CIP-ASSIGNMENT-2

TASLEEM SULTHANA
18MCMT17
MTECH-CS

In [1]:
```
import cv2
import matplotlib.pyplot as plt
import numpy as np
import glob
import copy
```

1. Search the web for Colour Halftoning algorithms. Select one of them and write a detailed report on it OR implement the selected algorithm and show results on the test images.

REPORT : colour halftoning algorithms.

Halftoning is method of generating the a continuous image by using binary outputs of printers. we know that a printer can either put a dot and leave the it without dotting,but cannot dot a portion.

In ordered dot dithering let us consider a alogirithm in which is a colour dithering algorithm implemented by resizing the new image 4 times of the original.

the order of the dithering will be as follows

```
M(i, j) =    [ 1 2
               3 0 ]
```

Let us consider a random positions for the CMY on the 4*4 of a single pixel

```
[ C M Y C
  M C M Y
  M C M Y
  C Y Y C ]
```

for the above assigned CMY we can dot C=6, M=5 , Y =5

now lets generate the order of dotting for 4*4 by using the conditions

```
[ 4*M    4*M+2
  4*M+3  4*M+1]
```

Then the genrated 4*4 ordered dithering positions will be as follows

```
[ 0  8  2  10
  12 4  14  6
  3  11  1  9
  15  7  13 5]
```

the colour position numbering

```
[C1 M3 Y1 C4
 M4 C2 M5 Y2
 M2 C5 M1 Y4
 C6 Y3 Y5 C3]

 The positions for C are [(0,0),(1,1),(3,3),(0,3),(2,1),(3,0)]     =>  total 6 posi
tions
 The positions for M are [(2,2),(2,0),(,1,0),(0,1),(1,2)]          =>  total 5 posi
tions
 The positions for Y are [(0,2),(1,3)(3,1)(2,3)(3,2)]              =>  total 5 posi
tions

 Now for a coloured image we have 3 channels
 =>the C dots must be dotted on to the C-channel
 =>the M dots must be dotted on to the M-channel
 =>the Y dots must be dotted on to the Y-channel

 Now we dithering each pixel of the image on iths 4*4 part the number of dots to be
kept will be calculated as follows:
 if a pixel is its CMY values( 178,165,182)
 then the number of  C M Y Ddots to be dotted can be calculated as follows:
```

TRIED IMPLEMENTATION:

In [2]:
```
def color_halftoning(image,dithered_image,C_positions,M_positions,Y_position
s):
    c_len=len(C_positions)
    m_len=len(M_positions)
    y_len=len(Y_positions)

    for i in range(image.shape[0]):
        for j in range(image.shape[1]):

            c_count = ( image[i,j,2]/255)*c_len
            y_count = (image[i,j,0]/255)* m_len
            m_count = (image[i,j,1]/255)*y_len


            x, y = i*4, j*4

            for k in range(0,int(c_count)):
                dithered_image[x+C_positions[k][0],y+C_positions[k][1]] = [2
55,255,0]
            for k in range(0,int(m_count)):
                dithered_image[x+M_positions[k][0],y+M_positions[k][1]] = [0
,255,255]
            for k in range(0,int(y_count)):
                dithered_image[x+Y_positions[k][0],y+Y_positions[k][1]] = [2
55,0,255]


    return dithered_image
```

