ROBT 310: Homework Project 2

Date: 25^{th} February 2021

Danissa Sandykbayeva

****All of the function tests are included in main_testbench.m file**

Part I - Warm up tasks

Task 1.1 - Nearest-Neighbor Interpolation

The first task was to implement Nearest-Neighbor Interpolation algorithm for a gray-scale image as a scaling function. The code for the resulting function with short explanations is below.

1) First of all we read the image and store the information about dimensions of the image as `size_x` ans `size_y` variables:

```
img = imread(input_file_name);

size_x = size(img,1);
size_y = size(img,2);
```

2)After this we simply use inverse mapping in order to assign each output pixel and it's corresponding neighbors an intensity of the input pixel:

```
new_c = ceil([1:size_x*scale_factor]./scale_factor);
new_r = ceil([1:size_y*scale_factor]./scale_factor);

intr_img = img(new_c,new_r);
```

3) Lastly we write the resulting image into the file with a specified name:

```
imwrite(intr_img, output_file_name);
```

The results of our interpolation function executed with a scale factor of 2 are present on Fig. 1 (a) and (b)

Task 1.2 - Local and Global histogram equalization

For this task we had to consider two approaches to histogram enhancement techniques used to reveal the hidden image. Global histogram equalization enhances an image based on the intensity distribution data from the entire image. While this approach is good for overall enhancement it fails to enhance small details as well as work effectively with images which have most of the intensity

(a) Original image

(b) Output image

Figure 1: Nearest-Neighbor Interpolation

level distributions equal to 0.

On the other hand the local histogram equalization maps the intensity of pixels in a precised neighborhood of size [x,y]. For the purpose of time-effectiveness this task uses local enhancement with non-overlapping neighborhood. While this approach creates visible square footprints of the regions it is significantly faster than considering a neighborhood for each pixel of an image. The procedure function is discussed below:

1) First of all we import the image and store its dimensions. The `dimension` variable is used to check that the image is gray-scale (since some images from the internet while can appear to be gray-scale actually still contain 3 channels) and in case it is more than 1 we need to convert an image to the gray-scale format:

```
img = imread(input_file_name);
[size_x, size_y, dimension] = size(img);
if (dimension > 1)
    img = rgb2gray(img);
end
```

2) After this we implement two different approaches to histogram enhancement using built-in `histeq()` function and `blockproc()` function in order to implement neighborhood regions:

```
auto_hist = histeq(img);
lc_hist = @(block_struct) histeq(block_struct.data);
local_hist = blockproc(img,[40 40], lc_hist);
```

3) Lastly we output both results in one Figure (Fig. 2) for comparison. As it can be seen from the Fig. 2 Global histogram equalization barely helped to see the disguised image since it uses the mean and variance and intensity distribution levels for the whole image, and since seemingly most

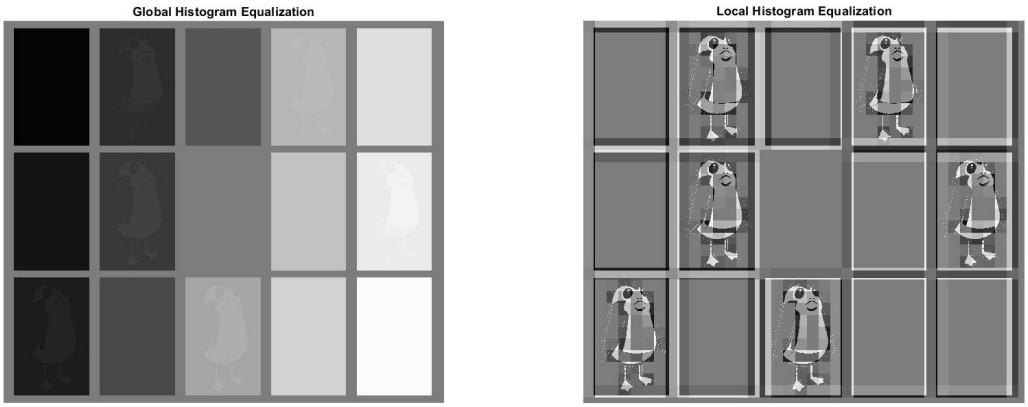

Figure 2: Local and Global Histogram Equalization

of the non-zero tones are distributed evenly, it doesn't really make much difference.

On the other hand local histogram equalization takes into account local mean and variance which represent local average intensity and image contrast level respectively. Therefore, it takes measures to enhance each region separately revealing greater details.

