National Taiwan University of Science and Technology

2020 Multimedia Signal Processing

Homework 1

Name: 王潔汝 Student ID: M10907316

Advisor: Prof. Jing-Ming Guo

2020 年 11 月 04 日

Content

Chapter 1. **Ordered Dithering**

Section 1.1 Problem

Topic: Point Process-Ordered Dithering using the Classical-4 & Bayer-5 Dither Array. Write an algorithm to convert the Gray Scale Image (0-255 Range) to Binary Image (0-1 Range) using the mentioned dither array.

Section 1.2 Method

1. Input RGB image to Gray.
2. **Classical-4** & **Bayer-5** Dither Array (both of them are 8x8 pattern).
3. All pixels on the Dither Array need to multiply 255.
4. Compare every pixel of the 8x8 pattern with 8x8 block.

$$if \begin{Bmatrix} pixel\ on\ pattern\ > pixel\ on\ block\ , output = 0 \\ pixel\ on\ pattern\ < pixel\ on\ block\ , output = 255 \end{Bmatrix}$$

0.567	0.635	0.608	0.514	0.424	0.365	0.392	0.486
0.847	0.878	0.910	0.698	0.153	0.122	0.090	0.302
0.820	0.969	0.941	0.667	0.180	0.031	0.059	0.333
0.725	0.788	0.757	0.545	0.275	0.212	0.243	0.455
0.424	0.365	0.392	0.486	0.567	0.635	0.608	0.514
0.153	0.122	0.090	0.302	0.847	0.878	0.910	0.698
0.180	0.031	0.059	0.333	0.820	0.969	0.941	0.667
0.275	0.212	0.243	0.455	0.725	0.788	0.757	0.545

(a) Classical-4

0.513	0.272	0.724	0.483	0.543	0.302	0.694	0.453
0.151	0.755	0.091	0.966	0.181	0.758	0.121	0.936
0.634	0.392	0.574	0.332	0.664	0.423	0.604	0.362
0.060	0.875	0.211	0.815	0.030	0.906	0.241	0.845
0.543	0.302	0.694	0.453	0.513	0.272	0.724	0.483
0.181	0.758	0.121	0.936	0.151	0.755	0.091	0.966
0.664	0.423	0.604	0.362	0.634	0.392	0.574	0.332
0.030	0.906	0.241	0.845	0.060	0.875	0.211	0.815

(b) Bayer-5

Figure 1. The dither array of (a) Classical-4 and (b) Bayer-5.

Section 1.3 Code

1.**Classical-4**

OD_Classical-4.py > ...

```python
from cv2 import cv2 as cv2
import numpy as np

img = cv2.imread('lena.jpg',0)
h1,w1 = img.shape


imgnews=np.empty((512,512))
DA=np.array([[0.567 ,0.635, 0.608, 0.514, 0.424, 0.365, 0.392, 0.486],
    [0.847, 0.878, 0.910, 0.698, 0.153, 0.122, 0.090, 0.302],
    [0.820, 0.969, 0.941, 0.667, 0.180, 0.031, 0.059, 0.333],
    [0.725, 0.788, 0.757, 0.545, 0.275, 0.212, 0.243, 0.455],
    [0.424, 0.365, 0.392, 0.486, 0.567, 0.635, 0.608, 0.514],
    [0.153, 0.122, 0.090, 0.302, 0.847, 0.878, 0.910, 0.698],
    [0.180, 0.031, 0.059, 0.333, 0.820, 0.969, 0.941, 0.667],
    [0.275, 0.212, 0.243, 0.455, 0.725, 0.788, 0.757, 0.545]])
h2,w2 = DA.shape
DA=DA*255
img = img.astype(np.float)
for i in range(0,h1):
    for j in range(0,w1):
        a = i % 8
        b = j % 8
        if img[i][j]>DA[a][b]:
            imgnews[i][j]=255
        else:
            imgnews[i][j]=0
cv2.imwrite('OD_Classical-4.jpg', imgnews)
cv2.imshow('Classical-4',imgnews)
cv2.waitKey(0)
```

Section 1.3 Code

2.Bayer-5

```python
from cv2 import cv2 as cv2
import numpy as np


img = cv2.imread('lena.jpg',0)
h1,w1 = img.shape


imgnews=np.empty((512,512))
DA=np.array([[0.513, 0.272, 0.724, 0.483, 0.543, 0.302, 0.694, 0.453],
    [0.151, 0.755, 0.091, 0.966, 0.181, 0.758, 0.121, 0.936],
    [0.634, 0.392, 0.574, 0.332, 0.664, 0.423, 0.604, 0.362],
    [0.060, 0.875, 0.211, 0.815, 0.030, 0.906, 0.241, 0.845],
    [0.543, 0.302, 0.694, 0.453, 0.513, 0.272, 0.724, 0.483],
    [0.181, 0.758, 0.121, 0.936, 0.151, 0.755, 0.091, 0.966],
    [0.664, 0.423, 0.604, 0.362, 0.634, 0.392, 0.574, 0.332],
    [0.030, 0.906, 0.241, 0.845, 0.060, 0.875, 0.211, 0.815]])
h2,w2 = DA.shape
DA=DA*255
img = img.astype(np.float)
for i in range(0,h1):
    for j in range(0,w1):
        a = i % 8
        b = j % 8
        if img[i][j]>DA[a][b]:
            imgnews[i][j]=255
        else:
            imgnews[i][j]=0
cv2.imwrite('OD_Bayer-5.jpg', imgnews)
cv2.imshow('Bayer-5',imgnews)
cv2.waitKey(0)
```

Section 1.4 Result

1. **Classical-4**

(a) The original image

(b) The result after using Classical-4 filter

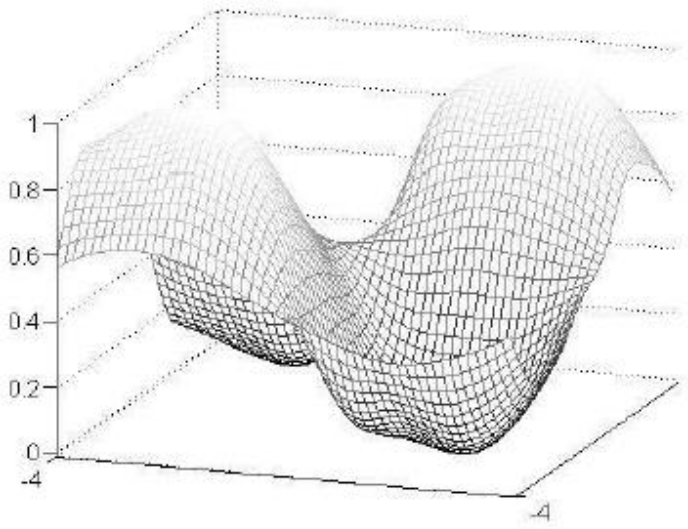
2. **Bayer-5**

(a) The original image

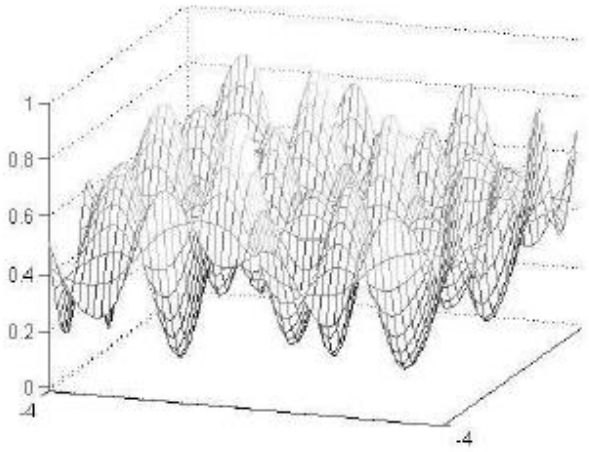
(b) The result after using Bayer-5 filter

Section 1.5 Inference/Discussion on results

As a result of the experiments between Figures 5 and 6, the output images are quite different by using Classical-4 and Bayer-5 filter. As can be seen from the Classical-4 matrix in Figure 7 (a), its waveform obviously has two peaks and two valleys. This feature that causes the black-and-white dots to have a gathering effect when the grayscale image is turned into a half-tone image, called point aggregation (clustered-dot ding). However, the Bayer-5 matrix in Figure 7 (b) can clearly see that multiple peaks and valleys are closely intertwined together. It means that the black-and-white dots are scattered apart, known as point diffusion (dispersed-dot dithering).



(a) Classical-4



(b) Bayer-5

Chapter 2. **Error Diffusion**

Section 2.1 Problem

In error diffusion three kernels are widely used Stucki (1981), Jarvis (1976), Floyd-Steinberg (1975) Write an algorithm to convert the Gray Scale Image (0-255 Range) to Binary Image (0-1 Range) based on the mentioned error diffusion kernels.

Section 2.2 Method

1. Input RGB image to Gray.
2. We use Floyd-Steinberg、Jarvis、Stucki.

	(i,j)	7/16
3/16	5/16	1/16

(a) Floyd-Steinberg

		(i,j)	7/48	5/48
3/48	5/48	7/48	5/48	3/48
1/48	3/48	5/48	3/48	1/48

(b) Jarvis

		(i,j)	8/42	4/42
2/42	4/42	8/42	4/42	2/42
1/42	2/42	4/42	2/42	1/42

(c) Stucki

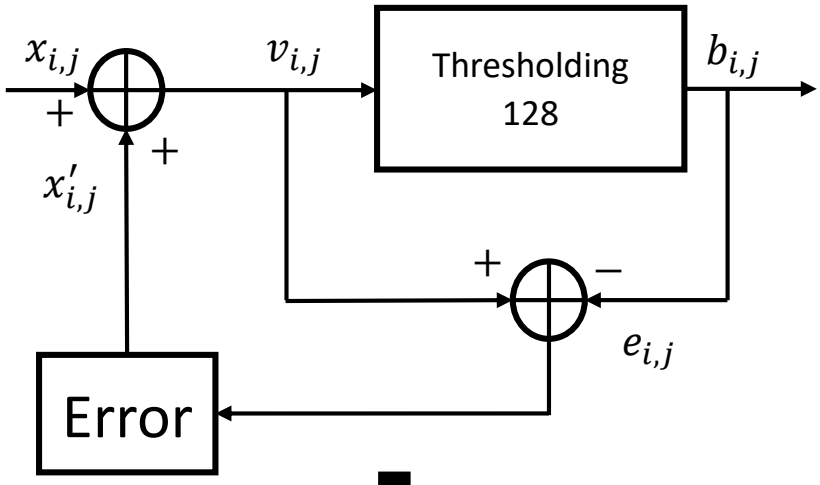
3. Start from the first pixel (0,0) , one by one to calculate every pixel,and the rules as follows:

first pixel (0,0)

next pixel

13	164	212	98
67	138	230	20
149	93	246	56

13<128, so 13 become 0,the error=13-0=13, then put the error to neighborhood and follow the patterns of Floyd-Steinberg.



0	164+13*7/16	212	98
67+13*5/16	138+13*1/16	230	20
149	93	246	56

Section 2.3 Code

1. Floyd-Steinberg

ED_Floyd-steinberg.py > ...

```python
from cv2 import cv2 as cv2
import numpy as np
import copy
import math
img = cv2.imread('lena.bmp',0)
h1,w1 = img.shape
img = img.astype(np.float)

for i in range(1,h1-1,1):
    for j in range(1,w1-1,1):
        #print(i,j)
        if img[i][j]>=128:
            dummy=255
        else:
            dummy=0
        error = img[i][j] - dummy
        #error = math.fabs(error)
        img[i][j] = dummy
        if (i<h1-1) and (j==0):
            img[i+1][j] = img[i+1][j] + error*5/13
            img[i][j+1] = img[i][j+1] + error*7/13
            img[i+1][j+1] = img[i+1][j+1] + error*1/13
        elif (i<h1-1) and (j==w1-1):
            img[i+1][j] = img[i+1][j] + error*5/8
            img[i+1][j-1] = img[i+1][j-1] + error*3/8
        elif (i==h1-1) and (j<w1-1):
            img[i][j+1] = img[i][j+1] + error*7/7
        elif (i<h1-1) and (j>0) and (j<w1-1):
            img[i+1][j] = img[i+1][j] + error*5/16
            img[i][j+1] = img[i][j+1] + error*7/16
            img[i+1][j+1] = img[i+1][j+1] + error*1/16
            img[i+1][j-1] = img[i+1][j-1] + error*3/16
        else:
            img[i][j]==img[i][j]
cv2.imwrite('ED_Floyd-steinberg.jpg', img*255)
cv2.imshow('ED',img)

cv2.waitKey(0)
```

2. Jarvis

ED_Jarvis.py > ...

```python
from cv2 import cv2 as cv2
import numpy as np
import copy
import math
img = cv2.imread('lena.bmp',0)
h1,w1 = img.shape
img = img.astype(np.float)

kernel = np.array([[0,0,0,7,5],
                   [3,5,7,5,3],
                   [1,3,5,3,1]])

for i in range(0,h1-1,1):
    for j in range(0,w1-1,1):
        #print(i,j)
        if img[i][j]>=128:
            dummy=255
        else:
            dummy=0
        error = img[i][j] - dummy
        #error = math.fabs(error)
        img[i][j] = dummy
        kernel_sum = 0
        for x in range(0,kernel.shape[0]):
            for y in range(0,kernel.shape[1]):
                if (i+x>=0) and (j+y>=2) and (i+x<h1) and (j+y<w1):
                    kernel_sum+=kernel[x][y]

        for x in range(0,kernel.shape[0]):
            for y in range(0,kernel.shape[1]):
                if (i+x>=0) and (j+y>=2) and (i+x<h1) and (j+y<w1):
                    img[i+x][j+y-2] = img[i+x][j+y-2] + (error*kernel[x][y]/kernel_sum)


cv2.imwrite('ED_Jarvis.jpg', img)
cv2.imshow('ED',img)
cv2.waitKey(0)
```

3. Stucki

ED_Stucki.py > ...

```python
from cv2 import cv2 as cv2
import numpy as np
import copy
import math
img = cv2.imread('lena.bmp',0)
h1,w1 = img.shape
img = img.astype(np.float)

kernel = np.array([[0,0,0,8,4],
                   [2,4,8,4,2],
                   [1,2,4,2,1]])

for i in range(0,h1-1,1):
    for j in range(0,w1-1,1):
        #print(i,j)
        if img[i][j]>=128:
            dummy=255
        else:
            dummy=0
        error = img[i][j] - dummy
        #error = math.fabs(error)
        img[i][j] = dummy
        kernel_sum = 0
        for x in range(0,kernel.shape[0]):
            for y in range(0,kernel.shape[1]):
                if (i+x>=0) and (j+y>=2) and (i+x<h1) and (j+y<w1):
                    kernel_sum+=kernel[x][y]

        for x in range(0,kernel.shape[0]):
            for y in range(0,kernel.shape[1]):
                if (i+x>=0) and (j+y>=2) and (i+x<h1) and (j+y<w1):
                    img[i+x][j+y-2] = img[i+x][j+y-2] + (error*kernel[x][y]/kernel_sum)

cv2.imwrite('ED_Stucki.jpg', img)
cv2.imshow('ED',img)
cv2.waitKey(0)
```

Section 2.4 Results

1. Floyd-Steinberg

(a) The original image

(b) The result after using Floyd-Steinberg

2. Jarvis

(a) The original image

(b) The result after using Jarvis

3. Stucki

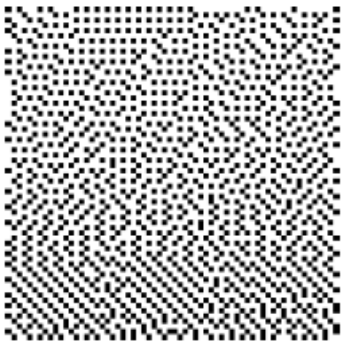
(a) The original image

(b) The result after using Stucki

Section 2.5 Inference/Discussion on results

Due to the halftone image resulting from the error diffusion weight of Floyd-Steinberg, the sampling error between the original image and the halftone image can be effectively enhanced the image quality. However, there are still some unnatural texture patterns, such as the worm effect or linear pattern, as shown in Figure 16. Therefore, Jarvis and Stucki proposed different error matrix to solve these problems.

(a) Worm effect



(b) Linear pattern

Additional Bonus:

Section 1.1 Problem

Implement Dot-diffusion halftones

Section 1.2 Method

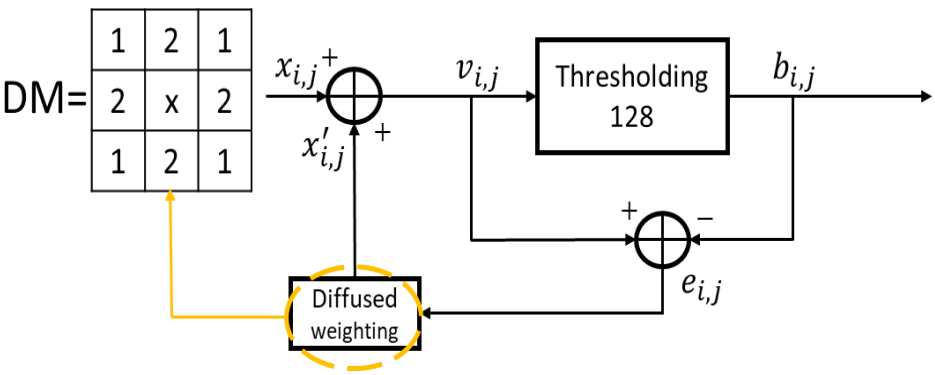
1. input RGB image to Gray, split the gray image into every 8x8 block.

2. sort the number which is on the pattern(Class Matrix,CM).

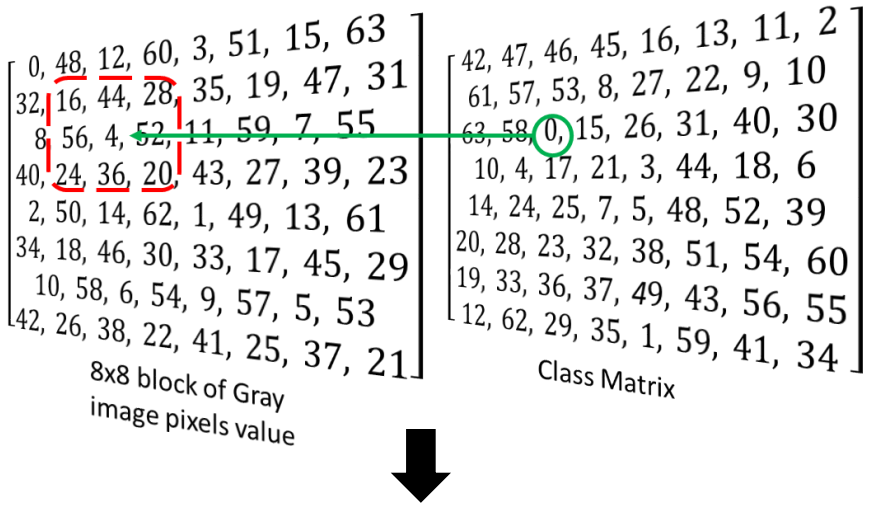
$$CM=\begin{bmatrix} 42, 47, 46, 45, 16, 13, 11, 2 \\ 61, 57, 53, 8, 27, 22, 9, 10 \\ 63, 58, 0, 15, 26, 31, 40, 30 \\ 10, 4, 17, 21, 3, 44, 18, 6 \\ 14, 24, 25, 7, 5, 48, 52, 39 \\ 20, 28, 23, 32, 38, 51, 54, 60 \\ 19, 33, 36, 37, 49, 43, 56, 55 \\ 12, 62, 29, 35, 1, 59, 41, 34 \end{bmatrix}$$

Start from 0,1,2,3…61,62,63, and remember each number's coordinates (x,y) in order to decide image processing sequence. So we also split the gray image into every 8x8 block, then we could begin to do all the pixels of every 8x8 block.

3. use weight (Diffused Matrix, DM), and the rules as follows:



4.How the weight(Diffused Matrix, DM) work ? For example：



Get the “error value”(pixels-output=error)

$$if\ pixel\ on \begin{cases} 8x8\ gray\ image < 128, output = 0 \\ 8x8\ gray\ image > 128, output = 255 \end{cases}$$

16	44	28
56	4	52
24	36	20

The first one block to process (start from 0)

16	44	28
56	0	52
24	36	20

4<255,become 0,error=4, then put the error to neighborhood

Apply Diffusion matrix

DM=

1	2	1
2	x	2
1	2	1

W=2+2+2+2+2+1+1+1+1=12

16+4*1/12	44+4*2/12	28+4*1/12
56+4*2/12	0	52+4*2/12
24+4*1/12	36+4*2/12	20+4*1/12

Section 1.3 Code

Dot_Diffusion.py > ...

```python
from cv2 import cv2 as cv2
import numpy as np
from matplotlib import pyplot as plt
import pandas as pd
import math

img = cv2.imread('lena.jpg',0)
height, width = img.shape[:2]
threshold = 128

imgnew = np.empty((height,width))
pattern = np.array([[42,47,46,45,16,13,11,2],
    [61,57,53,8,27,22,9,50],
    [63,58,0,15,26,31,40,30],
    [10,4,17,21,3,44,18,6],
    [14,24,25,7,5,48,52,39],
    [20,28,23,32,38,51,54,60],
    [19,33,36,37,49,43,56,55],
    [12,62,29,35,1,59,41,34]])

patMaps = np.zeros((height,width))

pheight, pwidth = pattern.shape[:2]

totheight = int(height / pheight)

totwidth = int(width / pwidth)

patProc = np.empty((pheight*pwidth,2))

for m in range(0,pheight):
    for n in range(0,pwidth):
        patProc[pattern[m][n]][0] = m

        patProc[pattern[m][n]][1] = n

img = img.astype(float)
pattern = pattern.astype(float)

```

Dot_Diffusion.py > {} math

```
pattern = pattern.astype(float)
weight = np.array([[0.271630,1,0.271630],
                   [1,0,1],
                   [0.271630,1,0.271630]])
dumArray = np.copy(img)
for i in range(0,height,pheight):
    for j in range(0,width,pwidth):
        #print(i,j)
        index = 0
        while index != (pheight * pwidth):
            ni = int(i + patProc[index][0])
            nj = int(j + patProc[index][1])

            if dumArray[ni][nj] > threshold:
                dummy = 255
            else:
                dummy = 0
            error = dumArray[ni][nj] - dummy
            img[ni][nj] = dummy
            patMaps[ni][nj] = 1

            fm = 0
            for m in range(-1,2):
                for n in range(-1,2):
                    if (ni+m >= 0) and (ni+m < height) and (nj+n >= 0) and (nj+n < width):
                        if patMaps[ni+m][nj+n] == 0:
                            fm = fm + weight[m+1][n+1]

            for m in range(-1,2):
                for n in range(-1,2):
                    if (ni+m >= 0) and (ni+m < height) and (nj+n >= 0) and (nj+n < width):
                        if patMaps[ni+m][nj+n] == 0:
                            dumArray[ni+m][nj+n] = dumArray[ni+m][nj+n] + (error * weight[m+1][n+1] / fm)

            index = index + 1
cv2.imwrite('Dot_Diffusion.jpg',img)
cv2.imshow("DD",img)
cv2.waitKey(0)
cv2.destroyAllWindows()
```

Section 1.4 Results

(a) The original image

(b) The result after using Dot Diffusion