mask used:

```
[ C M Y C
  M C M Y
  M C M Y
  C Y Y C ]
```

In [3]:

```
def main():

    C_positions=[[0,0],[1,1],[3,3],[0,3],[2,1],[3,0]]
    M_positions=[[2,2],[2,0],[1,0],[0,1],[1,2]]
    Y_positions=[[0,2],[1,3],[3,1],[2,3],[3,2]]

    image = cv2.imread('images/orange-flower.ppm')
    dithered_image = np.full((image.shape[0]*4, image.shape[1]*4, 3), 255, d
type = np.uint8)
    dithered_image=color_halftoning (image,dithered_image,C_positions,M_posi
tions,Y_positions)
    print("running on orange-flower")
    print(image.shape)
    print(dithered_image.shape)
    cv2.imwrite("Results/dithering/orange-flower.ppm", dithered_image)


    image = cv2.imread('images/waterplane.ppm')
    dithered_image = np.full((image.shape[0]*4, image.shape[1]*4, 3), 255, d
type = np.uint8)
    dithered_image=color_halftoning (image,dithered_image,C_positions,M_posi
tions,Y_positions)
    print("running on waterplane")
    print(image.shape)
    print(dithered_image.shape)
    cv2.imwrite("Results/dithering/waterplane.ppm", dithered_image)



    image = cv2.imread('images/fall-colours.jpg')
    dithered_image = np.full((image.shape[0]*4, image.shape[1]*4, 3), 255, d
type = np.uint8)
    dithered_image=color_halftoning (image,dithered_image,C_positions,M_posi
tions,Y_positions)
    print("running on fall-colour")
    print(image.shape)
    print(dithered_image.shape)
    cv2.imwrite("Results/dithering/fall-colours.jpg", dithered_image)



main()
```

```
running on orange-flower
(480, 640, 3)
(1920, 2560, 3)
running on waterplane
(450, 600, 3)
(1800, 2400, 3)
running on fall-colour
(402, 600, 3)
(1608, 2400, 3)
```

Analysis:-

The above algorithm results are DIFFERENT from the actual image,the results are slightly BLUISH as the mask we used results in dotting more blue parts of the RGB channels

2. Try coming up with your own error diffusion coefficients and implement the standard error-diffusion algorithm. Compare the performance of your coefficients against Floyd-Steinberg's on this image. Discuss the patterns visible in yours and in Floyd-Steinberg's at the various gray levels.

In error diffusion algorithm we get the each new pixel value based on a threshold implemented on the previous image, we consider 127 as our threshold limit the pixel value is less than 127 we set the new pixel as 0 and load the pixel value as the error and diffuse it to the next pixels based on the error diffusion criteria we are considering. similarly if the pixel value is greater than the threshold then we set the pixel to 255 and get the difference get pixel and the 255 as negative error and diffuse it.

we floyd's coefficients for diffusion

```
[          *     7/16

3/16     5/16   1/16 ]
```

floyd steinberg algorithm

pixel position [[0,1], [1,1], [1,0], [1,-1]] if current pixel is [0,0]

error diffused: [7 / 16, 1 / 16, 5 / 16, 3 / 16]

floyd steinberg algorithm from wikipedia:

```
   for each y from top to bottom
      for each x from left to right
         oldpixel  := pixel[x][y]
         newpixel  := find_closest_palette_color(oldpixel)
         pixel[x][y]  := newpixel
         quant_error  := oldpixel - newpixel
         pixel[x + 1][y    ] := pixel[x + 1][y    ] + quant_error * 7 / 16
         pixel[x - 1][y + 1] := pixel[x - 1][y + 1] + quant_error * 3 / 16
         pixel[x    ][y + 1] := pixel[x    ][y + 1] + quant_error * 5 / 16
         pixel[x + 1][y + 1] := pixel[x + 1][y + 1] + quant_error * 1 / 16
```

In [4]:
```
def floyd_steinberg(image):
    folyd_image = image.copy()
    total_error=0
    for i in range(0, image.shape[0]-1):
        for j in range(0, image.shape[1]-1):
            error = 0
            if folyd_image[i][j] < 128:
                quant_error = folyd_image[i][j]
                total_error+=quant_error
                folyd_image[i][j] = 0
            else:
                quant_error = folyd_image[i][j]-255
                total_error+=quant_error
                folyd_image[i][j] = 255

            folyd_image[i][j+1] =np.clip(quant_error * 7 / 16+folyd_image[i]
[j+1],0,255)
            folyd_image[i+1][j+1] =np.clip (quant_error * 1 / 16 + folyd_ima
ge[i+1][j+1],0,255)
            folyd_image[i+1][j-1] = np.clip (quant_error * 5 / 16+ folyd_ima
ge[i+1][j-1],0,255)
            folyd_image[i+1][j] = np.clip (quant_error * 3 / 16 + folyd_imag
e[i+1][j] ,0,255)

    print("executing")
    print("total error diffused in floyd's=")
    print(total_error)

    cv2.imwrite("Results/diffussion_results/FLOYDS.png", folyd_image)
```

