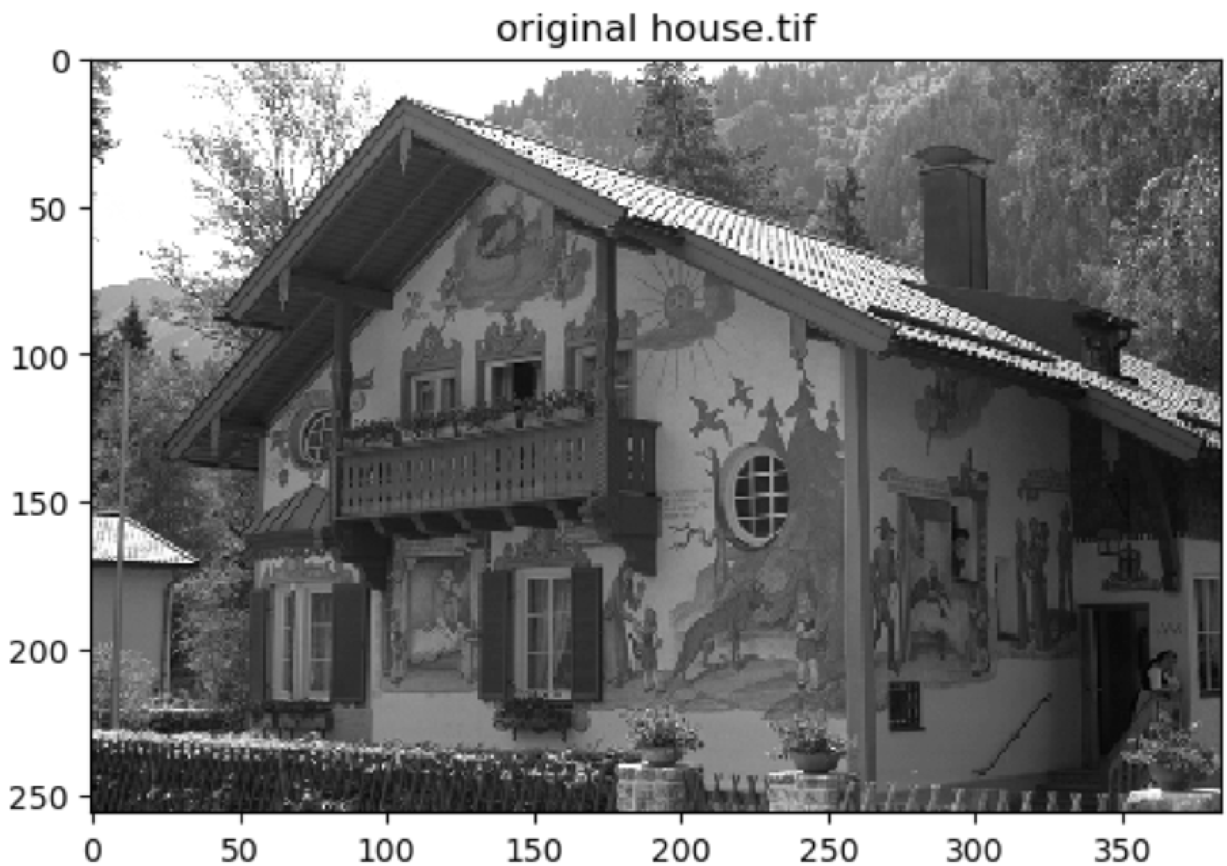
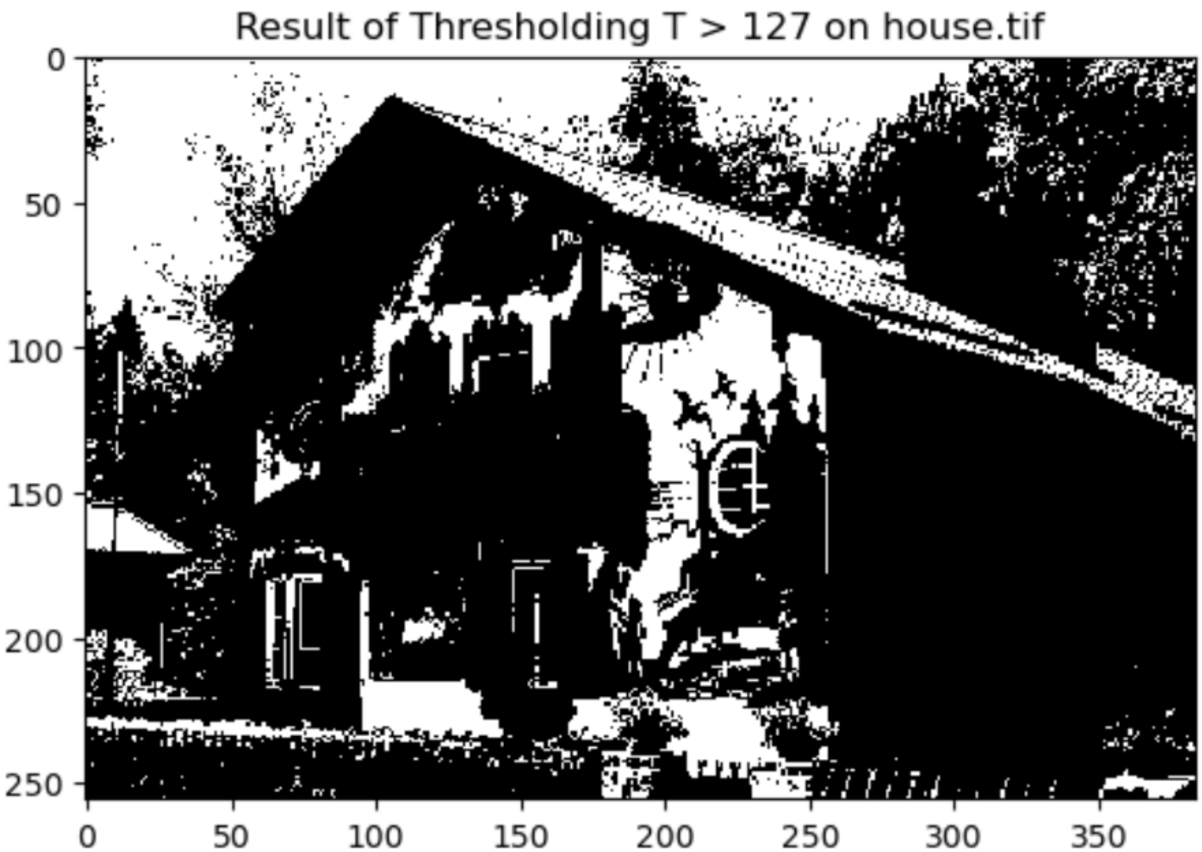
Section 1: Thresholding and Random Noise Binarization

