ECE 438 - Laboratory 10b
Image Processing (Week 2)

Last updated on April 24, 2022

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt
```

In [2]:
```
# make sure the plot is displayed in this notebook
%matplotlib inline
# specify the size of the plot
plt.rcParams['figure.figsize'] = (16, 10)

# for auto-reloading extenrnal modules
%load_ext autoreload
%autoreload 2
```

1. Introduction

This is the second part of a two week experiment in image processing. In the first week , we covered the fundamentals of digital monochrome images, intensity histograms, pointwise transformations, gamma correction, and image enhancement based on filtering.

During this week, we will cover some fundamental concepts of color images. This will include a brief description on how humans perceive color, followed by descriptions of two standard **color spaces**. We will also discuss an application known as **halftoning**, which is the process of converting a gray scale image into a binary image.

2. Color Images

2.1. Background on Color

Color is a perceptual phenomenon related to the human response to different wavelengths of light, mainly in the region of 400 to 700 nanometers (nm). The perception of color arises from the sensitivities of three types of neurochemical sensors in the retina, known as the long (L), medium (M), and short (S) cones. The response of these sensors to photons is shown in Figure 1. Note that each sensor responds to a range of wavelengths.

Due to this property of the human visual system, all colors can be modeled as combinations of the three primary color components: red (R), green (G), and blue (B). For the purpose of standardization, the CIE (Commission International de l'Eclairage — the International Commission on Illumination) designated the following wavelength values for the three primary colors: $\text{blue} = 435.8 \text{ nm}$, $\text{green} = 546.1 \text{ nm}$, and $\text{red} = 700 \text{ nm}$.

The relative amounts of the three primary colors of light required to produce a color of a given wavelength are called *tristimulus values*. Figure 2 shows the plot of tristimulus values using the CIE primary colors. Notice that some of the tristimulus values are negative, which indicates that colors at those wavelengths cannot be reproduced by the CIE primary colors.

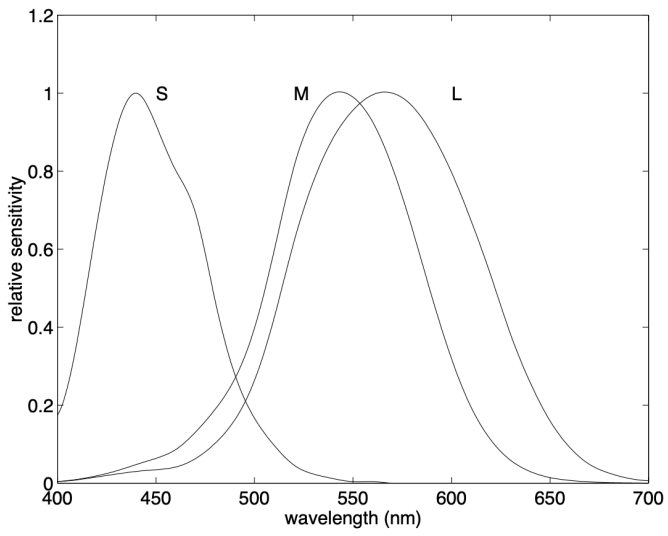

Figure 1: Relative photon sensitivity of long (L), medium (M), and short (S) cones.

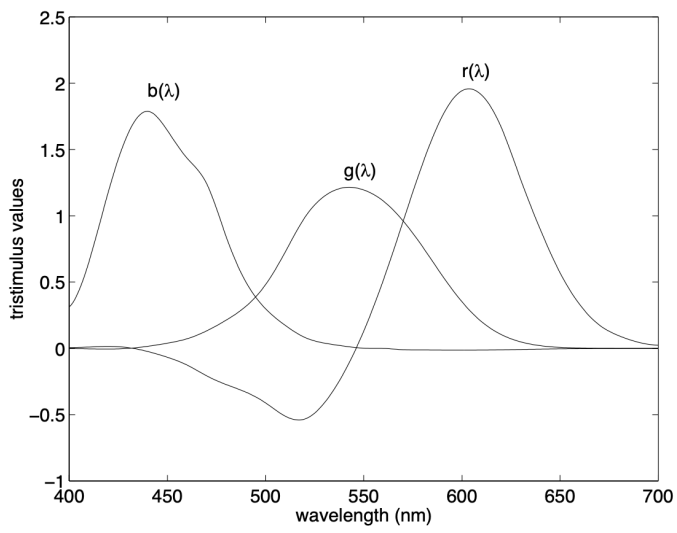

Figure 2: Plot of tristimulus values using CIE primary colors.

