Name: Remaldeep Singh
UID: u1143744
Email: u1143744@utah.edu

This project is divided into two parts:

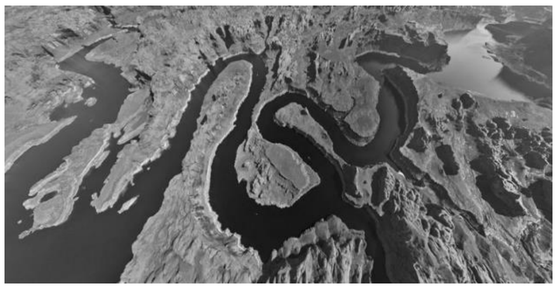
- Histogram Equalization and image coloring
- Cell counting with different thresholding techniques.

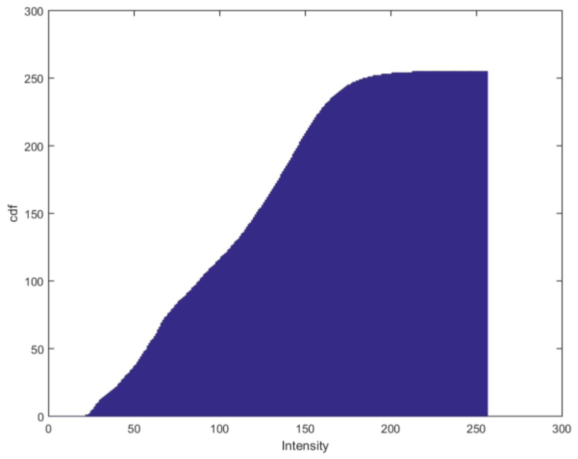
The main.m file in the project directory is used to call all other functions.

Image colorization

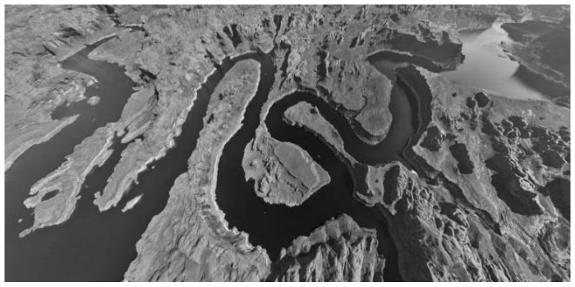
This part is under Step 1 of main.m

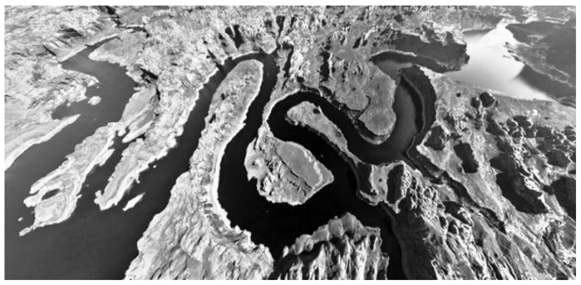
1) **Histogram Equalization array**. *Step1.1* of the main.m file.
   The function histogramEqualize, in "Functions/histogramEqualize.m", takes in Image as input and returns an 1-D array mapping values from 0 to L-1.
   First it calculates the histogram of the image then creates a PDF(Probability Distribution Function) out of it. Consequently, it calculates cdf and returns after rounding of. The createPdf method creates the pdf for the histogram.
   An example of histogram array output of the lake image.

