented for you in the main script `LabProject.m`. You mainly need to follow the steps of Section 2.3 and the instructions in Section 3 in order to run the script properly. The steps above and the main logic of the algorithm are summarized in Figure 2.

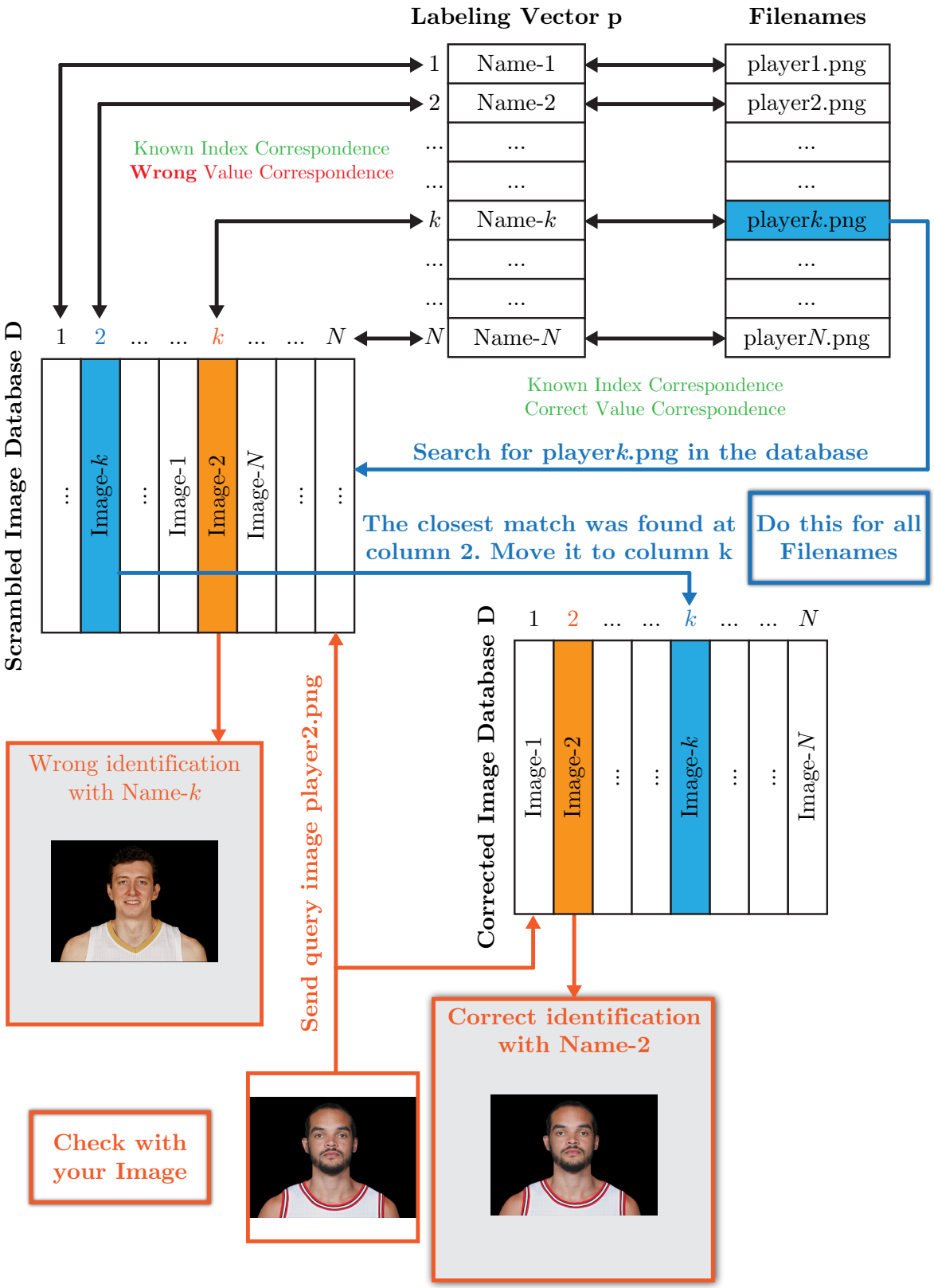

Figure 2: Illustration of the logic of the Face Recognition algorithm.

