**Introduction:**

Image processing and analysis is a branch of electrical engineering, which has wide applications in many fields of engineering. In the field of pavement engineering, image processing and analysis has wide range of applications Image processing and analysis can be used for Identifying cracks and rutting, Identifying the texture of the pavement, Determine the gradation of the mixture, Determine the contact points and orientation of the aggregates, Surrogate measure for mechanical tests. This report primarily deals with processing and analysis of image for studying the grading curve of aggregate and determining various mixture parameters through extracted data.

To study the morphological properties of the mixtures, 2D planar images can be utilized. 2D planar images can be obtained by sectioning the asphalt mixture either vertically or horizontal and scanning using a flatbed scanner.

Before image using for analysis some manipulation is needed(pre-processing) to get the most accurate result. Pre-processing is done as per the following method.

• Image acquisition

• Point processing

• Neighbourhood processing

• Image restoration

• Image segmentation

Every image consists of pixels. When image is clicked by the camera it takes value of every pixel in the form of value of RGB. Every code is in between 0-255. Before the analysis of an image is been converted to Grayscale(each pixel has value in between 0-255). Gray scale image helps during removing of unwanted noise and unwanted pixels.

For the pre-processing and analysis python language and open-cv framework is used in entire report.

img = cv.imread (cv.samples.findFile("G:\my Ml projects\image processing\p5.PNG"))

above code snippet takes the image from the local path and convert it into a NumPy array based of RGB values of each pixel.

For converting RGB image into Grayscale; following code help us.

gray = cv.cvtColor(img,cv.COLOR\_BGR2GRAY)

if we see mathematically it add value of R, G, B of every pixel and take an average of it. And give that 0-256 value to the respective pixel.

Grayscale image is not sufficient to get accurate results.

In order to measure the objects in the image, the contrast between the foreground and the background should be more.

* The contrast in the image can be enhanced for better visualization
* histogram of the grayscale image is an indicator of the contrast
* histogram which is skewed towards left end or right end indicates a poor contrast
* An image with good contrast has a broad histogram with pixels at all the intensity levels

Histogram equalization method usually increases the global contrast of many images, especially when the usable data of the image is represented by close contrast values. Through this adjustment, the intensities can be better distributed on the histogram. This allows for areas of lower local contrast to gain a higher contrast. Histogram equalization accomplishes this by effectively spreading out the most frequent intensity values.

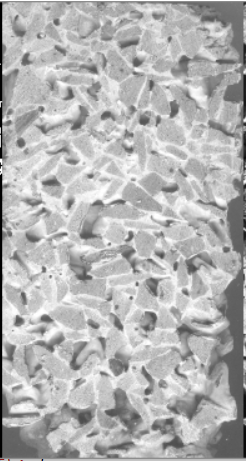
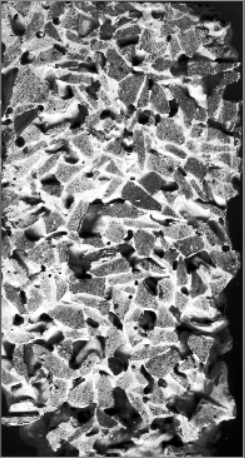
 

Figure : Grayscale image Figure : Grayscale image after histogram equalization

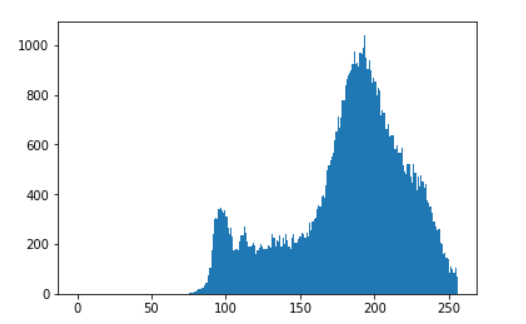


Figure : without histogram equalization

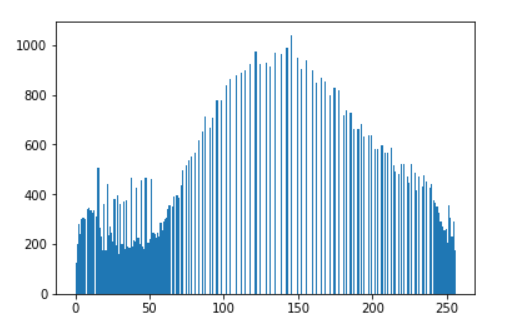


Figure : Automatic equalize histogram (Normalization)