Diffusion algorithm1 by slightly modifying the floyd's diffusion cofficients

```
[          *     6/16


2/16     6/16   2/16 ]
```

pixel position [[0,1], [1,1], [1,0], [1,-1]] if current pixel is [0,0]

error diffused: [6 / 16, 2 / 16, 6 / 16, 2 / 16]

```
pixel[x + 1][y    ] := pixel[x + 1][y    ] + quant_error * 6 / 16
pixel[x - 1][y + 1] := pixel[x - 1][y + 1] + quant_error * 2 / 16
pixel[x    ][y + 1] := pixel[x    ][y + 1] + quant_error * 6 / 16
pixel[x + 1][y + 1] := pixel[x + 1][y + 1] + quant_error * 2 / 16
```

In [5]:
```
def my_diffusion1(image):
    my_diff = image.copy()
    total_error=0
    for i in range(0, image.shape[0]-1):
        for j in range(0, image.shape[1]-1):
            error = 0
            if my_diff[i][j] < 128:
                quant_error = my_diff[i][j]
                total_error+=quant_error
                my_diff[i][j] = 0
            else:
                quant_error = my_diff[i][j]-255
                total_error+=quant_error
                my_diff[i][j] = 255

            my_diff[i][j+1] =np.clip(quant_error * 6 / 16+my_diff[i][j+1],0,
255)
            my_diff[i+1][j+1] =np.clip (quant_error * 2 / 16 + my_diff[i+1][
j+1],0,255)
            my_diff[i+1][j-1] = np.clip (quant_error * 6 / 16+ my_diff[i+1][
j-1],0,255)
            my_diff[i+1][j] = np.clip (quant_error * 2 / 16 + my_diff[i+1][j
] ,0,255)

    print("executing")
    print("total error diffused in algo-1=")
    print(total_error)

    cv2.imwrite("Results/diffussion_results/MY_DIFF1.png", my_diff)
```

Diffusion algorithm2 by slightly modifying the floyd's diffusion cofficients

```
[          *     8/20


             5/20   7/20 ]
```

pixel position [[0,1], [1,1], [1,0],] if current pixel is [0,0]

error diffused: [8 / 20, 5 / 20, 7 / 20]

```
my_diff[i][j+1] =np.clip(quant_error * 8 / 20+my_diff[i][j+1],0,255)
my_diff[i+1][j+1] =np.clip (quant_error * 5 / 20 + my_diff[i+1][j+1],0,255)
my_diff[i+1][j-1] = np.clip (quant_error * 7 / 20+ my_diff[i+1][j-1],0,255)
```

In [6]:
```
def my_diffusion2(image):
    my_diff = image.copy()
    total_error=0
    for i in range(0, image.shape[0]-1):
        for j in range(0, image.shape[1]-1):
            error = 0
            if my_diff[i][j] < 128:
                quant_error = my_diff[i][j]
                total_error+=quant_error
                my_diff[i][j] = 0
            else:
                quant_error = my_diff[i][j]-255
                total_error+=quant_error
                my_diff[i][j] = 255

            my_diff[i][j+1] =np.clip(quant_error * 8 / 20+my_diff[i][j+1],0,
255)
            my_diff[i+1][j+1] =np.clip (quant_error * 5 / 20 + my_diff[i+1][
j+1],0,255)
            my_diff[i+1][j-1] = np.clip (quant_error * 7 / 20+ my_diff[i+1][
j-1],0,255)
    print("executing")
    print("total error diffused in algo-2=")
    print(total_error)
    cv2.imwrite("Results/diffussion_results/MY_DIFF2.png", my_diff)
```