2.3 To Do Items

1. Look through the images located under the folder `Player_Images` and locate the image that you would like to choose. This is the image that you will be working with. Assign this number to the variable `PlayerNumber` at the `Initialization` section of the provided script `LabProject.m`.

2. Try to run the provided script `LabProject.m`. You will notice that you get a set of `warnings`. This is due to the fact that some functions are missing in order for the code to run properly.

3. Implement the functions discussed in Section 3 one-by-one and test the script `LabProject.m` until all the warnings dissappear. Your whole script should only run completely (warning/error-free) once you have implemented all the functions correctly.

4. Once you have finished writing all the functions test the whole script `LabProject.m`. You should get 3 figures as an output:

   (a) The first one should present the indices of the images in the database before and after correction (scrambled - corrected), as the example presented in Figure 3.

   (b) The second one should show a wrong recognition of your image due to the use of the scrambled database, as the example presented in Figure 4.

   (c) The third one should show a correct recognition of your image due to the use of the corrected database, as the example presented in Figure 5.

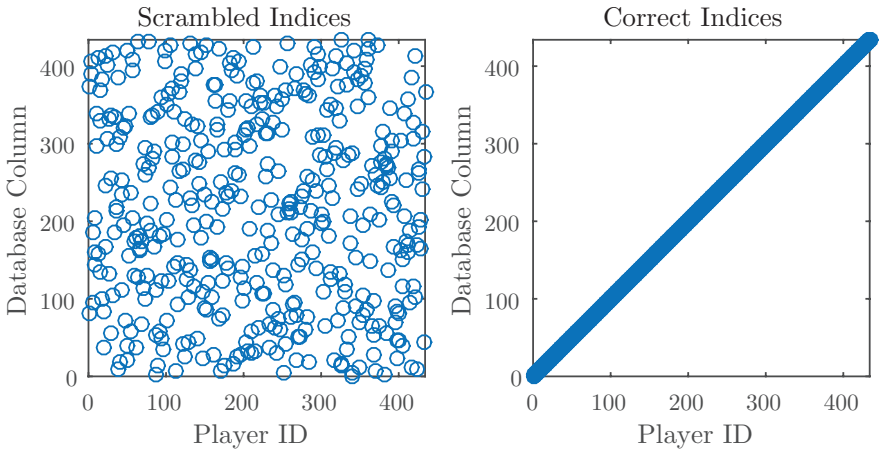

Figure 3: Example of database indices before and after correction.

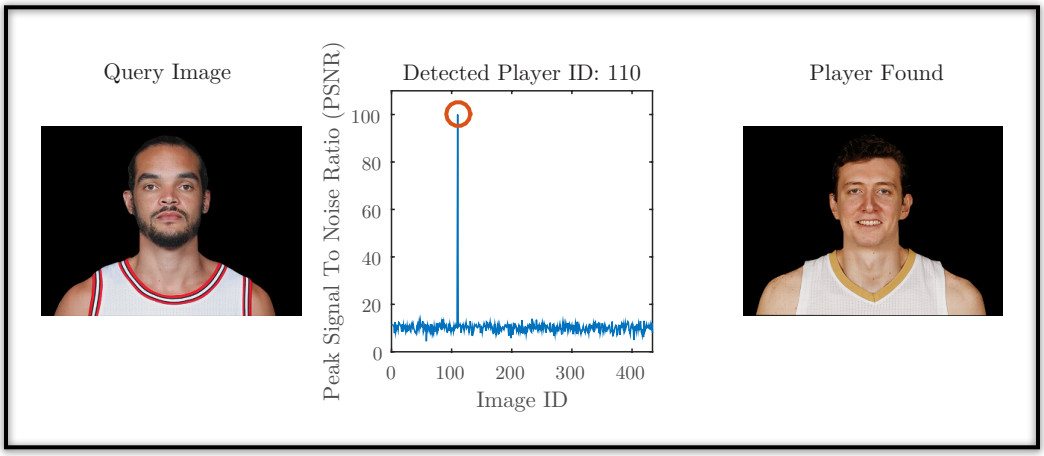

Figure 4: Example of erroneous detection using the scrambled Image Database.