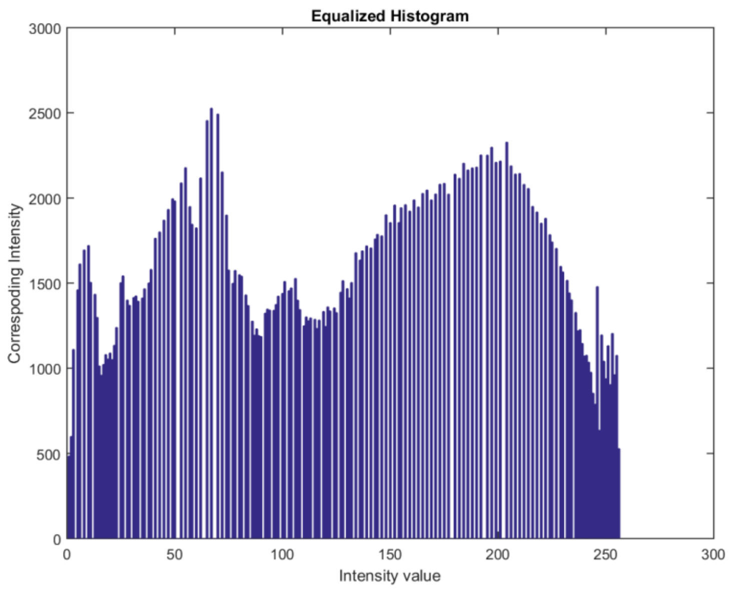




2) **Intensity Transformation.** *Step 1.2* of the main.m file.
   The function intensityTransformation, in "Functions/intensityTransformation.m", takes input as image and equalized histogram and returns an equalized image after pixel intensity transformations. The method maps the input pixel values to the histogram bin and then reads the corresponding bin value. An example of equalized image and histogram:

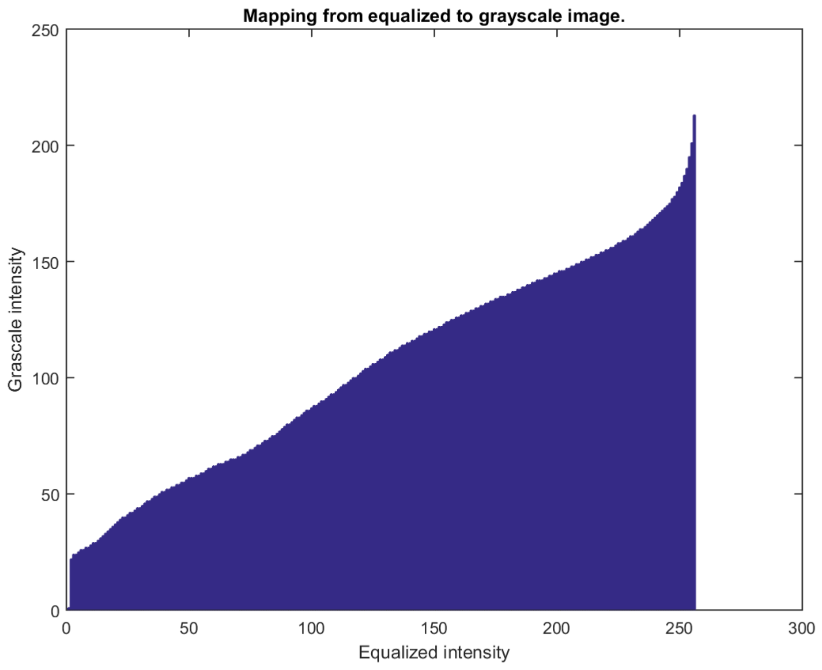




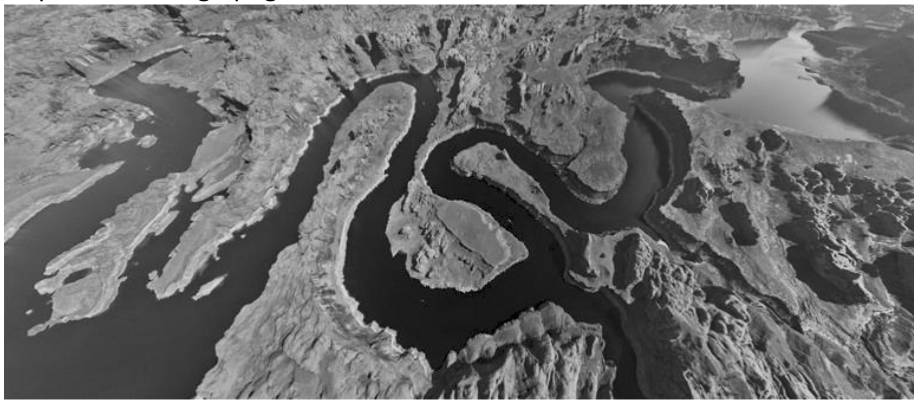


3) **Inverse histogram equalization transformation.** *Step1.3 of main.m file.*
The function inverseHistogramEq, in “Functions/ inverseHistogramEq.m”, takes in grayscale as input and returns the mapping of equalized grayscale image with the original grayscale image.

