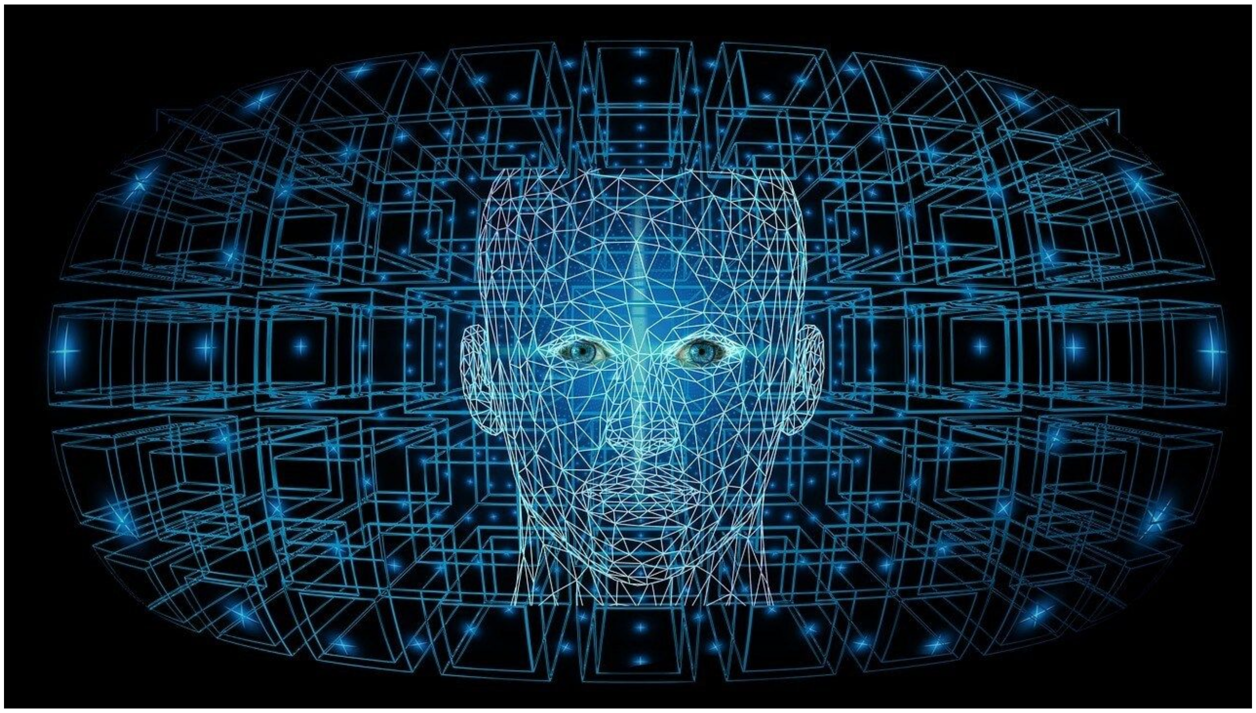


Image Processing

Final Project

Signals & systems

Sohrab Namazinia
97522085

Overview

This is an image processing project coded in Matlab as my signals and systems final project. This documentation consists of all required explanations about the project.

The project consists of different parts. I have mentioned each of them here with its own explanation. The main parts of the project are:

- **Reading/Showing image (storing matrix of image)**
- **store/representation of images (like RGB)**
- **Convert Matrix to image**
- **Image storing methods (RGB) research**
- **Image edge recognition**
- **Image edge recognition algorithms**
- **Adding noise to images**
- **Image noising research**
- **Denoising images**
- **Image denoising research**
- **Noise Measurement**
- **Summary**

- At the end of each section, I have written the name of the matlab code files related to that section.
- The topics are colored Orange.
- Also Keywords are highlighted as **bold.**
- At the end of each part, there is a code section, explaining the code files related to - that section, which is colored **green**.
- Important notes (reminders, etc) are colored **red**.
- I have used 'Mathworks.com' to get information about the details of methods for edge detection and noising images.
- Although all the code has been explained in this documentation, there are also comments on the code for better intuition.
- It is good to mention that like all my other projects, I have put a lot of time and effort on this one too.
- To be honest, it was one of my most exciting projects!