Diffusion algorithm3 by modifying the coefficients and diffusing the error in all the direction

this algorithm doesnot give correct diffusion of error as the error again gets diffused to the modified pixels

```
/* diagnoly backwards */

my_diff[i-1][j-1] =np.clip(quant_error * 1 / 28+my_diff[i-1][j-1],0,255)
my_diff[i-1][j+1] =np.clip (quant_error * 1 / 28 + my_diff[i-1][j+1],0,255)

/* backwards */

my_diff[i-1][j] = np.clip (quant_error * 4 / 28 + my_diff[i-1][j] ,0,255)
my_diff[i][j-1] = np.clip(quant_error * 4 / 28 + my_diff[i][j-1], 0 ,255)

/* diagonally farwards */

my_diff[i+1][j+1] =np.clip (quant_error * 3 / 28 + my_diff[i+1][j+1],0,255)
my_diff[i+1][j-1] = np.clip (quant_error * 3 / 28+ my_diff[i+1][j-1],0,255)

/*  forward directions*/

my_diff[i+1][j] = np.clip (quant_error * 6 / 28 + my_diff[i+1][j] ,0,255)
my_diff[i][j+1] = np.clip (quant_error * 6 / 28+ my_diff[i][j+1],0,255)
```

In [7]:
```
def my_diffusion3(image):
    my_diff = image.copy()
    total_error=0
    for i in range(0, image.shape[0]-1):
        for j in range(0, image.shape[1]-1):
            error = 0
            if my_diff[i][j] < 128:
                quant_error = my_diff[i][j]
                total_error+=quant_error
                my_diff[i][j] = 0
            else:
                quant_error = my_diff[i][j]-255
                total_error+=quant_error
                my_diff[i][j] = 255

            my_diff[i-1][j-1] =np.clip(quant_error * 1 / 28+my_diff[i-1][j-1
],0,255)
            my_diff[i-1][j] = np.clip (quant_error * 4 / 28 + my_diff[i-1][j
] ,0,255)
            my_diff[i-1][j+1] =np.clip (quant_error * 1 / 28 + my_diff[i-1][
j+1],0,255)
            my_diff[i][j+1] = np.clip (quant_error * 6 / 28+ my_diff[i][j+1]
,0,255)



            my_diff[i+1][j+1] =np.clip (quant_error * 3 / 28 + my_diff[i+1][
j+1],0,255)
            my_diff[i+1][j] = np.clip (quant_error * 6 / 28 + my_diff[i+1][j
] ,0,255)
            my_diff[i+1][j-1] = np.clip (quant_error * 3 / 28+ my_diff[i+1][
j-1],0,255)
            my_diff[i][j-1] =np.clip(quant_error * 4 / 28+my_diff[i][j-1],0,
255)
    print("executing")
    print("total error diffused in algo-3=")
    print(total_error)
    cv2.imwrite("Results/diffussion_results/MY_DIFF3.png", my_diff)
```

In [8]:
```
image = cv2.imread('images/ed-eg.png',0)
floyd_steinberg(image)
my_diffusion1(image)
my_diffusion2(image)
my_diffusion3(image)
```

```
executing
total error diffused in floyd's=
-3077300
executing
total error diffused in algo-1=
-2871158
executing
total error diffused in algo-2=
-3900382
executing
total error diffused in algo-3=
-915518
```

NOTE:

my_diffusion3 algorthms fails as it diffuses the error to the already modified pixel positions hence error can only be diffused to the un modified that is forward pixels

3.(a) Implement an algorithm to simulate the grayscale output from a colour filter array. The function prototype is image colour_filter (image, filter) That is, it takes an input colour image and a colour filter as parameters and returns a grayscale image.

colour_filter

filter used

```
[G ,B
,R ,B]
```

the filter is moved across the image by moving it by 2 pixels each time. this could also be done by moving the filter by 1 position at a time and to match the structure perform np.roll operation on the filter.this algorithm may give better solutions than the implemented algorithm as the operation on pixels is carried out twice.