The method starts by creating a cdf of input image. Then creates an equalized image with the “intensityTransformation” method and consequently a cdf for the equalized image. It then starts mapping the bin value of the equalized cdf (sGrayEq) with the closest cdf (sGray) value of the input graysclae image. It keeps on storing this mapping in the mapping array, which is later returned as A. Mapping array is saved as “Ouptut/mapping.png”
Mapping array looks like:



To **reproduce** the original grayscale image each intensity pixel value of the equalized image is mapped with the mapping array back to get the new pixel value. The output image is stored as "Outputs/reproducedImage.png".



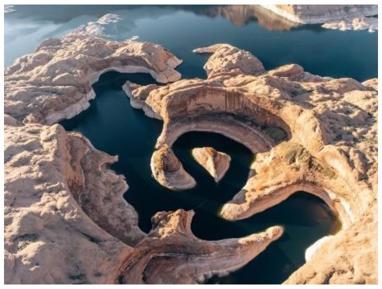
4) **Inverse histogram equalization and coloring.** *Step 1.4* of main.m file.
The function colorizeImage, in "Functions/ colorizeImage.m", takes input as grayscale image and color image. It returns the colorized output of the input grayscale image with reference to the input color image.

The method starts by creating a cdf of grayscale image with the histogramEqualize method. Then it creates 3 histogram 1-D arrays (cdf's), one for each channel (RGB) in the image. Iteratively for each channel of the color image, it loops over all the pixels in the image and chooses the closest neighbor to the histogram equalized value **(sk)** of that pixel. It does so by picking the first bin/intensity of the current channel histogram as the closest neighbor **(smallestZq, smallestGzq)** and then iterates over rest of the bins in the channel histogram, checking at each bin step if the difference between sk and the histogram bin value "**abs(sk-invHistEqualizationA(1,q))"** is less than the original **"sk - smallestGzq"**. For each pixel of the new colored image, the smallestZq value is the pixel value.

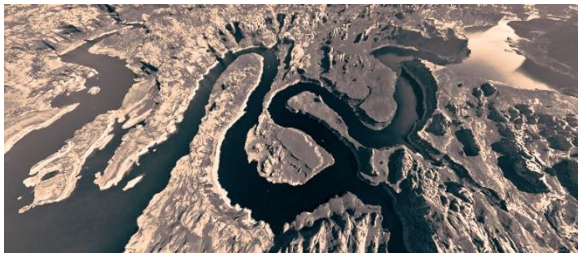
The output is stored as 'Outputs/ColoredImage.png'. Example:

Grayscale

Colored Reference

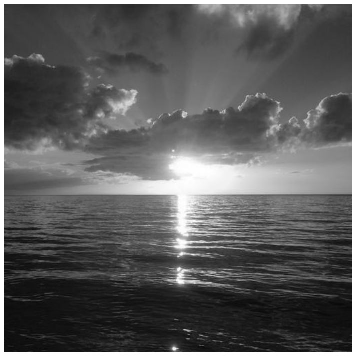


Colored Image

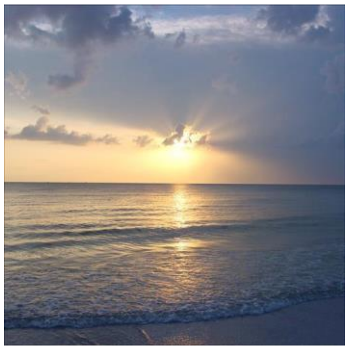


Additional Outputs:

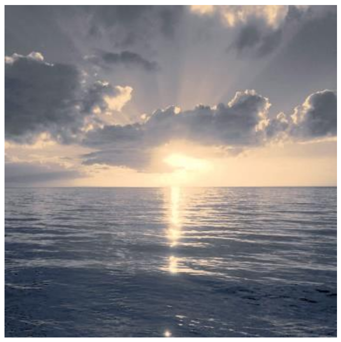
Grayscale



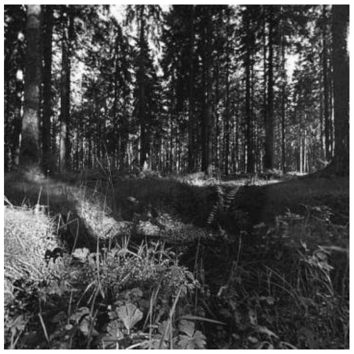
Colored Reference



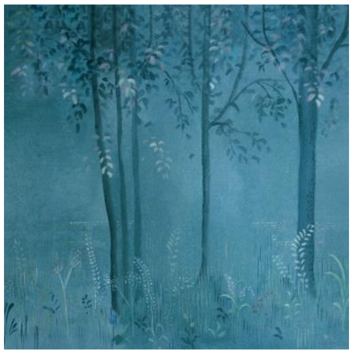
Colored Image



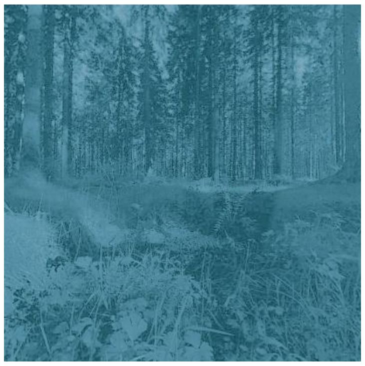
Grayscale



Colored Reference



Colored Image



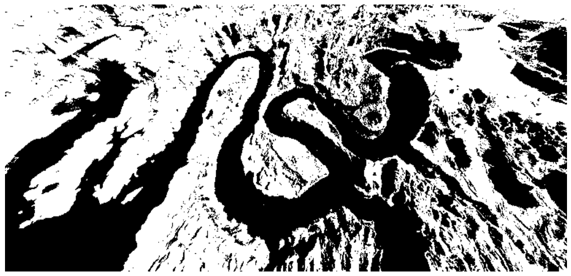
Counting Cells

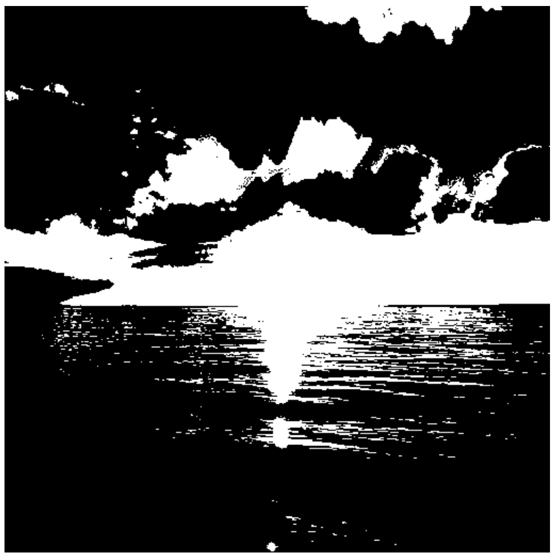
This part is under Step 2 of main.m

1) **Otsu Threshold method:** *Step 2.1* of the main.m file.
The function otsuThreshold, in “Functions/ otsuThreshold.m”, takes in image file name as input and returns the new threshold image as output. It takes in image file name as input to calculate the bitdepth with **imfinfo** command.

It starts by creating the histogram of the image and creates pdf out of it. Now the idea is the find that speed spot wherein the between class variance is maximum i.e in class variance is minimum. It does so by iteratively taking each bin of the pdf as the threshold and calculating the mean and weights of both sides of the threshold. Once it has calculated the mean and the weight it calculates the variance. If the variance is greater than the maxVariance (initialized with -1) it stores that variance and threshold in the maxThreshold and maxVairance. Then the threshold is applied on the input image based on the maxThreshold.

The method calculateWeights calculates the weights on both sides of the input threshold in the input pdf. Also, the method calculateMeans calculates the means on both input threshold. Example output:





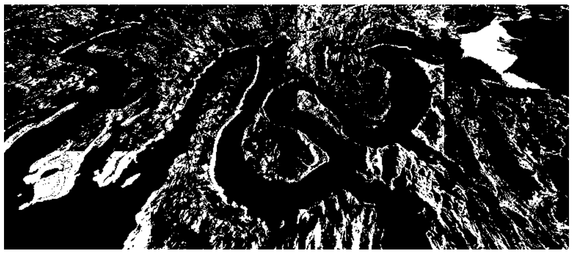
2) Adaptive Thresholding: *Step 2.2 of the main.m*

The function adaptiveThresholding, in "Functions/ adaptiveThresholding.m", takes in file name, block size and the minimum variance as the input. It returns back threshold image based on the block size and minimum variance. It reads the file name to calculate the bit depth of the image with the **imfinfo** method.

It divides the image into blocks of size input **bSize** and then loops over all the blocks individually. Within each block it calculates the start index and end index for i,j respectively and creates a mini image out of the complete image for that block. This mini image (**blockIm**) is then passed to the method thresholdBlock which threshold the image based on minimum variance within the block.