Reading/Showing Image (storing matrix of image)

Images can be easily read and converted to RGB in matlab. First I get the user image path from the command line. Then if there is no such file, I message the user that the path is invalid. Else, I read the image with **imread** command. It stores the image as a **3-D Matrix**, each combination of row and column, specifying RGB of a pixel color.

The image is shown to the user by **imshow** command**.**

But this is the simplest way, I researched about storing and showing images in other ways too.

But another task that my code does is to write the image matrix into a text file in **BMP format** in the same directory named in a file named “output.txt”. We will use that as input to show images later.

BMP: a bitmap is an array of binary data representing the values of pixels in an image or display

Code:

ReadShowImage.m : takes image path from input, then shows it. All my images are in the ‘images’ folder in the root directory. If you want to test your own image, type the full path in the command line after running the file.

ShowImage.m : shows a picture and displays 3-D color gamut as point cloud in RGB color space. After running it, you can see a black-white picture and a colorful one. Their color cloud RGB shows this truth too.

IMPORTANT: up to now, our input from the user is path. After passing the next part (Image Storing as Matrix), i’ll explain that there is another matlab file that takes the BMP as input, and then shows the image.

Storing/Representation of Images

1) Image **2D-Array**: we store an image as an array of colors, each element representing the color of a pixel. The values are compared based on **colormap**.

2) Image **3D-Array**: in this approach, we use true colors. It is like a RGB based method which is scaled to one.

Code :

StoreImage.m: i have coded examples of both types in this file.

Convert Matrix to Image

Now it is time to read the **Output.txt** and convert it to image to show that. Output.txt is the BMP of the last image that is requested to be shown. The total process is like this:

Text file (bmp format) => 3-D matrix => Image

Code:

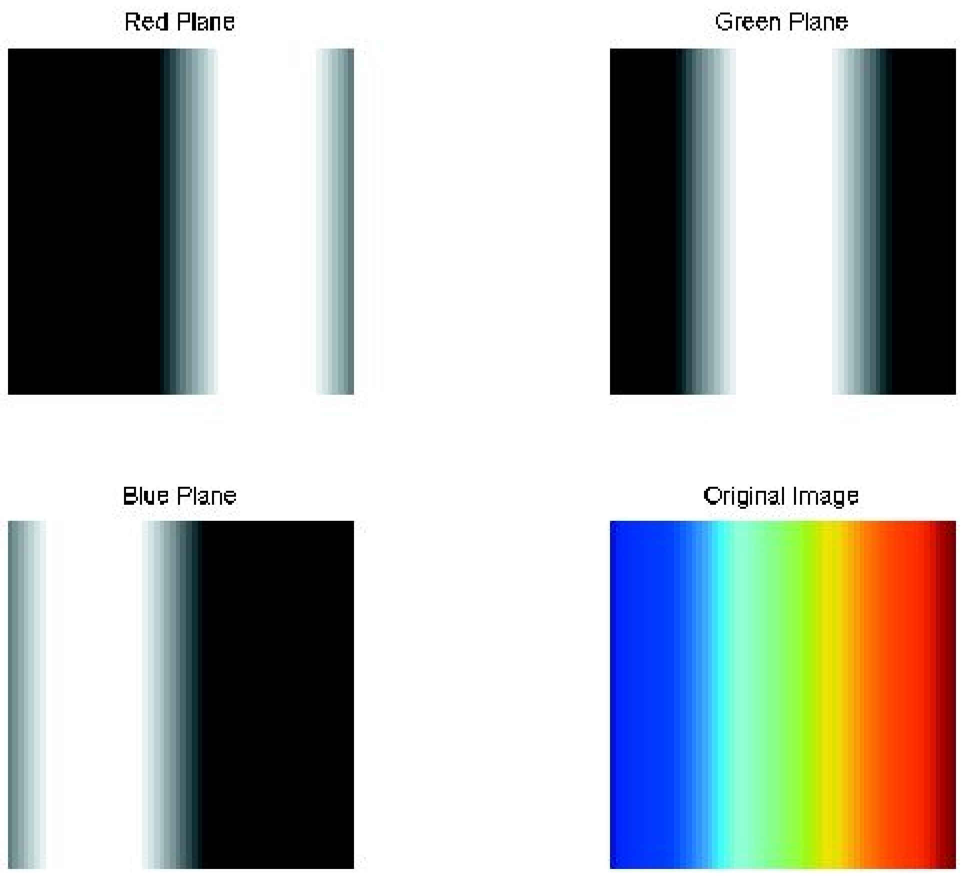
MatrixToImage.m: reads data from Output.txt, shows the corresponding image.

