Digital Signal Processing I
Purdue University
Spring 2021

Lab 8 Report

Jerry Wang

1 Thresholding and Random Noise Binarization

1. Hand in the original image and the result of thresholding

Figure 1: Original Image

Figure 2: Thresholding Image

2. The computed RMSE and fidelity values

RMSE	87.393
Fidelity	77.337

Table 1: RMSE and fidelity of the image

3. Code for the *fidelity* function

```
# %% filter function
def apply_filter_at_pixel(img, i, j, kernel):
    kernal_size = kernel.shape[0]
    pixel = 0.0
    for k in range(kernel.shape[0]):
        for l in range(kernel.shape[1]):
            loc_i = i + k - kernal_size // 2
            loc_j = j + l - kernal_size // 2
            if loc_i >= 0 and loc_i < img.shape[0] and
                loc_j >= 0 and loc_j < img.shape[1]:
                pixel += kernel[k, l] * img[loc_i, loc_j]
    return pixel

# %% FIR Filter
def apply_filter(img, kernel):
    img_filter = np.zeros_like(img)
    for i in range(img.shape[0]):
        for j in range(img.shape[1]):
            img_filter[i, j] = apply_filter_at_pixel(img, i, j, kernel)
    return img_filter
```

```
# %% Fidelity Function
def fidelity(f, b):
    # Low pass filter f and b
    kernel = np.zeros((7, 7)).astype(np.float)
    for i in range(7):
        for j in range(7):
            kernel[i, j] = math.exp(-((i-3) ** 2 + (j-3) ** 2) / 4.)
    kernel = kernel / np.sum(kernel)
    f_filter = apply_filter(f, kernel)
    b_filter = apply_filter(b, kernel)
    # apply transformation
    f_filter = 255. * (f_filter / 255.) ** (1. / 3.)
    b_filter = 255. * (b_filter / 255.) ** (1. / 3.)
    return np.sqrt(np.sum(np.power(f_filter - b_filter, 2)) /
                        np.prod(f_filter.shape))
```

2 Ordered Dithering

1. The three Bayer Index Matrix

$$I_2 = \begin{bmatrix} 95.625 & 159.375 \\ 223.125 & 31.875 \end{bmatrix}$$

$$I_4 = \begin{bmatrix} 87.65625 & 151.40625 & 103.59375 & 167.34375 \\ 215.15625 & 23.90625 & 231.09375 & 39.84375 \\ 119.53125 & 183.28125 & 71.71875 & 135.46875 \\ 247.03125 & 55.78125 & 199.21875 & 7.96875 \end{bmatrix}$$

$$I_8 = \begin{bmatrix} 85.664 & 149.414 & 101.601 & 165.351 & 89.648 & 153.398 & 105.585 & 169.335 \\ 213.164 & 21.914 & 229.101 & 37.851 & 217.148 & 25.898 & 233.085 & 41.835 \\ 117.539 & 181.289 & 69.726 & 133.476 & 121.523 & 185.273 & 73.710 & 137.460 \\ 245.039 & 53.789 & 197.226 & 5.976 & 249.023 & 57.773 & 201.210 & 9.960 \\ 93.632 & 157.382 & 109.570 & 173.320 & 81.679 & 145.429 & 97.617 & 161.367 \\ 221.132 & 29.882 & 237.070 & 45.820 & 209.179 & 17.929 & 225.117 & 33.867 \\ 125.507 & 189.257 & 77.695 & 141.445 & 113.554 & 177.304 & 65.742 & 129.492 \\ 253.007 & 61.757 & 205.195 & 13.945 & 241.054 & 49.804 & 193.242 & 1.992 \end{bmatrix}$$

2. The halftoned images produced by the three dither patterns

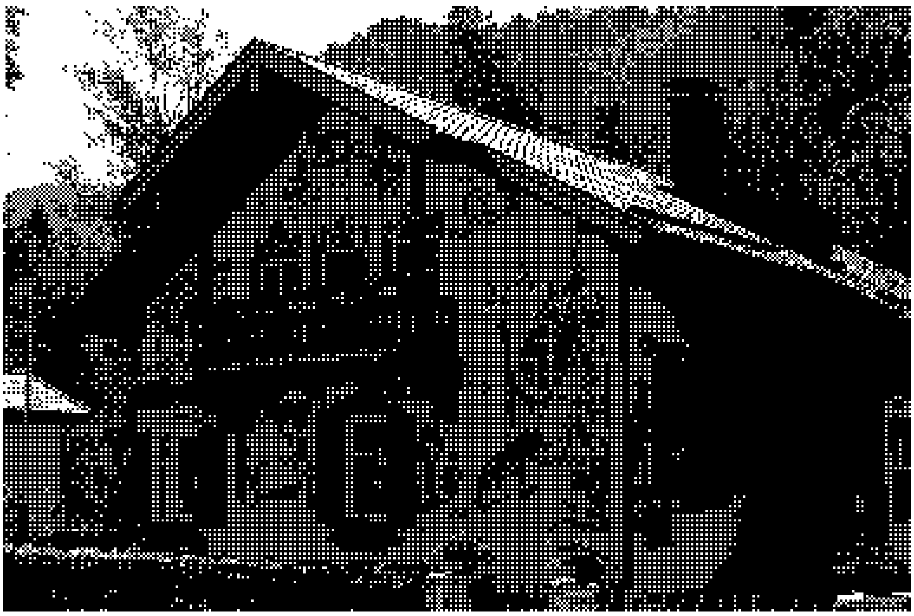

Figure 3: Halftoned Image with Bayer index matrices of size 2×2

Figure 4: Halftoned Image with Bayer index matrices of size 4×4

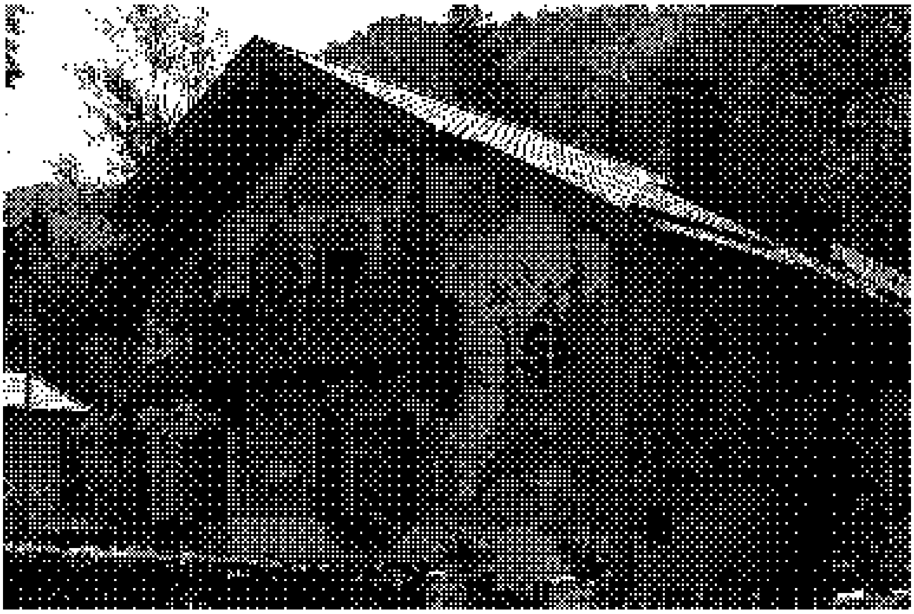

Figure 5: Halftoned Image with Bayer index matrices of size 8×8

3. The RMSE and fidelity for each of the three halftoned images.

Matrix Size	RMSE	Fidelity
2	97.669	50.057
4	101.007	16.558
8	100.915	14.692

Table 2: RMSE and Fidelity for each of the three halftoned images

3 Error Diffusion

1. Error Diffusion Python code.

```
# %% Error Diffusion
out_path = '/home/jerry/Documents/Github/ECE637/Lab8/'
T = 127
img_correct = 255. * np.power(img / 255., 2.2)
output_img = np.zeros_like(img_correct)
for i in range(output_img.shape[0]):
    for j in range(output_img.shape[1]):
        if (img_correct[i, j] > T):
            output_img[i, j] = 255.
        else:
            output_img[i, j] = 0.

        error = img_correct[i, j] - output_img[i][j]
        if (j + 1 < img.shape[1]):
            img_correct[i][j+1] += 7. / 16. * error

        if (i + 1 < img.shape[0] and j + 1 < img.shape[1]):
            img_correct[i+1][j+1] += error / 16.

        if (i + 1 < img.shape[0]):
            img_correct[i+1][j] += 5. / 16. * error

        if (i + 1 < img.shape[0] and j - 1 >= 0):
            img_correct[i+1][j-1] += 3. / 16. * error
RSME = np.sqrt(np.sum(np.power(img - output_img, 2)
                      ) / np.prod(img.shape))
print('Error Diffusion')
print('RSME: ', RSME)
img_correct = 255. * np.power(img / 255., 2.2)
fid = fidelity(img_correct, output_img)
print('fidelity: ', fid)
img_out = Image.fromarray(output_img.astype(np.uint8))
img_out.save(out_path + 'part5_out.tif')
```

2. The error diffusion result

Figure 6: Error Diffusion Result

3. The RMSE and fidelity of the error diffusion result.

RMSE	98.847
Fidelity	13.427

Table 3: RMSE and fidelity of the image

4. Tabulate the RMSE and fidelity for all methods.

Method	RMSE	Fidelity
Simple Thresholding	87.393	77.337
Ordered Dithering 2×2	97.669	50.057
Ordered Dithering 4×4	101.007	16.558
Ordered Dithering 8×8	100.915	14.692
Error Diffusion	98.847	13.427

Table 4: RMSE and Fidelity for each of the three halftoned images

From the halftoned images, the images are progressively looking closer to the original image. From the result table, the fidelity value reflect that change, the fidelity value decreases from simple thresholding to error diffusion. On the other hand, the RMSE value does not change. This shows that fidelity value is a better representation of the how close the image match with each other comparing with RMSE.