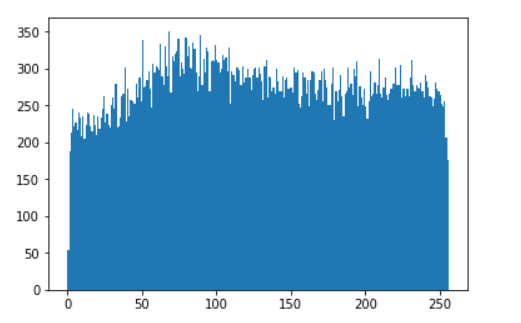


Figure : Clahe equalization

Normalization: ref Figure 4

equ1 = cv.equalizeHist(gray)

this python code stretches the histogram to equalize the pixel values over the range. **normalization** is a process that changes the range of pixel intensity values. Applications include photographs with poor contrast due to glare, for example. Normalization is sometimes called contrast stretching or histogram stretching.

Normalization transforms an n-dimensional grayscale image {\displaystyle I:\{\mathbb {X} \subseteq \mathbb {R} ^{n}\}\rightarrow \{{\text{Min}},..,{\text{Max}}\}}with intensity values in the range (Min,Max), into a new image {\displaystyle I\_{N}:\{\mathbb {X} \subseteq \mathbb {R} ^{n}\}\rightarrow \{{\text{newMin}},..,{\text{newMax}}\}}with intensity values in the range (newMin,newMax).

New value of pixel intensity will be

In =

In = New intensity value of pixel

I = original pixel intensity value

Adaptive histogram equalization: ref figure 5

clahe = cv.createCLAHE(clipLimit=12.0, tileGridSize=(8,8))

equ = clahe.apply(gray)

Image is divided into small blocks called "tiles" (tile Size is 8x8 by default in OpenCV). Then each of these blocks are histogram equalized as usual. So, in a small area, histogram would confine to a small region (unless there is noise). If noise is there, it will be amplified. To avoid this, contrast limiting is applied. If any histogram bin is above the specified contrast limit (by default 40 in OpenCV), those pixels are clipped and distributed uniformly to other bins before applying histogram equalization. After equalization, to remove artifacts in tile borders, bilinear interpolation is applied.

Filter:

Some images have extra noise which can make errors in analysis when we convert image in binary image. Filters helps to reduce that noise.

In neighbourhood processing, a function is applied to a pixel and its neighbourhood. The basic concept is to move a mask (1D or 2D) over the image, alter the pixel and obtain a new image.

The combination of a mask and function is called a filter. The 1D or 2D mask moves over the image pixels. Multiply each value under the mask matrix with pixel values add it and then give the new grayscale value to the pixel.

blur = cv.bilateralFilter(equ,5,51,51)

cv.bilateralFilter() is highly effective in noise removal while keeping edges sharp. But the operation is slower compared to other filters. Gaussian filter takes the neighbourhood around the pixel and finds its Gaussian weighted average. This Gaussian filter is a function of space alone, that is, nearby pixels are considered while filtering. It doesn't consider whether pixels have almost the same intensity. It doesn't consider whether a pixel is an edge pixel or not. So it blurs the edges also, which we don't want to do.

Bilateral filtering also takes a Gaussian filter in space, but one more Gaussian filter which is a function of pixel difference. The Gaussian function of space makes sure that only nearby pixels are considered for blurring, while the Gaussian function of intensity difference makes sure that only those pixels with similar intensities to the central pixel are considered for blurring. So, it preserves the edges since pixels at edges will have large intensity variation.

In pavement images we need to keep the edges sharp because it will help to make most accurate results about results.

Final step for pre-processing of an image is converting well contrasted image into binary image. Binary image is an image which has only two intensity value pixels. That is, 0 for black and 255 for white.

When we give some threshold then the python function will make all the intensity above threshold equals to 255 and below values to 0.

From equalized histogram, manual threshold value has been taken.

In cement paste pavement mixture, we have three components.

1. Cement paste
2. Aggregate
3. Air voids

For finding results, three binary images had taken. One is for cement paste, second is for air voids and third is for aggregate.