Storing image methods (RGB) research

I studied about this topic on the internet, and tried to summarize that here.

There are lots of methods of storing and representing images, here i have mentioned some of them.

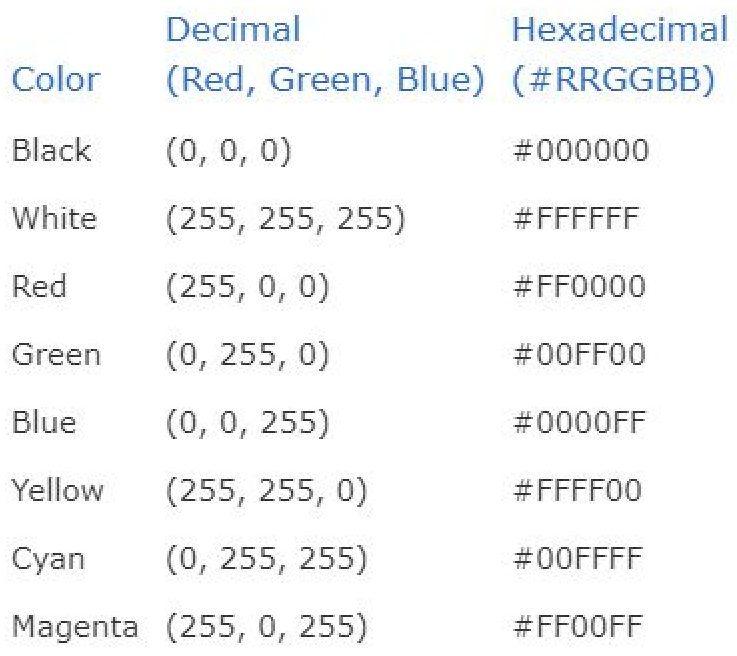
1) **RGB** : RGB is a tuple, consisting of three elements, that correspond to red, green, and blue values of the image. Each of them can be from zero to 255. For example black is (0, 0, 0), white (255, 255, 255).



In the above picture, you can see the separated color planes of an RGB image, each containing an area of white. The white regions show the highest concentration of the corresponding color value there.

2) **Hex**: it is another color representation method. It is just a hexadecimal number, representing a specific color. But why does it contain 6 bits? Because each of the three RGB colors can have values from 0 to 255, so 255 * 255 * 255 different colors must be represented. Since hex digit can contain 16 different values (0 to 15), 6 bits are required.

The format **("notation")** used on web pages is **#RRGGBB**, where RR is how much Red (showing by two hexadecimal digits), GG Green, and BB how much Blue.