2.2. Color Spaces

A color space allows us to represent all the colors perceived by human beings. We previously noted that weighted combinations of stimuli at three

wavelengths are sufficient to describe all the colors we perceive. These wavelengths form a natural basis, or coordinate system, from which the color measurement process can be described. In this lab, we will examine two common color spaces: RGB and YC_bC_r. For more information, refer to [1].

- RGB space is one of the most popular color spaces, and is based on the tristimulus theory of human vision, as described above. The RGB space is a hardware-oriented model, and is thus primarily used in computer monitors and other raster devices. Based upon this color space, each pixel of a digital color image has three components: red, green, and blue.
- YC_bC_r space is another important color space model. This is a gamma corrected space defined by the CCIR (International Radio Consultative Committee), and is mainly used in the digital video paradigm. This space consists of luminance (Y) and chrominance (C_bC_r) components. The importance of the YC_bC_r space comes from the fact that the human visual system perceives a color stimulus in terms of luminance and chrominance attributes, rather than in terms of R, G, and B values. The relation between YC_bC_r space and gamma corrected RGB space is given by the following linear transformation:

$$\begin{aligned} Y &= 0.299R + 0.587G + 0.114B \\ C_b &= 0.564(B - Y) + 128 \\ C_r &= 0.713(R - Y) + 128 \end{aligned} \tag{1}$$

In YC_bC_r, the luminance parameter is related to an overall intensity of the image. The chrominance components are a measure of the relative intensities of the blue and red components. The inverse of the transformation in equation (1) can easily be shown to be the following:

$$\begin{aligned} R &= Y + 1.4025(C_r - 128) \\ G &= Y - 0.3443(C_b - 128) - 0.7144(C_r - 128) \\ B &= Y + 1.7730(C_b - 128) \end{aligned} \tag{2}$$

Exercise 2.3: Color

1. Load the image file `girl.tif`. Check the size of array for this image by using the command `print(image.shape)`, where `image` is the image matrix. Also, print the data type of this matrix.

Notice that this is a three dimensional array of type `uint8`. It contains three gray scale image planes corresponding to the red, green, and blue components for each pixel. Since each color pixel is represented by three bytes, this is commonly known as a 24-bit image.

In [3]:
```
# insert your code here
```

2. Display the image. Note that `cmap`, `vmin`, `vmax` arguments are not needed.

In [4]:
```
# insert your code here
```

3. Extract each of the color components, then plot each color component.

Note that while the original is a color image, each color component separately is a monochrome image, so plotting each color component requires `cmap`, `vmin`, `vmax` arguments.

In [5]:
```
# insert your code here
```

4. Load the files `ycbcr.npy` using `np.load()` (https://numpy.org/doc/stable/reference/generated/numpy.load.html), and print its type and data shape `dtype`.

This file contains a NumPy array for a color image in YC_bC_r format. The array contains three gray scale image planes that correspond to the luminance (Y) and two chrominance (C_bC_r) components.

In [6]:
```
# insert your code here
```

5. Plot each of the components.

In [7]:
```
# insert your code here
```

In order to properly display this color image, we need to convert it to RGB format.

6. Complete the function below that will perform the transformation of equation (2). It should accept a 3-D YC_bC_r image array as input, and return a 3-D RGB image array.

- Make sure `ycbcr` is in `double` or `float` before any processing.
- After conversion, to make sure the values of `rgb` are in $[0, 255]$, use `np.clip()` (https://numpy.org/doc/stable/reference/generated/numpy.clip.html).

In [8]:
```
def ycbcr2rgb(ycbcr):
    """
    Parameters
    ---
    ycbcr: image in YCbCr

    Returns
    ---
    rgb: image RGB
    """

    rgb = None
    return rgb
```

7. Now, convert the ycbcr array to an RGB representation and display the color image.

- Before displaying the image, make sure its data type is `np.uint8` .

