18MCMI09
Krishnan.S
Colour Image Processing
Assignment No:2

All operations were implemented in python3 using OpenCV and NumPy. The code for each of the questions are in the folders labeled “Question X”, along with the input and output images.

1.
I implemented the naive ordered dithering algorithm based on a custom mask in both RGB and CMY. Since the second question had error diffusion in it already, I wanted to do just dithering for this one. A lot of the papers had a combination of dithering of error diffusion, and both kinds of dithering were present.

There are two kinds of dithering algorithms; dispersed dot and clustered dot dithering. Clustered dot dithering is analogous to AM Modulated Halftoning, it increases the size of the “dot” printed, or the cluster as the presence of the color increases. Dispersed dot dithering is analogous to FM Modulated Halftoning, which is what I did.

I blew up the image 4x4 times and then put CMY/RGB dots in the image according to the masks given below.

$$RGB_{mask} = \begin{bmatrix} G1 & B1 & G7 & G5 \\ R1 & G6 & G2 & R3 \\ B2 & G4 & R2 & B3 \\ G3 & B4 & R4 & B5 \end{bmatrix}$$

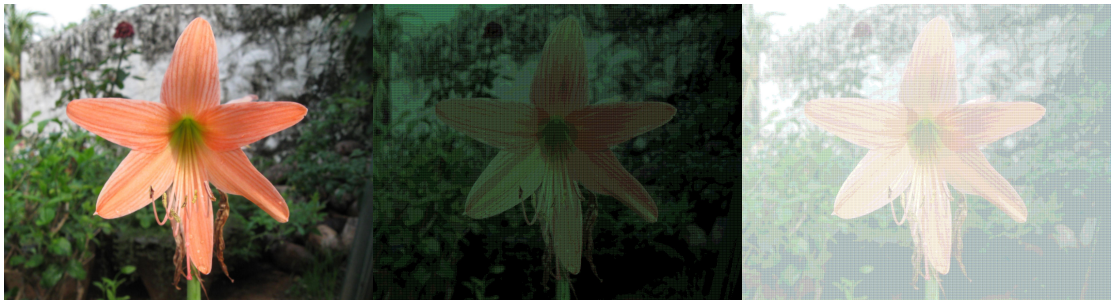
$$CMY_{mask} = \begin{bmatrix} Y1 & C1 & M4 & C4 \\ Y3 & M1 & C3 & M2 \\ C6 & M3 & M5 & C2 \\ Y5 & C3 & Y2 & Y4 \end{bmatrix}$$

The masks were made (loosely) according to the orientation of the meshes or screens that they use in printers. The subscripts represent the order in which the dots need to be put. The number of dots to be put for each color is based on a threshold.
The threshold ranges were calculated by dividing 255 by the number of dots of each color in a mask. So in the RGB mask, there are four red points, so the threshold ranges are 0-50, 51-101, 102-152, 153-203 and 204-255. If a pixel were to have a red value of 202, then four dots would be put in the 4x4 region.

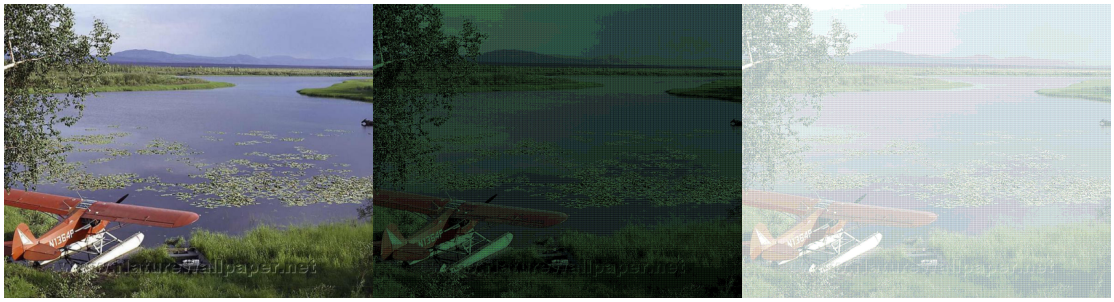
The new images had to have a background of black and white for RGB and CMY respectively as RGB can’t produce black and CMY can’t produce white.

The function prototypes are RGBDither(image) and CMYDither(image).