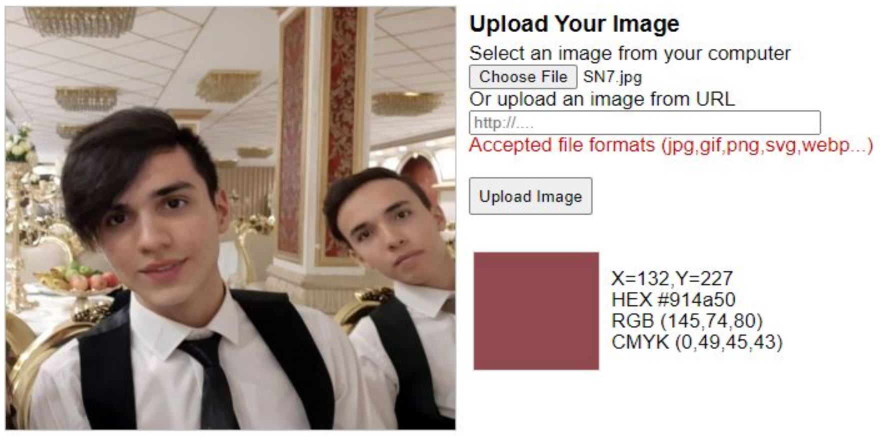


Color	Decimal (Red, Green, Blue)	Hexadecimal (#RRGGBB)
Black	(0, 0, 0)	#000000
White	(255, 255, 255)	#FFFFFF
Red	(255, 0, 0)	#FF0000
Green	(0, 255, 0)	#00FF00
Blue	(0, 0, 255)	#0000FF
Yellow	(255, 255, 0)	#FFFF00
Cyan	(0, 255, 255)	#00FFFF
Magenta	(255, 0, 255)	#FF00FF

In the above picture, you can see the Hex value of some known colors.

3) **CMYK** : just like RGB, but for the colors (Cyan, Magenta, Yellow, Black). It is mostly used in printer devices.

In the picture below, a specific pixel has been chosen to be represented by the above three methods.



Upload Your Image

Select an image from your computer

Choose File SN7.jpg

Or upload an image from URL

http://....

Accepted file formats (jpg,gif,png,svg,webp...)

Upload Image

X=132,Y=227
HEX #914a50
RGB (145,74,80)
CMYK (0,49,45,43)

Image Edge recognition

Finding the edges of pictures, is a famous computer image processing problem. There are some algorithms to do that. Some famous such algorithms are:

1) **Roberts cross**
2) **Sobel**
3) **Prewitt**
4) **Log**
5) **ZeroCross**
6) **Canny**
7) **approxCanny**

IMPORTANT: i'll explain these algorithms in the next section. For now i explain how i implemented them in matlab.

Code:

DetectEdge.m:

It takes three sample photos, reads them, converts them to image matrices. But each image matrix is 3 dimensional. The algorithms for edge detection have 2 dimensional inputs. So that means we need to convert it to 2-D grayscale image by matlab method **rgb2gray**.

Then, we use matlab **edge** method which takes at least two important arguments, one is the 2-D image, the other is the edge detection algorithm that we want to use. It also has other parameters like **threshold**, which ignores all edges that are not stronger than that.

Finally, by matlab **imshowpair** method, i have shown both the original image and the resulting image next to each other for easy comparison.

Three sample images : i tried to use different images from different aspects. One photo that I shot myself, one a city view behind a window (as documentation said), and finally I searched for good pictures for edge detection testing and found a photo from a butterfly!

Important: after running this code, it prints the name of some algorithms, then the user must print the name of one of these algorithms so that the code uses that for edge detection. If the user prints something else, the program will warn you that the name is out of available options.

Image Edge recognition Algorithms

All the mentioned algorithms in the previous part, are used to detect edges in photos. They can also take sum optional parameters to do this job in a specific way. I will explain the term **gradient** after reviewing algorithms.

But now we want to see how each of these algorithms attempts to find edges, one by one:

1) **Roberts**:

   It Finds edges at those points where the gradient of image is maximum, using the Roberts approximation to the derivative

2) **Sobel**:

   Finds edges at those points where the gradient of the image is maximum, using the Sobel approximation to the derivative.

3) **Prewitt**:

   Finds edges at those points where the gradient of the Image is maximum, using the Prewitt approximation to the derivative.

4) **log**:

   This algorithm Finds edges by looking for zero-crossings after filtering Image with a Laplacian of Gaussian filter. This filter is called LoG.

5) **zerocross**:

   It Finds edges by looking for zero-crossings after filtering Image with a filter that the user choses.