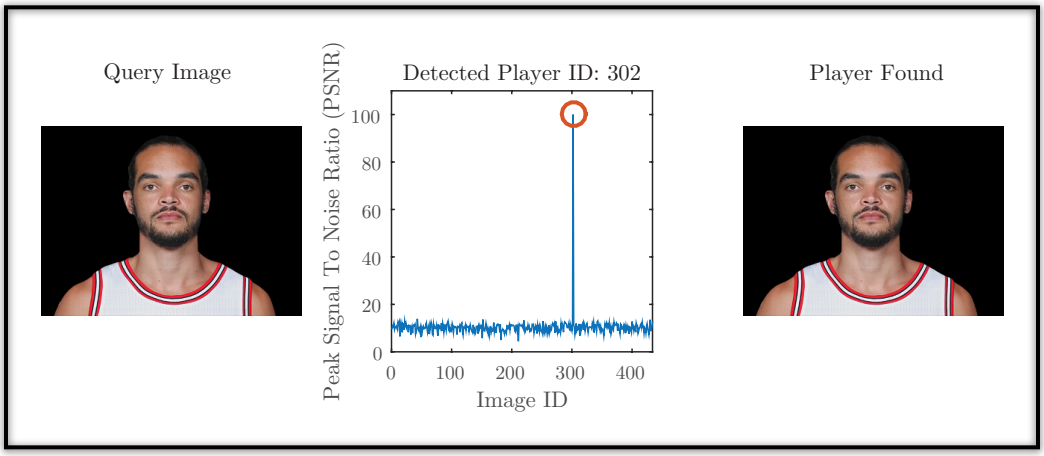

Figure 5: Example of correct detection after correcting the Image Database.

3 Set of Functions to be Implemented

You will create a set of helper functions that will be useful.

- `function vecOut = makeVector(matrixIn)`

  This function accepts a matrix (`matrixIn`) as an input and returns the vectorized version of this matrix (`vecOut`) as an output (following the standard column-ordered MATLAB rule). In this function, you should:

  1. Check that the input is of numeric type (using MATLAB's built-in function `isnumeric()`), otherwise report a relevant `error` message.
  2. Check the dimensions of the input matrix (use MATLAB's built-in function `ndims()`). If it has more than two dimensions the function should report a relevant `error` message.
  3. Vectorize the input matrix and assign the vector to the output variable `vecOut`.

- `function MSE = calcMSE(x1,x2)`

This function accepts two column vectors (`x1`,`x2`) as an input, calculates the *Mean Squared Error* (MSE) between them (according to equation (1)) and assigns the result to the output variable `MSE`. No error checking is necessary. In order to make sure that the inputs are vectors force them to become vectors at the beginning of the function using the function `makeVector()` you created above.

- `function PSNR = calcPSNR(x1,x2,maxX)`

  This function accepts two column vectors (`x1`,`x2`) and a maximum value for their elements as an input and calculates the *Peak Signal-to-Noise Ratio* (PSNR) between them (according to equation (2)). Then it assigns the calculated value to the output variable `PSNR`.

  In this function, you should:

  1. Check if the number of input variables is less than 3. If so, assign `maxX` to get its default value 1.
  2. Calculate the PSNR based on equation (2) and using the function `calcMSE()`.
  3. If the MSE is calculated as 0 set the PSNR to get the value 100, otherwise the division by 0 would cause the PSNR to be calculated as Infinity.

- `function PSNRs = computePSNRs(imgVec, imageDatabase)`

  This function accepts a vectorized image (`imgVec`) and a database (matrix) of vectorized images (`imageDatabase`) as an input and computes the PSNR between `imgVec` and each image in the database. The calculated PSNR for each image in the database should be stored in an array of appropriate size and assigned to the output variable `PSNRs`.