In [9]:
```
# insert your code here
```

An interesting property of the human visual system, with respect to the YC_bC_r color space, is that we are much more sensitive to distortion in the luminance component than in the chrominance components. To illustrate this, we will smooth each of these components with a Gaussian filter and view the results.

8. Load the file `h.npy` . This is a 5×5 Gaussian filter with $\sigma^2 = 2.0$. (See the first week of the experiment for more details on this type of filter.)

In [10]:
```
# insert your code here
```

9. Alter the `ycbcr` array by filtering only the luminance component, `ycbcr[:,:,0]` , using the Gaussian filter (use `convolve2d()` function from last lab). Convert the result to RGB, and display it.

- Instead of altering the original `ycbcr` , you can create a copy by `ycbcr1 = ycbcr.copy()` .

In [11]:
```
def convolve2d(image, kernel):
    """
    Parameters
    ---
    image: the input image
    kernel: the filter

    Returns
    ---
    filtered: the filtered image
    """
    filtered = None
    return filtered
```

In [12]:
```
# insert your code here
```

10. Now alter `ycbcr` by filtering both chrominance components, `ycbcr[:,:,1]` and `ycbcr[:,:,2]` , using the Gaussian filter. Convert this result to RGB, and display it.

- Again, instead of altering the original `ycbcr` , you can create a copy by `ycbcr2 = ycbcr.copy()` .

In [13]:
```
# insert your code here
```

Do you see a significant difference between the filtered versions and the original image? This is the reason that YC_bC_r is often used for digital video. Since we are not very sensitive to corruption of the chrominance components, we can afford to lose some information in the encoding process.

3. Halftoning

In this section, we will cover a useful image processing technique called halftoning. The process of halftoning is required in many present day electronic applications such as facsimile (FAX), electronic scanning/copying, laser printing, and low bandwidth remote sensing.

3.1. Binary Images

As was discussed in the first week of this lab, an 8-bit monochrome image allows 256 distinct gray levels. Such images can be displayed on a computer monitor if the hardware supports the required number intensity levels. However, some output devices print or display images with much fewer gray levels. In the extreme case, the gray scale images must be converted to binary images, where pixels can only be black or white.

Figure 3: (a) Original gray scale image. (b) Binary image produced by simple fixed thresholding.

The simplest way of converting to a binary image is based on thresholding, i.e. two-level (one-bit) quantization. Let $f[i, j]$ be a gray scale image, and $b[i, j]$ be the corresponding binary image based on thresholding. For a given threshold T, the binary image is computed as the following:

$$b[i,j] = \begin{cases} 255 & \text{if } f[i,j] > T \\ 0 & \text{else} \end{cases} \tag{3}$$

Figure 3 shows an example of conversion to a binary image via thresholding, using $T = 80$.

It can be seen in Figure 3 that the binary image is not “shaded” properly–an artifact known as false contouring. False contouring occurs when quantizing at low bit rates (one bit in this case) because the quantization error is dependent upon the input signal. If one reduces this dependence, the visual quality of the binary image is usually enhanced.

One method of reducing the signal dependence of the quantization error is to add uniformly distributed white noise to the input image prior to quantization. To each pixel of the gray scale image $f[i, j]$, a white random number n in the range $[-A, A]$ is added, and then the resulting image is quantized by a one-bit quantizer, as in equation (3). The result of this method is illustrated in Figure 4, where the additive noise is uniform over $[-40, 40]$. Notice that even though the resulting binary image is somewhat noisy, the false contouring has been significantly reduced.

Figure 4: Random noise binarization.

Exercise 3.2: Halftoning - Simple Thresholding

1. Load the grayscale image file `house.tif` and display it.

In [14]:
```
# insert your code here
```

2. Try the simple thresholding technique based on equation (3), using $T = 108$, and display the result.

- In Python, an easy way to threshold an image X is to use the command `Y = 255 * (X > T)`.

In [15]:
```
# insert your code here
```

3. Create an “absolute error” image by subtracting the binary from the original image, and then taking the absolute value. The degree to which the original image is present in the error image is a measure of signal dependence of the quantization error. Display the error image.