Deliverable 1: Original Image and the result of thresholding:

Original Image:



Result of Thresholding:



__Deliverable 2: Computed RMSE and Fidelity Values:__

```
RMSE is 87.3933165438293

0.00090471]]
Fidelity is 69.96427824449194
```

__Deliverable 3: code for fidelity function:__

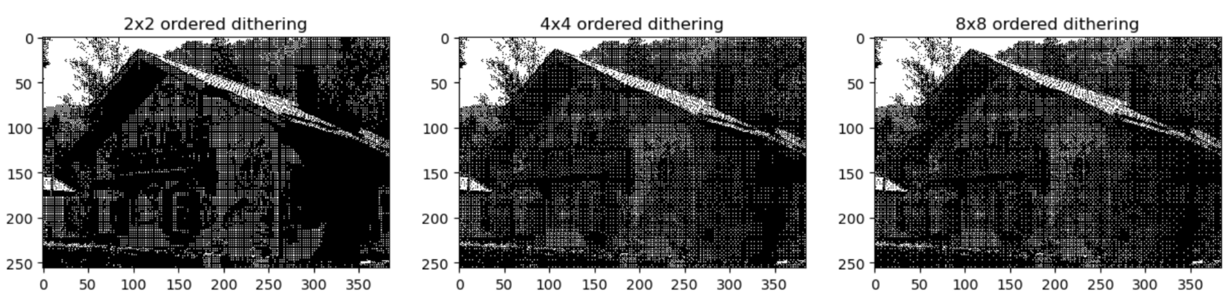
```python
def computeFidelity(img_np, img_binary, plot, gamma):
    #For original (img_np)
    un_gamma_orig = 255 * (img_np / 255)** gamma

    #For binary image (img_binary)

    un_gamma_binary = 255 * (img_binary / 255)** gamma

    #Now, apply LPF to un_gamma_orig and un_gamma_binary using 7 x 7 Gaussian filter

    filt = np.zeros((7,7))
    sigma = np.sqrt(2)
    for m in range(7):
        for n in range(7):
            filt[m,n] = np.exp(-((m-3)**2 + (n-3)**2) / (2 * sigma**2) )
    filt = filt / np.sum(filt)
    print(filt)

    for i in range(img_np.shape[0]):
        for j in range(img_np.shape[1]):
            if (i > 3) and (j > 3) and (i < img_np.shape[0]-3) and (j < img_np.shape[1] - 3):
                #apply filter:
                input_signal_un_gamma_orig = un_gamma_orig[i-3 :i + 4, j - 3: j+ 4] #7 x 7 window:
                #print(input_signal_un_gamma_orig.shape)
                #print('grabbing px centered at ', i, j)
                un_gamma_orig[i,j] = np.sum(input_signal_un_gamma_orig * filt) #filter the ungamma orig

                input_signal_un_gamma_binary = un_gamma_binary[i-3 :i + 4, j - 3: j+ 4] #7x 7 window of binary img
                un_gamma_binary[i,j] = np.sum(input_signal_un_gamma_binary * filt) #filter the ungamma binary



    #Apply transformation y = 255 * (x/255) ^ (1/3) for each pixel of the filtered images:
    un_gamma_orig_transformed = 255 * (un_gamma_orig / 255) ** (1/3)
    un_gamma_binary_transformed = 255 * (un_gamma_binary / 255) ** (1/3)

    # Plot images:
    if (plot == True):
        fig, ax = plt.subplots(2,2, figsize=(15,15))
        ax[0,0].imshow(un_gamma_orig, cmap='gray', interpolation = 'none')
        ax[0,0].set_title('orig filtered')
        ax[1,0].imshow(un_gamma_binary, cmap='gray', interpolation = 'none')
        ax[1,0].set_title('binary filtered')
        ax[0,1].imshow(un_gamma_orig_transformed, cmap='gray', interpolation = 'none')
        ax[0,1].set_title('orig filtered and transformed')
        ax[1,1].imshow(un_gamma_binary_transformed, cmap='gray', interpolation = 'none')
        ax[1,1].set_title('binary filtered and transformed')

    #compute fidelity:
    error_sum = 0
    for i in range(img_np.shape[0]):
        for j in range(img_np.shape[1]):
            error_squared = np.square(un_gamma_orig_transformed[i,j] - un_gamma_binary_transformed[i,j])
            error_sum = error_sum + error_squared

    total_px = img_np.shape[0] * img_np.shape[1]
    fidelity = np.sqrt(error_sum / total_px)
    print("Fidelity is", fidelity)

computeFidelity(img_np, img_binary, plot=0, gamma=2.2)

```

Section 2: Ordered Dithering

Deliverable 1: The 3 Bayer index matrices of size 2x2, 4x4, and 8x8:

```
Bayer 2x2 index matrix:
[[ 95.625 159.375]
 [223.125  31.875]]
Bayer 4x4 index matrix:
[[ 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]]
Bayer 8x8 index matrix:
[[ 85.6640625 149.4140625 101.6015625 165.3515625  89.6484375 153.3984375
   105.5859375 169.3359375]
 [213.1640625  21.9140625 229.1015625  37.8515625 217.1484375  25.8984375
  233.0859375  41.8359375]
 [117.5390625 181.2890625  69.7265625 133.4765625 121.5234375 185.2734375
   73.7109375 137.4609375]
 [245.0390625  53.7890625 197.2265625   5.9765625 249.0234375  57.7734375
  201.2109375   9.9609375]
 [ 93.6328125 157.3828125 109.5703125 173.3203125  81.6796875 145.4296875
   97.6171875 161.3671875]
 [221.1328125  29.8828125 237.0703125  45.8203125 209.1796875  17.9296875
  225.1171875  33.8671875]
 [125.5078125 189.2578125  77.6953125 141.4453125 113.5546875 177.3046875
   65.7421875 129.4921875]
 [253.0078125  61.7578125 205.1953125  13.9453125 241.0546875  49.8046875
  193.2421875   1.9921875]]
```