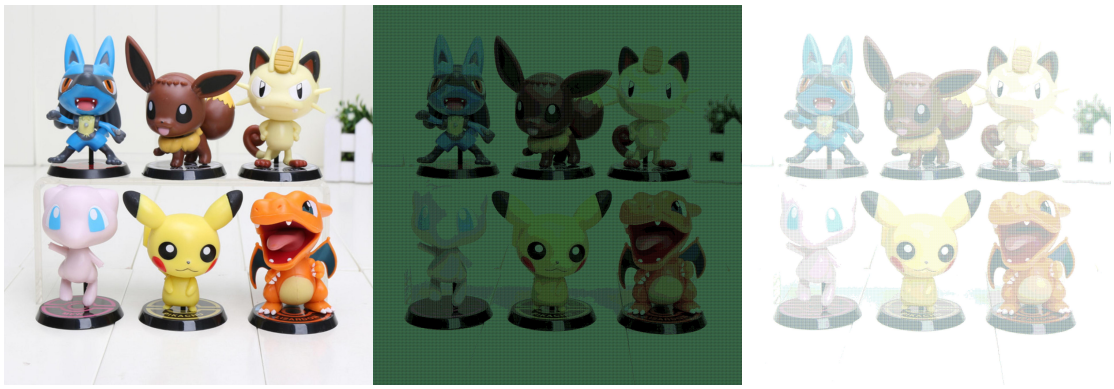
Here are the output images. The first one is the original, the second is the RGB dithered one and third one is the CMY dithered one.



orange-flower.ppm



waterplane.ppm



figurines.jpg

The CMY dithered ones look brighter than the original because of the background being white initially. Similarly the RGB dithered ones look darker because of the background being black initially. Since the masks aren't that perfect, you can see the dither patterns clearly. The screen tends to smoothen out and make the patterns even more apparent on my end, it might be a software thing.

2.
The error diffusion algorithm works like this; you take a pixel and threshold it with respect to 127. If its less than 127 its set to 0, 255 otherwise. The distance of the intensity value of this pixel before thresholding from 127 is the error. It's negative if it's greater than 127 and positive if it's lesser. Fractions of this error is propagated to the surrounding pixels (in the forward direction according to raster scan), the error is diffused throughout the image. So effectively, the image gets recreated with only black and white.
This is for grayscale images. For color images, one would check the distance from a set of colours, typically CMYRGBWK and propagate that distance/error.

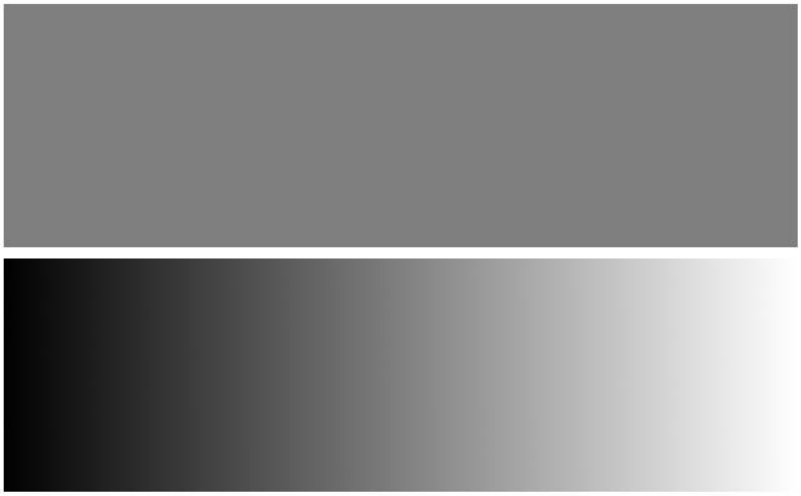
I tried error diffusion using two sets of coefficients. The coefficients I used and the Floyd Steinberg ones are given below.

$$FS = \begin{bmatrix} & * & \frac{7}{16} \\ \frac{3}{16} & \frac{5}{16} & \frac{1}{16} \end{bmatrix}$$

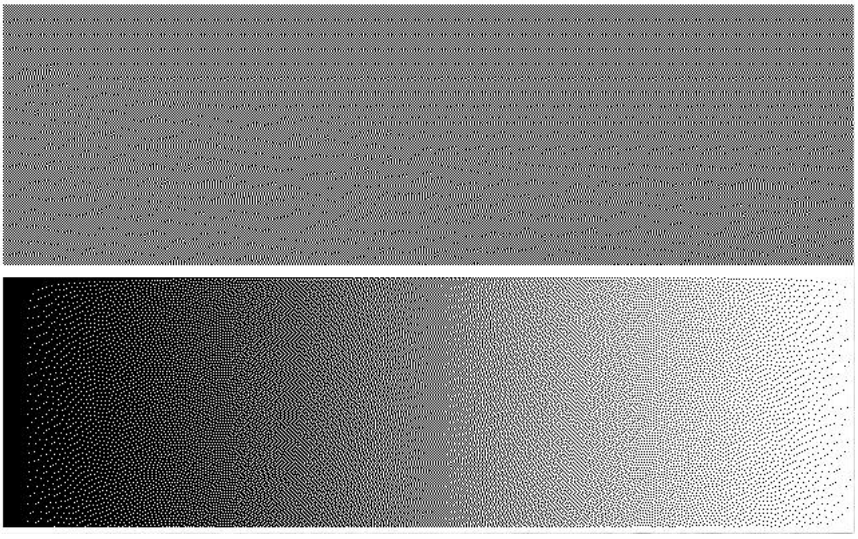
$$Mine_1 = \begin{bmatrix} & * & \frac{18}{340} \\ & \frac{62}{340} & \frac{260}{340} \end{bmatrix}$$

$$Mine_2 = \begin{bmatrix} & * & \\ \frac{94}{219} & & \frac{80}{219} & \frac{45}{219} \end{bmatrix}$$