In [16]:
```
# insert your code here
```

4. Compute the mean square error (MSE), which is defined by

$$\text{MSE} = \frac{1}{MN} \sum_{i,j} \{f[i,j] - b[i,j]\}^2 \tag{9}$$

where MN is the total number of pixels in each image, f is the original image and b is the binarized image.

In [17]:
```
# insert your code here
```

3.3. Ordered Dithering

Halftone images are binary images that appear to have a gray scale rendition. Although the random thresholding technique described in Section 3.1 can be used to produce a halftone image, it is not often used in real applications since it yields very noisy results. In this section, we will describe a better halftoning technique known as ordered dithering.

The human visual system tends to average a region around a pixel instead of treating each pixel individually, thus it is possible to create the illusion of many gray levels in a binary image, even though there are actually only two gray levels. With 2×2 binary pixel grids, we can represent 5 different "effective" intensity levels, as shown in Figure 5. Similarly for 3×3 grids, we can represent 10 distinct gray levels. In dithering, we replace blocks of the original image with these types of binary grid patterns.

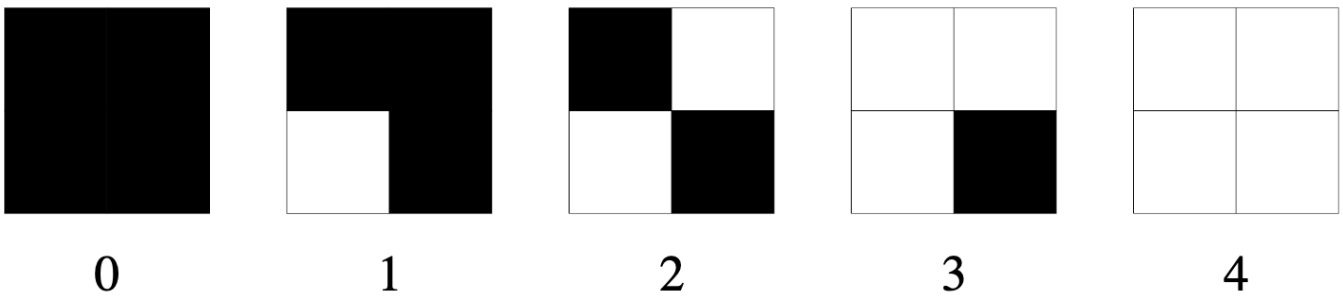

Figure 5: Five different patterns of 2×2 binary pixel grids.

Remember from Section 3.1 that false contouring artifacts can be reduced if we can reduce the signal dependence of the quantization error. We showed that adding uniform noise to the monochrome image can be used to achieve this decorrelation. An alternative method would be to use a variable threshold value for the quantization process.

Ordered dithering consists of comparing blocks of the original image to a 2-D grid, known as a dither pattern. Each element of the block is then quantized using the corresponding value in the dither pattern as a threshold. The values in the dither matrix are fixed, but are typically different from each other. Because the threshold value varies between adjacent pixels, some decorrelation from the quantization error is achieved, which has the effect of reducing false contouring.

The following is an example of a 2×2 dither matrix,

$$T[i, j] = 255 \times \begin{bmatrix} 5/8 & 3/8 \\ 1/8 & 7/8 \end{bmatrix} \tag{4}$$

This is a part of a general class of optimum dither patterns known as Bayer matrices. The values of the threshold matrix $T[i, j]$ are determined by the order that pixels turn "ON". The order can be put in the form of an index matrix. For a Bayer matrix of size 2, the index matrix $I[i, j]$ is given by

$$I[i, j] = \begin{bmatrix} 3 & 2 \\ 1 & 4 \end{bmatrix} \tag{5}$$

and the relation between $T[i, j]$ and $I[i, j]$ is given by

$$T[i, j] = \frac{255 \times (I[i, j] - 0.5)}{n^2} \tag{6}$$

where n^2 is the total number of elements in the matrix.

Figure 6 shows the halftone image produced by Bayer dithering of size 4. It is clear from the figure that the halftone image provides good detail rendition. However the inherent square grid patterns are visible in the halftone image.