Part II - Visual Illusion (dithering)

Task 2.1 - Floyd–Steinberg dithering

Simply speaking, Floyd-Steinberg (FS) dithering algorithm applies error created by the difference in input and output pixel's intensity onto neighbouring pixels. Here are the insides of the function for FS dithering:

1) Firstly as used previously we make sure that an image is gray-scale by checking it's dimension for one pixel, and if it's not convert it to gray-scale one.
2) We define a copy of the image that would store the resulting image (we create a copy so that at the end we could show both original and output image side-by-side). And also we begin our loop for going through each pixel:

```
img_c = img; % future resulting image

    for i = 1:size_x
        for j = 1:size_y
        ...
        end
    end
    ...
```

3) Inside the loop we first of all save a value of the given input (`old_p`) pixel and assign a new

value to it. Since we only have two levels (black and white) we use simple threshold to determine new value for the pixel: if the original intensity is higher that $256/2 = 128$ we assign **white - 255**, else, we assign **black - 0**:

```
old_p = img_c(i,j);
if old_p > 128
    new_p = 255;
else
    new_p = 0;
end
```

4) Next we begin the adjustment of the neighbouring pixel's current intensity levels according to the quantization error defined as $e = I_{old\_p} - I_{new\_p}$. Needless to say, that the if statements are provided in order to prevent current pixel indexing from falling out of the image constrains.

```
if i < size_x-1
    img_c(i+1,j) = img_c(i+1,j) + err*7/16;
end
if (i > 1) && (j < size_y - 1)
    img_c(i-1,j+1) = img_c(i-1,j+1) + err*3/16;
end
if j < size_y - 1
    img_c(i,j+1) = img_c(i,j+1) + err*5/16;
end
if (i < size_x - 1) && (j < size_y - 1)
    img_c(i+1,j+1) = img_c(i+1,j+1) + err*1/16;
end
```

5) Final step is just to output both original and resulting image and save result a a file with a specified `output_file_name`. The results of constructed FS dithering are shown on Fig. 3 (b) as well as the original image (a) used.

Task 2.1 - Bayer (Ordered) dithering

Bayer (Ordered) dithering approach uses dither matrices. Dither matrix is a square matrix with dimensions equal to some power-of-2. Each of the element in this matrix represents a threshold. These values are distributed so that consecutive prioritizing of certain pixels' intensity creates an illusion of gradual variation of color, in our case - gray-scale variation. The general formula for Bayer matrices is described as the following:

$$D_n = \begin{bmatrix} 4D_{n/2} + 0 & 4D_{n/2} + 2 \\ 4D_{n/2} + 3 & 4D_{n/2} + 1 \end{bmatrix} \tag{1}$$

For the purpose of this task we are going to take 4 by 4 dither matrix since it provides an image of a good enough quality with stable dithering quality. The Bayer matrix we need according to (1) is:

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

(a) Original image

(b) FS dithering resulting image

Figure 3: Floyd-Steinberg Dithering

Below is the function procedure for the Bayer (Ordering) dithering algorithm applied on a gray-scale image.

1) Firstly as used previously we make sure that an image is gray-scale by checking it's dimension for one pixel, and if it's not convert it to gray-scale one.
2) Next we define our dither (Bayer) matrix we are going to use for the task and variable for it's dimension:

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

3) We again create a copy of an image to work with to preserve the original for the final comparison.
4) First we define the loops that would go through the regions of pixels 4 by 4 on the original image:

```
for i = 1:M_dith_d:size_x-M_dith_d
    for j = 1:M_dith_d:size_y-M_dith_d
    ...
    end
end
```

Next inside of this nested loop we define another nested loop that would go through each element of our dither matrix for pixel intensity comparison. The final loop structure would look like the following:

```
for i = 1:M_dith_d:size_x-M_dith_d
        for j = 1:M_dith_d:size_y-M_dith_d
            for mt_x = 1:M_dith_d
                for mt_y = 1:M_dith_d
                    ...
```

```
                end
            end
        end
    end
```

Finally inside the loop we performs a simple comparison of given pixel's normalized intensity level (`double(img(i\mt_x,j\mt_y))/256`) and corresponding element of the matrix, also normalized according to the number of dimensions (`double(M_dith(mt_x,mt_y))/16`). The resulting algorithmic part is:

```
for i = 1:M_dith_d:size_x-M_dith_d
        for j = 1:M_dith_d:size_y-M_dith_d
            for mt_x = 1:M_dith_d
                for mt_y = 1:M_dith_d
                    if (double(img(i+mt_x,j+mt_y))/256 >
                    double(M_dith(mt_x,mt_y))/16)
                        new_p = 255; % white
                    else
                        new_p = 0;   % black
                    end
                    img_c(i+mt_x,j+mt_y) = new_p;
                end
            end
        end
    end
```

5)The problem that appeared with this algorithm is that whenever one of the original image dimensions was divisible by 4, the algorithm omitted the last trail of 3 pixels along the corresponding dimension. Therefore, there is an additional part to this algorithm that handles those special cases. We simply process the left-out column or row manually:

```
if rem(size_x,M_dith_d) == 0
        for j = 1:M_dith_d:size_y-M_dith_d
            for mt_x = 1:M_dith_d
                for mt_y = 1: M_dith_d
                    if (double(img(size_x-4+mt_x,j+mt_y))/256 >
                    double(M_dith(mt_x,mt_y))/16)
                        new_p = 255;
                    else
                        new_p = 0;
                    end
                    img_c(size_x-4+mt_x,j+mt_y) = new_p;
                end
            end
        end
    end
```

And the similar algorithm is then applied to the case with left-out rows. **6)** Final step is just to output both original and resulting image and save result a a file with a specified `output_file_name`.

(a) Original image

(b) Bayer(Ordered) dithering resulting image

Figure 4: Floyd-Steinberg Dithering

The results of constructed Bayer(Ordered) dithering are shown on Fig. 3 (b) as well as the original image (a) used.