Within the thresholdBlock method the histogram of the image is calculated and a pdf out of it is created. Then we calculate the mean and variance inside the block and check if the variance is greater than the minimum variance. If it is this tells us there is a cell in this block. Hence we therhsold the block with the threshold of mean plus standard deviation (square root of variance). The thresholded block is then returned and is placed in exactly the same position in the complete image from where it came from.

Example Output with (200,200) block size and 1000 minimum variance.



3) Cell Counting: *Step 2.3 of the main.m file.*

The function countCells, in "Functions/countCells.m", takes in file anme as input and calculates the number of cells in the image by two methods otsu and adaptive thresholding.

This method first tries to identify the number of cells with otsu thresholding and gets a count of 38 cells. It applies otsu threshold method and then performs connected components to get the total cells in the threshold image. It then denoises the image to remove cells with area less than 10 (**which are not cells**). I tried a lot with this threshold value and arrived at 10 as the sweet spot. The final output as number of cells is displayed on the screen.

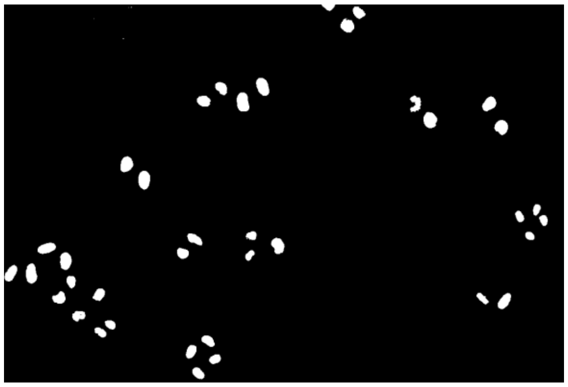
Secondly adaptive thresholding is applied with block size (60,60) and minimum variance of 200. Found this block size as the sweet spot since it's a bit bigger than the area of the cell and we can safely assume that there is a high probability of having an entire cell in this block (although blocks are non-overlapping). It then applies connected components and consequently image denoising to remove cells that are not cells (with area less than 10).

Difference: The difference in these two methods in context of this problem is that although otsu selects the threshold automatically by maximizing the between class variance. It is unable to find the optimal threshold (It selected 33486 as threshold). There are still minor peaks that could have been identified to the left of this threshold that were actually cells. Maybe with multiple classes threshold it would have been able to find the cells that are towards the left of the max threshold.

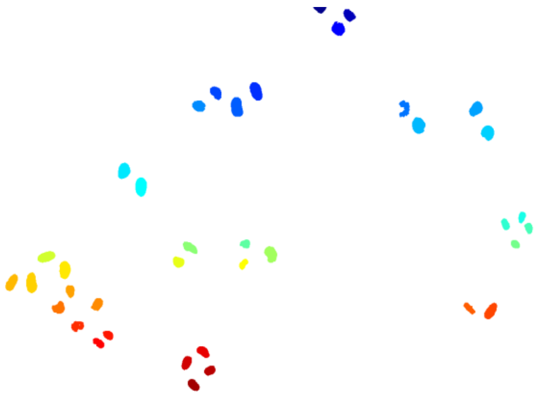
Whereas in the adaptive thresholding we select the threshold as well as block size manually. So, whenever we encounter a cell within the block we know the variance is going to be high (hence I selected 200 in this case) as opposed to having only background in the block in which case variance will be low. Consequently, this method was able to find those minor change in intensites better than the otsu method. The only downside to this is that we have to select eh block and threshold manually for each problem separately.

Example Outputs **Otsu**:

Otsu Threshold Image

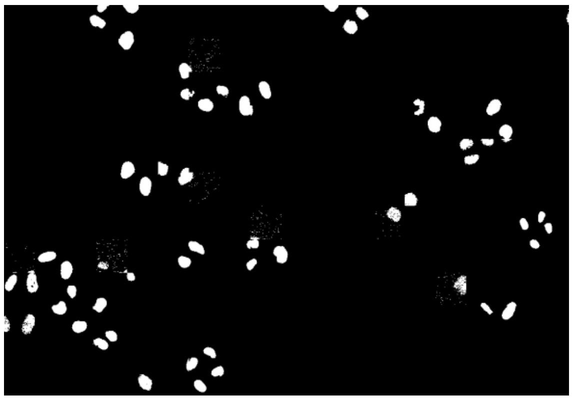


Connected component denoised.



Example Output **Adaptive thresholding**:

Adaptive Threshold Image



Connected component denoised.