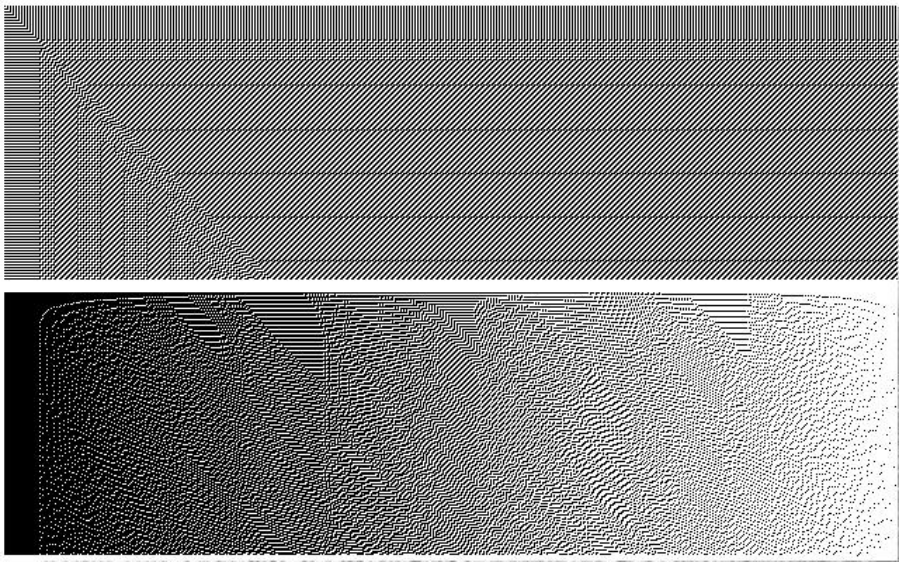
The * represents the pixel we're currently on. The coefficients around it determine how much of the error should be propagated to that pixel. For Floyd Steinberg, the pixel to the right of the current gets 7/16ths of the error, the one below it gets 5/16th and so on.
The coefficients I picked were arbitrary, they just had to add up to one. For the first one I decided to give this huge diagonal sort of error movement towards the right, but stuck with only one pixel movement. For the second I moved two pixels ahead to the right and there's sort of a combat between going forward and backward. This shows in the output images.
Here's the original,



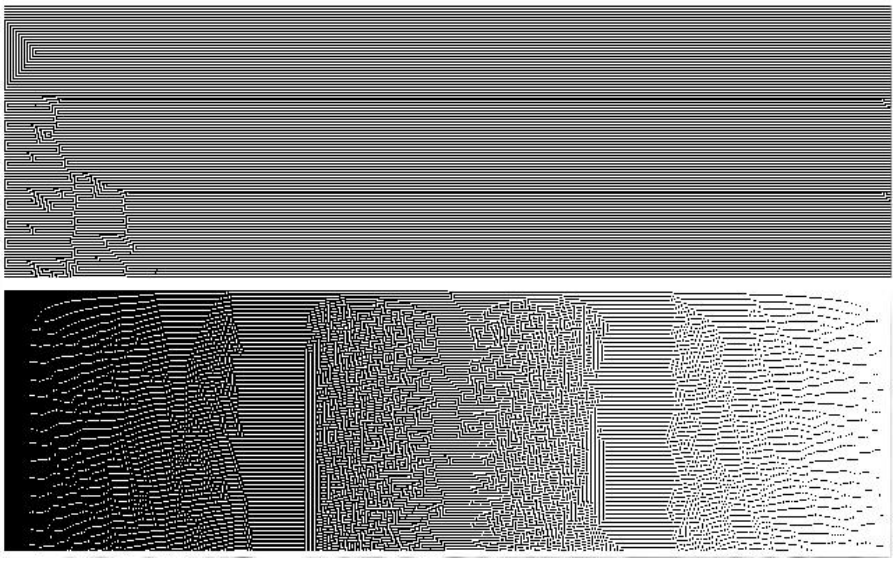
With the Floyd Steinberg coefficients,



With my first set of coefficients,



With my second set of coefficients,



With Floyd Steinberg there seems to be a somewhat even distribution of the points, and there's not much of a pattern except for the wavy things at places. It looks really smooth compared to mine. In the middle section of the gradient image, around 127, there seems to be a solid block of 127 there which is absent in the images with my coefficients. Mine makes a lot of patterns there.