6) **canny**:

   Canny Finds edges by looking for the local maxima of the gradient of Image. The edge function calculates the gradient using the derivative of a Gaussian filter. This method uses two thresholds to detect strong and weak edges, including weak edges in the output if they are connected to strong edges. By using two thresholds, the Canny method is less likely than the other methods to be fooled by noise, and more likely to detect true weak edges.

7) **approxcanny**:

   This is a special case of the former one, canny algorithm. It Finds edges using an approximate version of the Canny edge detection algorithm that provides faster execution time at the expense of less precise detection. Floating point images are expected to be normalized to the range [0, 1].

Now time to look at Roberts cross algorithm in more detail to see what we mean by gradient, I searched about it, studied and learned how this algorithm works, the rest are different in the aspects that were specified above.

This algorithm has an operator. the operator consists of a pair of 2×2 convolution kernels. One kernel is simply the other rotated by 90°. Here are the convolution kernels:

$$\begin{array}{|c|c|} \hline +1 & 0 \\ \hline 0 & -1 \\ \hline \end{array} \qquad \begin{array}{|c|c|} \hline 0 & +1 \\ \hline -1 & 0 \\ \hline \end{array}$$

Gx Gy

These kernels are designed to respond maximally to edges running at 45° to the pixel grid, one kernel for each of the two perpendicular orientations. The kernels can be applied separately to the input image, to produce separate measurements of the gradient component in each orientation (call these *Gx* and *Gy*).

These can then be combined together to find the absolute magnitude of the gradient at each point and the orientation of that gradient. The size of the gradient is:

$$|G| = \sqrt{Gx^2 + Gy^2}$$

And for computing an approximate magnitude, we use:

$$|G| = |Gx| + |Gy|$$

The angle of orientation of the edge giving rise to the spatial gradient (relative to the pixel grid orientation) is given by:

Theta = Arctan(Gy / Gx)

The two components of the gradient are conveniently computed and added in a single pass over the image using the operator that is shown below:

P_1	P_2
P_3	P_4

It is called a Pseudo-convolution kernel and is used to quickly approximate gradient magnitude. Using this kernel the approximate magnitude will be:

$$|G| = |P1 - P4| + |P2 - P3|$$

Now to summarize, let’s see roberts cross pros and cons:

Advantages:

1) **It is very fast :** because it checks 4 input pixels to determine the value of each output pixel.
2) **The calculations only include addition/subtraction**.
3) **There are no mandatory parameters to set**.

Disadvantages:

1) **It is very sensitive to noise** : because the size of the kernel is very small.
2) **It ignores the edges that are not very sharp, but the Sobel operator is much better at this point.**

Code:

Roberts_Canny_Difference.m:

I downloaded a good image (Pizza!) for showing the difference between Roberts and Canny algorithms. After running it, you can compare the results.

:

Add noise to images

Noise is a random variation of information about color or brightness in an image. There are lots of methods to add noise to images in matlab. Here i introduce and implement them, in the next part i will explain the methods of nosing images.

There are **gaussian, Poisson, speckle** and **salt & pepper** noise types, all take a grayscale image as input and produce the noisy image. I have implemented all with multiple images.

Code:

NoiseImage.m:

All methods are implemented in matlab via **imnoise** method. I have implemented the following algorithms on my sample images, here are the outputs:

Fig1 => salt & pepper, d = 0.1

Fig2 => speckle

Fig3 => Poisson

Fig4 => gaussian, mean = 0, variance = 0.01

Fig5 => gaussian, mean = 0.5, variance = 0.1

Image noising research

Noising and denoising images is an important topic in the image processing field. In the last section, I implemented some famous types of noises, in the next section I will implement denoising images. In this section, I briefly explain the result of my research about this topic.

Gaussian noise

This type of noise method takes two extra arguments, a **mean** and a **variance.**

Mean: is the mean of gaussian noise as a number. It is default to zero in matlab. This number is added to each pixel, so the resulting image will have higher (in average) image pixel values. Since the maximum pixel value is generally white, the resulting image is closer to white, briefly it causes the image to be brighter.