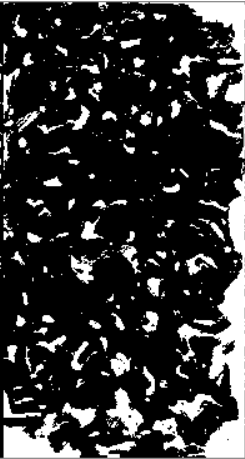
ret,th = cv.threshold(equ,170,255,cv.THRESH\_BINARY) #for cement paste

ret,th1 = cv.threshold(equ1,50,255,cv.THRESH\_BINARY\_INV) # for air voids

cv.threshold(equ,170,255,cv.THRESH\_BINARY) converts pixels whose intensity is greater than 170 to 255 and pixels whose intensity is below 170 converts to 0.

cv.threshold(equ1,50,255,cv.THRESH\_BINARY\_INV) due to cv.THRESH\_BINARY\_INV converts pixels whose intensity is greater than 50 to 0 and pixels whose intensity is below 50 converts to 255.

This Binary inversion image is used to make a binary image of aggregate.

By observing above two images we can make image for aggregate by adding both the images. After addition of both the images if binary inversion happens then white pixels has given aggregate.

dst1 = cv.addWeighted(th,1,th1,1,0) #added image 1 and 2

ret,dst = cv.threshold(dst1,150,255,cv.THRESH\_BINARY\_INV) #for binary inversion



After getting the pre processed image. We can proceed towards analysis for grading curve of the aggregate.

Analysis:

Two steps have done for Analysis.

1. Total Area of aggregate, Cement paste and Air voids.
2. Area of each aggregate using connected components method.

**Total Area of aggregate, Cement paste and Air voids**:

Previously, we got three different images for cement paste, Aggregate and Air voids. In each image white pixels shows cement paste, Aggregate and Air voids.

Following python code help in finding respective areas.

n\_white\_pix\_w = np.sum(th2==255) #total pixels in the whole image

n\_white\_pix\_v = np.sum(th1==255) #White pixels in air voids image

n\_white\_pix\_c = np.sum(th==255) #White pixels in cement paste

n\_white\_pix\_a = np.sum(dst==255) #White pixels in aggregate image

When image got converted into binary then we only have pixels with intensity values 0(black) and 255 (white). Measuring total number of pixels having intensity value 255 gave number of white pixels. Using geometric scaling total area of those white pixels has been calculated.

Following python code do the geometric scaling to find out area.

dimensions = gray.shape

speciman\_area = 100 \* 200

number\_of\_total\_pixels = dimensions[0] \* dimensions[1]

mm\_square\_per\_pixel = speciman\_area / number\_of\_total\_pixels

area\_of\_v = n\_white\_pix\_v \* mm\_square\_per\_pixel #area of voids

area\_of\_c = n\_white\_pix\_c \* mm\_square\_per\_pixel #area of cement

area\_of\_a = n\_white\_pix\_a \* mm\_square\_per\_pixel #area of aggr.

**Area of each aggregate using connected components method:**

* **Connected component:**

Connected components labelling scans an image and groups its pixels into components based on pixel connectivity, i.e. all pixels in a connected component share similar pixel intensity values and are in some way connected with each other. Once all groups have been determined, each pixel is labelled with a Gray level or a colour (colour labelling) according to the component it was assigned to.

Extracting and labelling of various disjoint and connected components in an image is central to many automated image analysis applications.

* How is it work?

The algorithm involves two whole passes through each pixel in the image.

First pass:

For each non-zero pixel, we check its neighbours.

If it has no non-zero neighbours — we know it is a new component — so we give it a new label.

If it has one non-zero neighbour — these pixels are connected — we give it the same label as the neighbour.

If it has more than one non-zero neighbour, there are two cases:

The neighbours all have the same label. So, we give the current pixel the same label.

The neighbours have different labels. This is the tricky part. We know all of these pixels are connected now, so the labels should be all the same. The classical approach to solve this is the following. We set the current pixel to the neighbours’ lowest label. Then we also keep a note of the equivalences of all the connected labels — that is, which different labels should actually be the same. We will solve these on the second pass!

Each non-zero pixel will now have a label. However, some connected regions will have different labels. So we need to go over the image one more time to solve this. We do this by using our recordings of the equivalence classes: all the labels that are equivalent (i.e. refer to the same blob), will get the same label.

Second pass:

For each pixel with a label:

Check if this label has equivalent labels and solve them.