The strong diagonal coefficient set produces strong diagonal like patterns in the image too. In fact this pattern seems to be there throughout all the greyscale regions, particularly at the middle region between 255 and 127, and 127 and 0. There are nice stripey patterns on the once pure grey block in the directions of the coefficients.
The second set of coefficients seem to have made a worse error diffused image compared to the first. It almost seems like the greys are stretched out like how the coefficients are stretched out. The levels seem to converge to two or three greyscales and then diverge from it. The pure gray block now has a lot of vertical stripes since the error move vertical rather than horizontal.

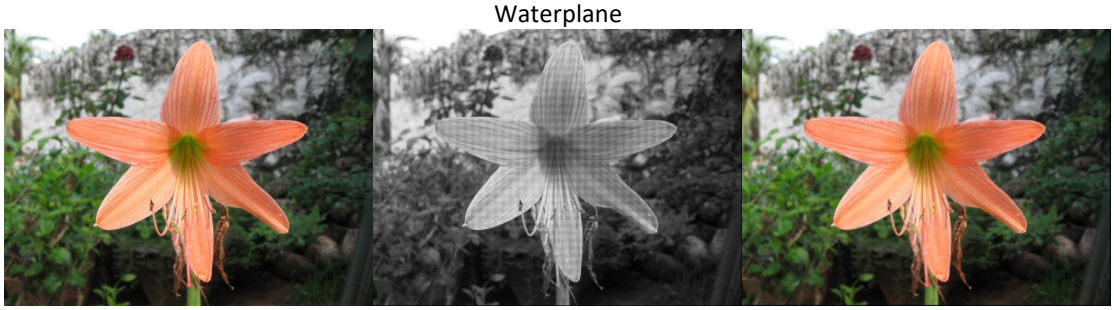
All of them have this rounded rectangle sort of boundary where the different dots start to appear on either side of the gradient image.

3.
The color filter array I used was

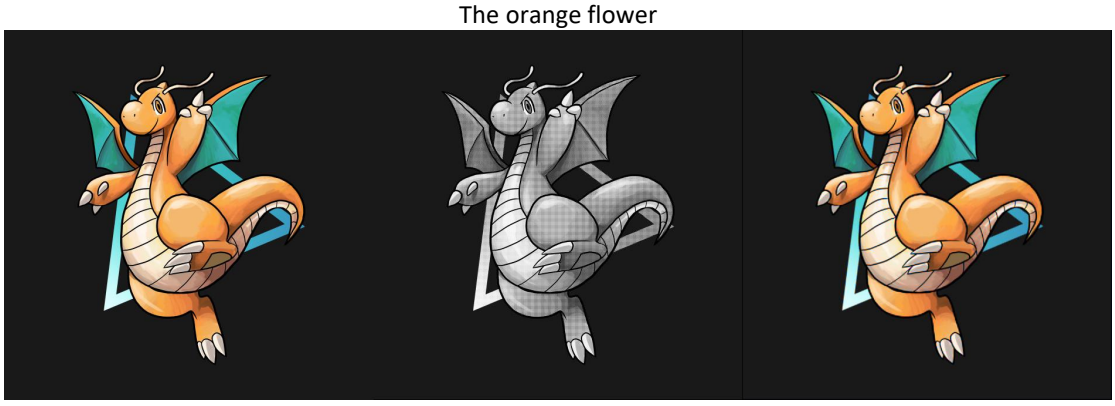
$$Filter = \begin{bmatrix} B & G \\ G & R \end{bmatrix}$$

Any filter will work with the functions as long as it's a square filter. I changed the prototype to colour_filter(image, filt) since "filter" pops up as a list function.
For the filter color array part, what I did was take the intensity value of the color indicated by the filter's first index at the given position and then use that as the grayscale value for the raw image. So say at a given point G was over the current pixel. I would use the value of G in the original image as the grayscale value. Zooming in on the image obtained from this process shows you lots of differently grey coloured squares, just like how a filter array would look like.
I moved the filter over to it's various positions using the **np.roll()** function. Each time I moved to the next pixel in the same row, I would shift the second axis of the filter by one. Each time I moved to the next row, I'd shift the first axis by one. I suspect I could've done it more easily using some modulus trickery.
For the demosaicking part, you check what the structure of the filter is like at the current position and calculate values according to that. For the filter array I tried it with, you would always have one blue value, the average of two green values and one red value. These three make up the vector for the output pixel. As long as the filter used for the filter color array and demosaicking part are the same, most of the input image is reconstructed. Color bleed shows up on the output images as shown.
Here are the various images I tried it with,

Waterplane



The orange flower



Dragonite