Deliverable 2: The 3 halftoned images produced by the 3 dither patterns:



Deliverable 3: The RMSE and fidelity for each of the 3 halftoned images:

```
stats for 2x2 ordered dithering halftoned image:
RMSE is 97.66897219213996
Fidelity is 47.06039336965755
```

```
stats for 4x4 ordered dithering halftoned image:
RMSE is 101.00692201569473
Fidelity is 26.616147703975123
```

```
stats for 8x8 ordered dithering halftoned image:
RMSE is 100.91452962396079
Fidelity is 25.88165457695 94
```

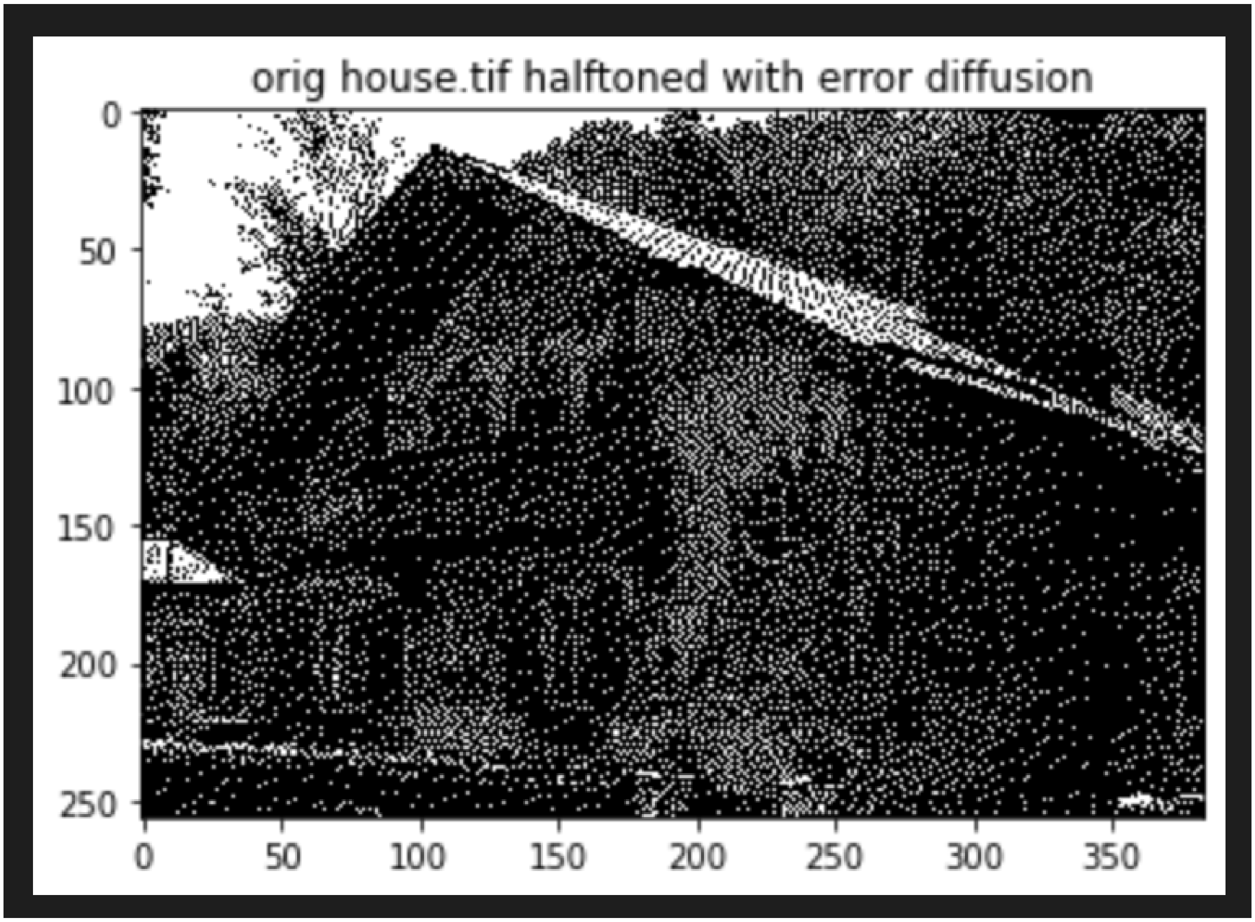
Section 5: Error Diffusion

Deliverable 1: Error Diffusion Python Code

```
### Part 5: Error Diffusion

#Note, this algorithm needs to be applied to a linear scale version of house.tif.
#Therefore, need work with the img_ungamma, which was already ungamma corrected using:
#255 * (gamma_corrected_px / 255) ** gamma.

#We also need to work in raser order, which is pixels left to right, then, top to
bottom.

#Part 1: Init an output matrix image with 0s
PIL_img = Image.open('house.tif')
np_img = np.uint8(PIL_img)
lin_img = 255 * ((np_img / 255.0) ** 2.2)
rows = np_img.shape[0]
cols = np_img.shape[1]
output_matrix = np.zeros((rows,cols))

diffusion_filter = np.array([[0, 0, 0],
                             [0, 0, 7/16],
```

```
                              [3/16, 5/16, 1/16]])

print(diffusion_filter)
quantization_error_matrix = np.zeros((rows,cols))

#Part 2, 3, 4, 5: Quantize the current pixel to 0 or 255 using the threshold T = 127
(need to operate on img_ungamma!), compute the error
# between lin img and binary img, then, propogate that error using coefficients in
Floyd/Steinberg filter to the next pixels in the linear img.
# then, repeat quantizing for next pixels in raster order based on the updated lineaer
pixels!
T = 127
for i in range(rows):
   for j in range(cols):
       if (lin_img[i,j] > T): #quantize based on threshold, placing result in output
matrix
           output_matrix[i,j] = 255
       #compute the quantization error by subtracting the binary pixel from grayscale:
       quantization_error_matrix[i,j] = lin_img[i,j] - output_matrix[i,j] #linear
grayscale - binary

       #add scaled versions of this error using filter (only filtering if we are
within an adequate boundary)
       #pass quantization error onto following pixels: [i, j+ 1] gets 7/16 * error
added, [i+1, j-1] gets 3/16 of the error added, [i+1, j] gets 5/16 error added
       # and [i+1, j+1] gets 1/16 of the error added.
       if (i > 0) and (j > 0) and (i < rows-1) and (j < cols-1):
           lin_img[i, j+1] = lin_img[i,j+1] + (diffusion_filter[1,2] *
quantization_error_matrix[i,j])
           lin_img[i+1, j-1] = lin_img[i + 1, j - 1] + (diffusion_filter[2,0] *
quantization_error_matrix[i,j])
           lin_img[i+1, j] = lin_img[i+1, j] + (diffusion_filter[2,1] *
quantization_error_matrix[i,j])
           lin_img[i+1, j+1] = lin_img[i+1, j+1] + (diffusion_filter[2,2] *
quantization_error_matrix[i,j])

plt.imshow(output_matrix, cmap='gray', interpolation='none')
plt.title('orig house.tif halftoned with error diffusion')

# Compute RMSE
computeRMSE(np.uint8(np_img), output_matrix)
computeFidelity(np_img, output_matrix, plot=0, gamma=2.2)
```

Deliverable 2: Error Diffusion Result:



Deliverable 3: RMSE and Fidelity of the error diffusion result

```
RMSE is 98.84985901290777
Fidelity is 25.270244737487417
```

Deliverable 4: Tabulate the RMSE and Fidelity results for all the experiments in the lab, comment on observations and relate the metrics to the observed visual quality

	RMSE	Fidelity
Simple Thresholding	87.39	69.96

Ordered Dithering 2x2	97.66	47.96
Ordered Dithering 4x4	101.00	26.61
Ordered Dithering 8x8	100.91	25.88
Error Diffusion	98.84	25.27

Overall, the Error Diffusion method creates the best fidelity metric even though the RMSE is higher than simple thresholding. Ordered Dithering is definitely better in terms of fidelity compared to simple thresholding as well. Simple thresholding has the worst fidelity. That poor fidelity seems quite apparent when looking at the output image.