- `function minPos = findMinimumErrorPosition(imgVec, imageDatabase)`

  This function accepts a vectorized image (`imgVec`) and a database (matrix) of vectorized images (`imageDatabase`) as an input and finds the position (column) in the database that produces the smallest error when compared to the input vector `imgVec`. The result (column number) is assigned to the output `minPos`. In this function, you should:

  1. Calculate the MSE between `imgVec` and each image in the `imageDatabase`.
  2. Find the position (column) where the MSE is minimum. You can use the MATLAB's built-in functions `min()` and `find()`.

- `function image = readImage(imgName)`

  This function accepts the *string* `imgName` as an input and reads the image from the hard-drive. If the image has color (3 Channels - RGB) it converts it to grayscale and normalizes it. The resulting image is assigned to the output variable `image`. **Note**: The variable `imgName` must be the full-path of the image in your hard-drive. Follow the notation at the `Initialization` part of the provided script `LabProject.m` to properly construct the path. In this function, you should:

  1. Read the image using the MATLAB's built-in function `imread()`.
  2. If the dimensions of the image are more than 2, the image should be converted to grayscale using the MATLAB's built-in function `rgb2gray()`.
  3. Convert the image from `uint8` type to `double` type and normalize its intensity in the range $(0, 1)$ for processing purposes. This can be achieved using the MATLAB's built-in function `im2double()`.

Before proceeding, test your `readImage` function using the commands:

```
x = readImage('cameraman.tif');
figure, imshow(x);
```

This should open up a new Figure Window which will display the "Cameraman" image which is stored internally in MATLAB.

- `function [newDatabase, indices] = unScrambleDatabase(imagePath, database)`

  This function should read all the images from the folder `Player_Images` and search for their position (column) in the scrambled database. Then it should unscramble the database by reordering its columns according to the results of this search. The function accepts as inputs the `imagePath` or folder where the player images are stored as well as the `database` (matrix with vectorized images) that the user wants to unscramble. In this function, you should:

  1. Initialize the variable `indices` to be an array of 0's of proper dimensions.
  2. Read all the images in the path `imagePath` using the function `readImage()` you created above.
  3. Find the minimum error position for this image in the database using the function `findMinimumErrorPosition()` you created above.
  4. Store all the minimum error positions, for each tested image, in the array `indices`.
  5. After finding the correct indices, reorder the columns in the database (shuffle the images around) so that the new ordering is according to the ordering of the stored names of the players, as it was explained earlier in this document.
  6. Finally, assign the resulting reordered database to the output variable `newDatabase`.

- `function plotIndices(scrambledIndices, correctIndices)`

  This function accepts the initial `scrambledIndices` and the final `correctIndices` as inputs and creates a figure with two plots for comparison. The result should look similar to the one presented in Figure 3. **Note**: Every time you run the program the database is created using a random order for the images so the result of the scrambled indices is expected to look different at different tries or compared to the one presented in Figure 3. The corrected indices, however, should always represent a line passing through the origin, meaning that the name database has a one-to-one correspondance with the image database (i.e., Name1 $\rightarrow$ Image1, Name2 $\rightarrow$ Image2, Name3 $\rightarrow$ Image3, ...). In this function you should:

  1. Create a new figure using MATLAB's built-in function `figure()`.
  2. Use *Marker Type* `'o'` when plotting.
  3. Make the `axis` of the figure `square`.
  4. Limit both `axis` in the range of [0 – number of indices].
  5. Use appropriate `title` as well as `xlabel` and `ylabel`.
  6. Use MATLAB's built-in function `subplot()` to put both plots in the same figure.
  7. Make sure that your resulting figure looks as similar as possible to the one presented in Figure 3.

4 Lab Project - Submission Instructions

Once you finished the implementation, you will need to submit the Lab Project. **Note**: If you are not completely done, submit what you have, **BUT make sure that the submitted files are able to run without reporting any errors**.

Your submission should include the following:

1. A zipped (compressed) folder containing all the files that are required to run the Lab Project.m. You may only omit the `Player_Images` folder that we provided you with.

2. A published version of your code in html format. In order to produce this, go to the "PUBLISH" tab of MATLAB and click on the "Publish" button. Let the code run and produce all the results (this might take some time). Now, you should have a folder named "html" under your MATLAB working directory. Rename this folder as "html_your_name", zip (compress) it and submit this compressed folder as well.