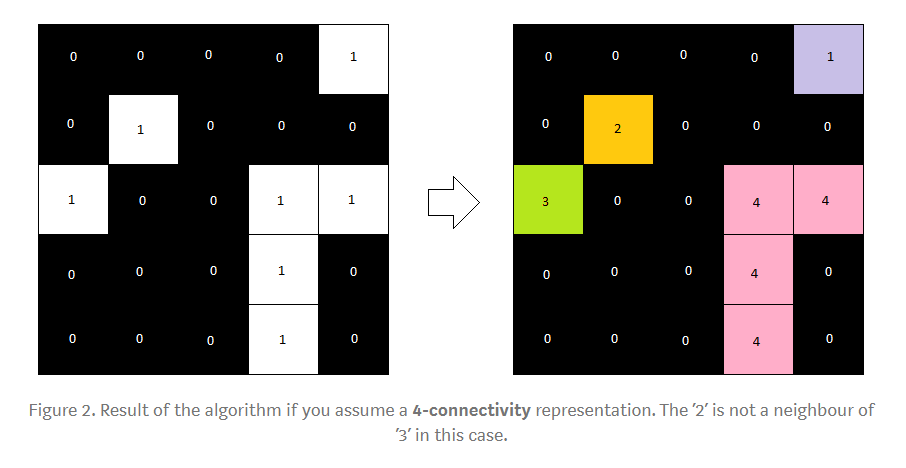
****

Figure : 4 connectivity representation

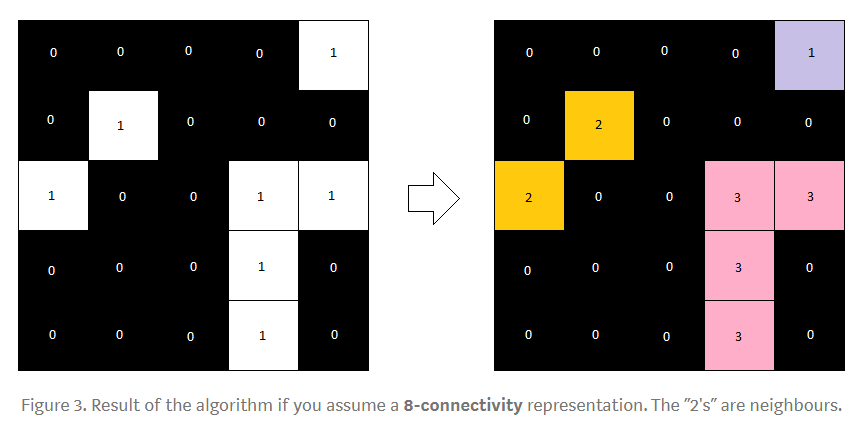
****

Figure : 8 connectivity representation

Pseudo code for Connected component labelling:

**algorithm** TwoPass(data) **is**

linked = []

labels = structure with dimensions of data, initialized with the value of Background

*First pass*

**for** row **in** data **do**

**for** column **in** row **do**

**if** data[row][column] **is not** Background **then**

neighbors = connected elements with the current element's value

**if** neighbors **is** empty **then**

linked[NextLabel] = ***set*** containing NextLabel

labels[row][column] = NextLabel

NextLabel += 1

**else**

*Find the smallest label*

L = neighbors labels

labels[row][column] = ***min***(L)

**for** label **in** L **do**

linked[label] = ***union***(linked[label], L)

*Second pass*

**for** row **in** data **do**

**for** column **in** row **do**

**if** data[row][column] **is not** Background **then**

labels[row][column] = ***find***(labels[row][column])

**return** labels

There is a function in opencv to deal with the above psudo code for Connecting component method.

num\_labels, labels = cv.connectedComponents(dst,connectivity=8)

# connectivity is an argument which takes value 4 or 8 as shown above.

# Map component labels to hue val, 0-179 is the hue range in OpenCV

label\_hue = np.uint8(179\*labels/np.max(labels),axis=1)

blank\_ch = 255\*np.ones\_like(label\_hue)

labeled\_img = cv.merge([label\_hue, blank\_ch, blank\_ch])

This converts each label pixel into hue value colour.

labeled\_img[label\_hue==0] = 0

This converts black pixel label to 0.

labels = labels.reshape(np.size(gray))

print(np.shape(labels))

label\_list = labels.tolist()

label\_set = set(label\_list)

total\_labels = list(label\_set)

for label in total\_labels:

print("area of each label image {} = ".format(label),label\_list.count(label) \* mm\_square\_per\_pixel)

This give us the area of each label. i.e area of each aggregate.