Variance: it is a number specifying the variance of gaussian noise. Its default value is 0.01 in matlab. Actually the variance of the noise is the power of the noise in an image.

Poisson noise

Its main difference with many other noise algorithms is that it generates noise from the image itself rather than inserting artificial noise.

Salt & pepper

It adds salt and pepper noise with an input variable noise density. Noise density is a number between zero and one. For example if d = 0.03, it means 3% of image pixels have been noisy.

Speckle

It is a multiplicative noise with the following equation where I is input image array, O is out image array and n is a uniformly distributed random noise with mean = 0 and variance = 0.05 (or 5%):

$$O = I + n * I$$

Denoising image

There can be various types of noises on images. Thus another important field in image processing is to recover the image from noise that is called denoising images. In this section I will implement two important methods for image denoising, in the next section i will explain the methods in details.

Code:

DenoiseImage.m:

It has taken some of my sample images, noise them and finally denoise them. It includes:

Image => salt & pepper noise => denoised image

Image => gaussian noise => denoised image

important : For denoising salt & pepper, I have used methods **fspecial** and **filter2,** and method **wiener2** for gaussian noise. **They have been explained in the next section**.

Image denoising research

Different types of noises, result in different approaches for removing them. One famous method for removing noises is **linear filtering**. Averaging and gaussian filters are good for this task.

Averaged/Median Filtering

It is useful for removing grain noise from a picture. What it actually does is that each pixel gets set to the average of the pixels in its neighbourhood, so local variations that are caused by grains, will be reduced. For example, for removing salt & pepper noise, I have used a mixture of average filtering and median filtering in the previous section. These filtering types set the value of the output pixel to the average of the pixel values near the input pixel. But in median filtering it is not average, it is the median operator. For outliers of an image, median is so much less sensitive than average. So median filtering is better for removing the outliers without changing the sharpness of the image. So, first I made the picture noisy with salt & pepper. Then, i have used a 2 by 2 neighbourhood for implementing averaged filtering, it is done like this:

```
AF = filter2(fspecial('average', 2), image) / 255
```

Finally, we implement median filtering on the result:

```
MF = medfilt2(AF)
```

Adaptive Filtering

It is another famous linear filtering method. We can do it by the **wiener2** function in matlab. It actually tailors itself to the local image variance. So the larger the variance, the less smoothing it does. But where the variance is small, it performs much more smoothing. I have used it for the second image that was noised by gaussian method:

```
AF = wiener2(Image, [3  3])
```

The advantage of this approach is that it preserves edges and high-frequency parts of an image. But the disadvantage is that it takes much more time for computation.

Noise Measurement

Now just one thing is left, measuring the amount of noise in an image so that we can rate how good we denoise images by comparing the amount of noise in an image with its denoised version. For that there is **snr** method in matlab. First i explain **SNR**, then i will explain my implementation in **MeasureNoise.m**

SNR(Signal-To-Noise Ratio):

SNR returns the signal to noise ratio of an image in decibels. It is done by computing the ratio of its summed up squared magnitude to that of the noise. So assuming the difference between the smoothed image and the original is considered SNR.

The first input of the SNR method must be the original image so that the result becomes positive.

A low snr means the noise is big and vice versa. Now time to implement that.

Code:

MeasureNoise.m:

It takes my image and makes it noisy, then denoises it as I explained in previous sections. Finally computes the snr between the original image with the noisy and denoised version of the image. The second one must be higher since it has been denoised and it is.

Important: after running this file, outputs are printed. The output is about the comparison between SNRs.

Summary

This project was so interesting for me. Actually I always try to do my best on my courses. This project totally covered everything that the original documentation of the project has requested to implement or research about. The topics of the project are mentioned in the first page in the overview section. I'm glad to put a lot of time and effort on this project.

Briefly, I learned many things about the world of image processing. Also my matlab coding skill improved a lot. Not only that, but also I learned much more things in the combination of matlab and image processing.

Thanks for your attention!

- Sohrab Namazinia
- 97522085