In [9]:
```
def colour_filter (image, filtter):

    gray_image = np.zeros((image.shape[0],image.shape[1]))

    for i in range(0,image.shape[0]-2, 2):
        for j in range(0,image.shape[1]-2, 2):

                gray_image[i][j]=image[i][j][filtter[0]]
                gray_image[i][j+1]=image[i+1][j+1][filtter[1]]
                gray_image[i+1][j]=image[i+2][j+2][filtter[2]]
                gray_image[i+1][j+1]=image[i+3][j+3][filtter[3]]
    print("executing")


    return np.uint8(gray_image)
```

In [10]:
```
filtter = [1, 2, 0, 1]                #[G ,B ,R ,B]

image = cv2.imread('images/orange-flower.ppm')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gray_image=colour_filter (image, filtter)
cv2.imwrite("Results/colour_filter/orange-flower.ppm", gray_image)

image = cv2.imread('images/waterplane.ppm')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gray_image=colour_filter (image, filtter)
cv2.imwrite("Results/colour_filter/waterplane.ppm", gray_image)

image = cv2.imread('images/fall-colours.jpg')
image = cv2.cvtColor(image, cv2.COLOR_BGR2RGB)
gray_image=colour_filter (image, filtter)
cv2.imwrite("Results/colour_filter/fall-colours.jpg", gray_image)
```

```
executing
executing
executing
```

Out[10]:
```
True
```

(b) Implement a demosaicking algorithm with the prototype image demosaic (image, filter)The input image is a grayscale image output by the colour_filter algorithm and the corresponding filter array; the output is a colour image.

demosaicking the gray image

```
/*
 B G B G
 G R G R
 B G B G
 */

 filter considered
Actual 2*2 filter => B G
                     G R

 roll the filter for every column,row to match the structure.

 for next column => G B   by performing np.roll to match the structure
                    R G

 for next row=>  G R   roll the filter for every column,row to match the struct
ure.
                 B G

 While passing filter on the gray image get the blue components,red and average
of green components as        there are 2 greens,for each pixel pass the filter and
calculate the values & assign these components as the three colour values.
```

In [11]:
```
def demosaic (image, filtter):
    color_image = np.zeros((image.shape[0],image.shape[1],3))
    for i in range(0, image.shape[0]-1, 1):
        for j in range(0, image.shape[1]-1, 1):

#             obtaining the positions of the colour components
            red = np.where(filtter == 0)
            green = np.where(filtter == 1)
            blue = np.where(filtter == 2)

#             red colour components on to red channel
            for x,y in zip(red[0],red[1]):
                color_image[i][j][2] = image[i+x][j+y]

#             green colour components on to green channel
            for x,y in zip(green[0],green[1]):
                color_image[i][j][1] = color_image[i][j][1]+image[i+x][j+y]
            color_image[i][j][1] = color_image[i][j][1] / 2

#             blue colour components on to blue channel
            for x,y in zip(blue[0],blue[1]):
                color_image[i][j][0] = image[i+x][j+y]

#         roll the filter to match the next column structure
            filtter = np.roll(filtter, shift=1, axis=1)

#         roll the filter to match the next rowstructure
        filtter = np.roll(filtter, shift=1, axis=1)
        filtter = np.roll(filtter, shift=1, axis=0)
    print("executing demosaic")

    return np.uint8(color_image)
```

In [12]:
```
filtter = np.array([[1,2],[0,1]])

image = cv2.imread('Results/colour_filter/orange-flower.ppm',0)
color_image=demosaic (image, filtter)
cv2.imwrite("Results/demosaic/orange-flower.ppm",color_image)

image = cv2.imread('Results/colour_filter/fall-colours.jpg',0)
color_image=demosaic (image, filtter)
cv2.imwrite("Results/demosaic/fall-colours.jpg", color_image)

image = cv2.imread('Results/colour_filter/waterplane.ppm',0)
color_image=demosaic (image, filtter)
cv2.imwrite("Results/demosaic/waterplane.ppm", color_image)
```

```
executing demosaic
executing demosaic
executing demosaic
```

Out[12]: True

In [ ]: