Lab 8: Image Halftoning

Course Title: Image Processing I (Spring 2022)

Course Number: ECE 63700

Instructor: Prof. Charles A. Bouman

Author: **Zhankun Luo**

3. Thresholding and Random Noise Binarization

3.1. the original image `house.tif` and the result of thresholding

the Original Image house.tif

Result of Thresholding for the Image house.tif

3.2. the computed RMSE and fidelity values

RMSE	FIDELITY
87.39	77.46

3.3. the Python code for `fidelity()` function

```
from numpy import ndarray, square, sqrt, cbrt, vectorize, \
    zeros, exp
import numpy as np
MAX = 255

def fidelity(f: ndarray, b: ndarray,
            gamma: float=2.2,
            size_kernel: int=7, square_sigma: float=2) -> float:
    if f.dtype == np.uint8 or b.dtype == np.uint8:
        f, b = f.clip(0, MAX).astype(np.double), \
                b.clip(0, MAX).astype(np.double)
    f, b = undo_gamma(f, gamma), undo_gamma(b, gamma)
    kernel = kernel_Gauss(size_kernel, square_sigma)
    f, b = filter_FIR(f, kernel), filter_FIR(b, kernel)
    f, b = cuberoot(f), cuberoot(b)
    return RMSE(f, b)

def RMSE(f: ndarray, b: ndarray) -> float:
    H, W = f.shape
    if f.dtype == np.uint8 or b.dtype == np.uint8:
        f, b = f.clip(0, MAX).astype(np.double), \
                b.clip(0, MAX).astype(np.double)
    return sqrt( square(f - b).sum() / (H*W) )

def undo_gamma(x: ndarray, gamma: float) -> ndarray:
    func = lambda t: MAX*pow(t/MAX, gamma)
    f = vectorize(func)
    return f(x)

def cuberoot(x: ndarray) -> ndarray:
    return MAX*cbrt(x/MAX)

def kernel_Gauss(size_kernel: int, square_sigma: float) -> ndarray:
    assert size_kernel%2 == 1 and square_sigma > 0
    dx = size_kernel//2
    sq = square(range(-dx, dx+1)).reshape([size_kernel, 1])
    sq = sq + sq.T
    kernel = exp( -sq/(2*square_sigma) )
    kernel /= kernel.sum()
```

```
    return kernel

def filter_FIR(x: ndarray, kernel: ndarray) -> ndarray:
    height, width = x.shape
    ky, kx = kernel.shape[0], kernel.shape[-1]
    assert kx%2 == 1 and ky%2 == 1
    dy, dx = ky//2, kx//2
    x_out = zeros((height, width))
    x_out[:dy, :]   = x[:dx, :]
    x_out[-dy:, :]  = x[-dy:, :]
    x_out[:, :dx]   = x[:, :dx]
    x_out[:, :-dx]  = x[:, :-dx]
    for i in range(dy, height-dy):
        for j in range(dx, width-dx):
            x_out[i][j] = (x[i-dy:i-dy+ky, j-dx:j-dx+kx] \
                           * kernel).sum()
    return x_out
```

4. Ordered Dithering

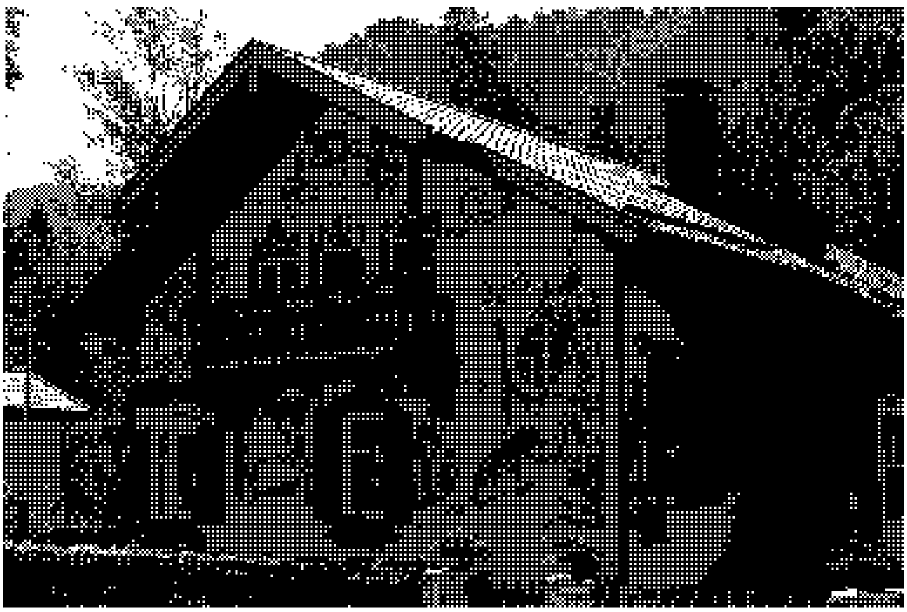
4.1. the three Bayer index matrices of sizes $2 \times 2, 4 \times 4, 8 \times 8$

$$I_2 = \begin{bmatrix} 1 & 2 \\ 3 & 0 \end{bmatrix}$$

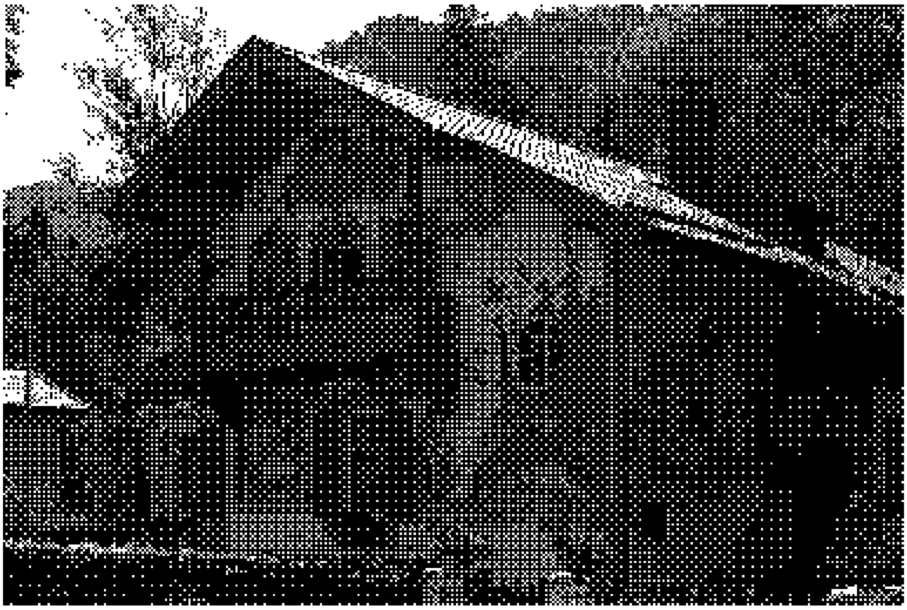
$$I_4 = \begin{bmatrix} 5 & 9 & 6 & 10 \\ 13 & 1 & 14 & 2 \\ 7 & 11 & 4 & 8 \\ 15 & 3 & 12 & 0 \end{bmatrix}$$

$$I_8 = \begin{bmatrix} 21 & 37 & 25 & 41 & 22 & 38 & 26 & 42 \\ 53 & 5 & 57 & 9 & 54 & 6 & 58 & 10 \\ 29 & 45 & 17 & 33 & 30 & 46 & 18 & 34 \\ 61 & 13 & 49 & 1 & 62 & 14 & 50 & 2 \\ 23 & 39 & 27 & 43 & 20 & 36 & 24 & 40 \\ 55 & 7 & 59 & 11 & 52 & 4 & 56 & 8 \\ 31 & 47 & 19 & 35 & 28 & 44 & 16 & 32 \\ 63 & 15 & 51 & 3 & 60 & 12 & 48 & 0 \end{bmatrix}$$

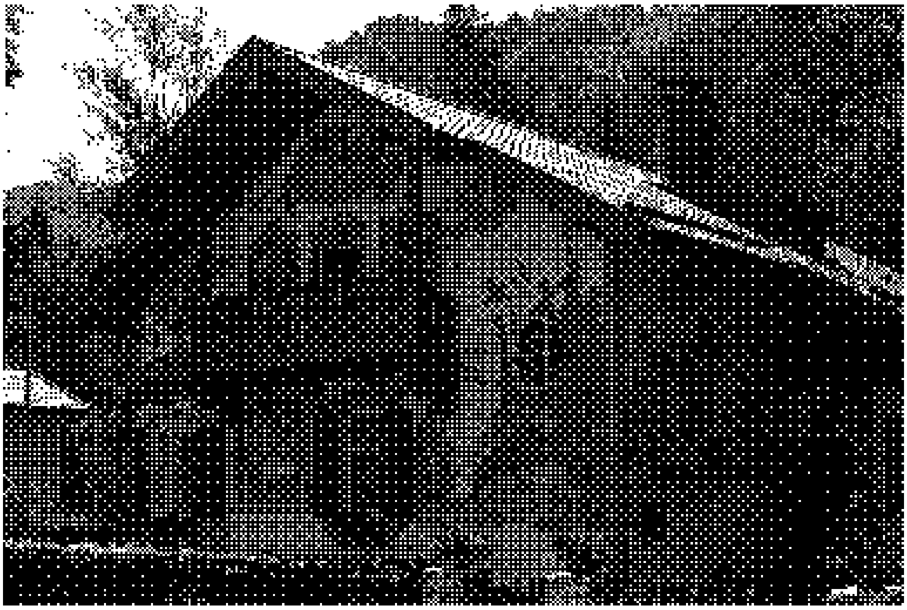
4.2. the three halftoned images produced by the three dither patterns



Result of Ordered Dithering with Patern Size 2x2 for the Image house.tif



Result of Ordered Dithering with Patern Size 4x4 for the Image house.tif



Result of Ordered Dithering with Patern Size 8x8 for the Image house.tif

4.3. the RMSE and fidelity for each of the three halftoned images

PATTERN SIZE	RMSE	FIDELITY
2×2	97.67	50.19
4×4	101.01	16.83
8×8	100.91	15.00

5. Error Diffusion

5.1. the error diffusion Python code

the input arguments of the function `diffuse_error()`

- `x`: the linear-scale version for input image `f`, such as `house.tif`, after undoing the gamma correction $x := 255 \cdot \left(\frac{f}{255}\right)^{\gamma}$ (typically $\gamma = 2.2$)
- `T`: the threshold used for generating the binary image $b := \begin{cases} 255 & \text{if } \hat{f} > T \\ 0 & \text{otherwise} \end{cases}$

```python
from numpy import ndarray, zeros
MAX = 255
def diffuse_error(x: ndarray, T: int) -> ndarray:
    g1_1, g10, g11 = 3/16, 5/16, 1/16
    g01 = 7/16
    H, W = x.shape
    reg, e, b = zeros(W), zeros(x.shape), zeros(x.shape)
    b[0][0] = MAX*(x[0][0] > T)
    e[0][0] = x[0][0] - b[0][0]
    for col in range(1, W):
        e[0, col] = x[0, col] + g01 * e[0, col-1]
        b[0, col] = MAX*(e[0, col] > T)
        e[0, col]-= b[0, col]
    for row in range(1, H):
        reg[1:-1] = g1_1 * e[row-1, 2:] \
            + g10 * e[row-1, 1:-1] + g11 * e[row-1, :-2]
        reg[-1]   = g10  * e[row-1, -1] + g11 * e[row-1, -2]
        e[row, 0] = x[row, 0] \
            + g10 * e[row-1, 0] + g1_1 * e[row-1, 1]
        b[row, 0] = MAX*(e[row, 0] > T)
        e[row, 0]-= b[row, 0]
        for col in range(1, W):
            e[row, col] = x[row, col] \
                + g01 * e[row, col-1] + reg[col]
            b[row, col] = MAX*(e[row, col] > T)
            e[row, col]-= b[row, col]
    return b
```

where $\hat{f}(i,j), e(i,j)$ are sequentially defined by

$$\begin{aligned}\hat{f}(i,j) &:= x(i,j) \\ &\quad + [g_{1,-1}\ g_{1,0}\ g_{1,1}][e(i-1,j+1)\ e(i-1,j)\ e(i-1,j-1)]^\top \\ &\quad + g_{0,1} \cdot e(i,j-1) \\ e(i,j) &:= \hat{f}(i,j) - b(i,j) = \begin{cases} \hat{f}(i,j) - 255 & \text{if } \hat{f} > T \\ \hat{f}(i,j) & \text{otherwise} \end{cases}\end{aligned}$$

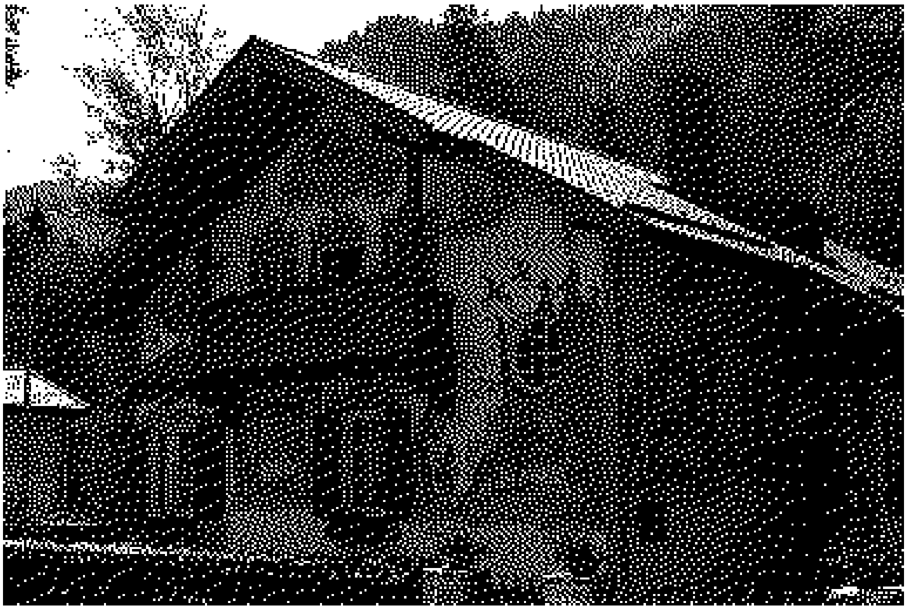
and the error diffusion filter proposed by Floyd and Steinberg is
$[g_{1,-1}\ g_{1,0}\ g_{1,1}] = [\frac{3}{16}, \frac{5}{16}, \frac{1}{16}], g_{0,1} = \frac{7}{16}$

check for more details in paper **Optimized Error Diffusion for Image Display**

https://engineering.purdue.edu/~bouman/publications/orig-pdf/jei1.pdf

or https://engineering.purdue.edu/~bouman/publications/pdf/jei1.pdf

5.2. the error diffusion result



Result of Error Diffusion for the Image house.tif

5.3. the RMSE and fidelity of the error diffusion result

RMSE	FIDELITY
98.85	13.70

5.4. comment on the observations of both the RMSE and fidelity for the different methods

The **Order Dithering** with 8×8 dithering pattern and **Error Diffusion** methods achieve the "best" *observed visual quality* from my personal view, while they have the "smallest" two *fidelities*.

Even though the **Thresholding** method obtains the "smallest" *RMSE*, we cannot perceive the corresponding quantized result well.

Thus, we can draw such a conclusion that *fidelity* is a "better" metric than *RMSE* to measure the observed visual quality of the quantized images.

METHOD	RMSE	FIDELITY
Thresholding	87.39	77.46
Ordered Dithering for 2×2	97.67	50.19
Ordered Dithering for 4×4	101.01	16.83
Ordered Dithering for 8×8	100.91	15.00
Error Diffusion	98.85	13.70

Appendix

Python code for functions: `utils.py`

```
from numpy import ndarray, square, sqrt, cbrt, vectorize, \
    zeros, exp, array, ones, kron, ones_like, zeros_like
from scipy.ndimage import convolve
import numpy as np
MAX = 255

def fidelity(f: ndarray, b: ndarray,
            gamma: float=2.2,
            size_kernel: int=7, square_sigma: float=2) \
            -> float:
    if f.dtype == np.uint8 or b.dtype == np.uint8:
        f, b = f.clip(0, MAX).astype(np.double), \
        b.clip(0, MAX).astype(np.double)
    f, b = undo_gamma(f, gamma), undo_gamma(b, gamma)
    kernel = kernel_Gauss(size_kernel, square_sigma)
    # f, b = filter_FIR(f, kernel), filter_FIR(b, kernel)
    f, b = convolve(f, kernel, mode='nearest'), \
        convolve(b, kernel, mode='nearest')
    f, b = cuberoot(f), cuberoot(b)
    return RMSE(f, b)

def RMSE(f: ndarray, b: ndarray) -> float:
    H, W = f.shape
    if f.dtype == np.uint8 or b.dtype == np.uint8:
        f, b = f.clip(0, MAX).astype(np.double), \
        b.clip(0, MAX).astype(np.double)
    return sqrt( square(f - b).sum() / (H*W) )

def correct_gamma(x: ndarray, gamma: float) -> ndarray:
    func = lambda t: round(MAX*pow(t/MAX, 1./gamma))
    f = vectorize(func)
    return f(x).astype(np.uint8)

def undo_gamma(x: ndarray, gamma: float) -> ndarray:
    func = lambda t: MAX*pow(t/MAX, gamma)
    f = vectorize(func)
```

```python
    return f(x)

def cuberoot(x: ndarray) -> ndarray:
    return MAX*cbrt(x/MAX)

def kernel_Gauss(size_kernel: int, square_sigma: float) \
                -> ndarray:
    assert size_kernel%2 == 1 and square_sigma > 0
    dx = size_kernel//2
    sq = square(range(-dx, dx+1)).reshape([size_kernel, 1])
    sq = sq + sq.T
    kernel = exp( -sq/(2*square_sigma) )
    kernel /= kernel.sum()
    return kernel

def filter_FIR(x: ndarray, kernel: ndarray) -> ndarray:
    height, width = x.shape
    ky, kx = kernel.shape[0], kernel.shape[-1]
    assert kx%2 == 1 and ky%2 == 1
    dy, dx = ky//2, kx//2
    x_out = zeros((height, width))
    x_out[:dy, :]   = x[:dx, :]
    x_out[-dy:, :]  = x[-dy:, :]
    x_out[:, :dx]   = x[:, :dx]
    x_out[:, :-dx]  = x[:, :-dx]
    for i in range(dy, height-dy):
        for j in range(dx, width-dx):
            x_out[i][j] = (x[i-dy:i-dy+ky, j-dx:j-dx+kx] \
                               * kernel).sum()
    return x_out

def dither_ordered(x: ndarray, size: int) -> ndarray:
    H, W = x.shape
    b = zeros_like(x, dtype=np.uint8)
    T = index_matrix(size) + 0.5
    T *= MAX / (size*size)
    for i in range(0, H, size):
        di = min(H-i, size)
        for j in range(0, W, size):
            dj = min(W-j, size)
            b[i:i+di, j:j+dj] \
                = MAX*(x[i:i+di, j:j+dj] > T[:di, :dj])
    return b
```

```python
def index_matrix(size: int) -> ndarray:
    # check `size` is a power of 2
    assert ( size != 0 and (size & (size - 1)) == 0 )
    I2 = array([[1, 2],
                [3, 0]])
    k = 4 * ones((2, 2), dtype=np.int)
    I = 0
    for _ in range(size.bit_length()-1):
        I = kron(k, I) + kron(I2, ones_like(I, dtype=np.int))
    return I

# see paper `Optimized Error Diffusion for Image Display`
# https://engineering.purdue.edu/~bouman/publications/orig-
pdf/jei1.pdf
# or https://engineering.purdue.edu/~bouman/publications/pdf/jei1.pdf
def diffuse_error(x: ndarray, T: int) -> ndarray:
    g1_1, g10, g11 = 3/16, 5/16, 1/16
    g01 = 7/16
    H, W = x.shape
    reg, e, b = zeros(W), zeros(x.shape), zeros(x.shape)
    b[0][0] = MAX*(x[0][0] > T)
    e[0][0] = x[0][0] - b[0][0]
    for col in range(1, W):
        e[0, col] = x[0, col] + g01 * e[0, col-1]
        b[0, col] = MAX*(e[0, col] > T)
        e[0, col]-= b[0, col]
    for row in range(1, H):
        reg[1:-1] = g1_1 * e[row-1, 2:] \
            + g10 * e[row-1, 1:-1] + g11 * e[row-1, :-2]
        reg[-1]   = g10  * e[row-1, -1] + g11 * e[row-1, -2]
        e[row, 0] = x[row, 0] \
            + g10 * e[row-1, 0] + g1_1 * e[row-1, 1]
        b[row, 0] = MAX*(e[row, 0] > T)
        e[row, 0]-= b[row, 0]
        for col in range(1, W):
            e[row, col] = x[row, col] \
                + g01 * e[row, col-1] + reg[col]
            b[row, col] = MAX*(e[row, col] > T)
            e[row, col]-= b[row, col]
    return b
```

Python codes for solutions

solution to section 3: `soln_3.py`

```python
import sys
from os.path import dirname
sys.path.insert(0, dirname(dirname(__file__)))
from PIL import Image
from numpy import array
import numpy as np
from src.utils import RMSE, fidelity
from os.path import join

if __name__ == "__main__":
    T = 127 # threshold to produce binary image
    f = array(Image.open(join('resource', 'house.tif')))
    b = (255*(f>T)).astype(np.uint8)
    rmse, fid = RMSE(f, b), fidelity(f, b)
    print("RMSE: ", rmse.round(2))
    print("fidelity: ", fid.round(2))
    img_binary = Image.fromarray(b.clip(0, 255).astype(np.uint8))
    img_binary.save(join('result', 'fig_3.tif'))
```

solution to section 4: `soln_4.py`

```python
import sys
from os.path import dirname
sys.path.insert(0, dirname(dirname(__file__)))
from PIL import Image
from numpy import array
import numpy as np
from src.utils import RMSE, fidelity, undo_gamma, dither_ordered,
index_matrix
from os.path import join
from sympy import Matrix, latex

if __name__ == "__main__":
    f = array(Image.open(join('resource', 'house.tif')))
    x = undo_gamma(f, gamma=2.2)
    for char, size in list(zip(['a', 'b', 'c'], [2, 4, 8])):
        b = dither_ordered(x, size)
        rmse, fid = RMSE(f, b), fidelity(f, b)
        print("RMSE: ", rmse.round(2))
        print("fidelity: ", fid.round(2))
        # print(index_matrix(size))
        print(latex((Matrix(index_matrix(size)))), '\n')
        img_binary = Image.fromarray(b.clip(0, 255).astype(np.uint8))
        img_binary.save(join('result', 'fig_4' + char + '.tif'))
```

solution to section 5: `soln_5.py`

```
import sys
from os.path import dirname
sys.path.insert(0, dirname(dirname(__file__)))
from PIL import Image
from numpy import array
import numpy as np
from src.utils import RMSE, fidelity, undo_gamma, diffuse_error
from os.path import join

if __name__ == "__main__":
    f = array(Image.open(join('resource', 'house.tif')))
    x = undo_gamma(f, gamma=2.2)
    T = 127
    b = diffuse_error(x, T)
    rmse, fid = RMSE(f, b), fidelity(f, b)
    print("RMSE: ", rmse.round(2))
    print("fidelity: ", fid.round(2))
    img_binary = Image.fromarray(b.clip(0, 255).astype(np.uint8))
    img_binary.save(join('result', 'fig_5.tif'))
```