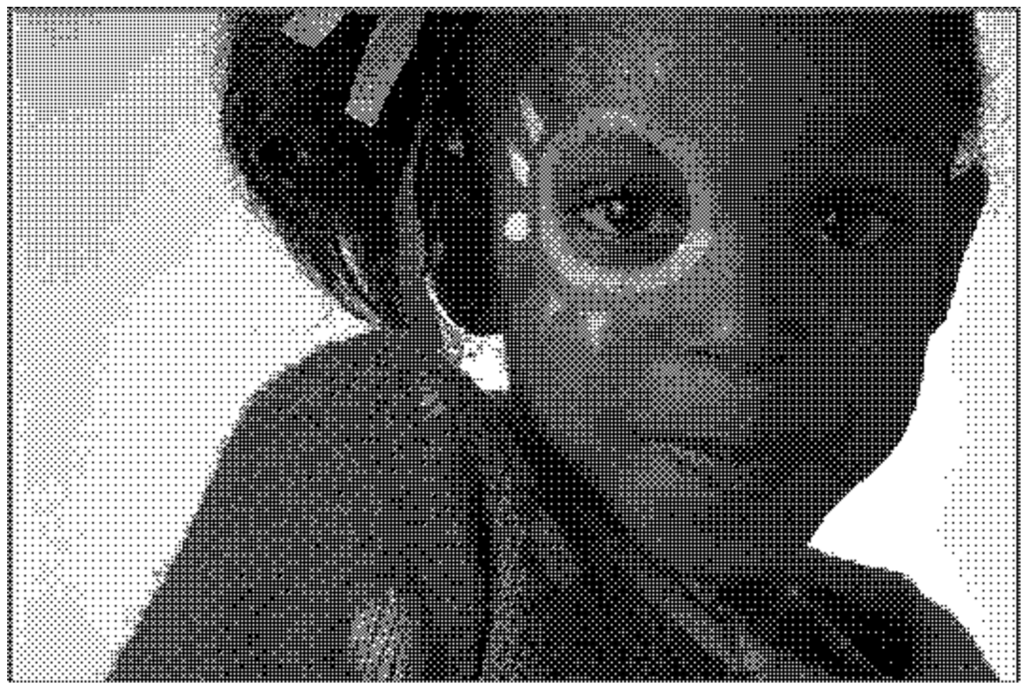

Figure 6: The halftone image produced by Bayer dithering of size 4.

Exercise 3.4: Halftoning - Ordered Dithering

Now try implementing Bayer dithering of size 4. You will first have to compute the dither pattern. The index matrix for a dither pattern of size 4 is given by

$$I[i,j] = \begin{bmatrix} 12 & 8 & 10 & 6 \\ 4 & 16 & 2 & 14 \\ 9 & 5 & 11 & 7 \\ 1 & 13 & 3 & 15 \end{bmatrix} \tag{10}$$

1. Based on this index matrix and equation (6), create the corresponding threshold matrix and print it.

In [18]:
```
# insert your code here
```

For ordered dithering, it is easiest to perform the thresholding of the image all at once. This can be done by creating a large threshold matrix by repeating the 4×4 dither pattern. A useful command here is `np.tile(m, (num_y, num_x))`, where `m` is the matrix to be repeated, `num_y` is the repeated times in the vertical direction, and `num_x` is the repeated times in the horizontal direction.

The thresholding can then be performed using the command `Y = 255 * (X > m)` where `X` and `Y` are the input and the output images.

2. Apply the ordered dithering and display the halftoned image.

In [19]:
```
# insert your code here
```

3. Compute the error image and display it.

In [20]:
```
# insert your code here
```

4. Compute the MSE of the error image.

In [21]:
```
# insert your code here
```

3.5. Error Diffusion

Another method for halftoning is random dithering by error diffusion. In this case, the pixels are quantized in a specific order (raster ordering† is commonly used), and the residual quantization error for the current pixel is propagated (diffused) forward to unquantized pixels. This keeps the overall intensity of the output binary image closer to the input gray scale intensity.

†: Raster ordering of an image orients the pixels from left to right, and then top to bottom. This is similar to the order that a CRT scans the electron beam across the screen.

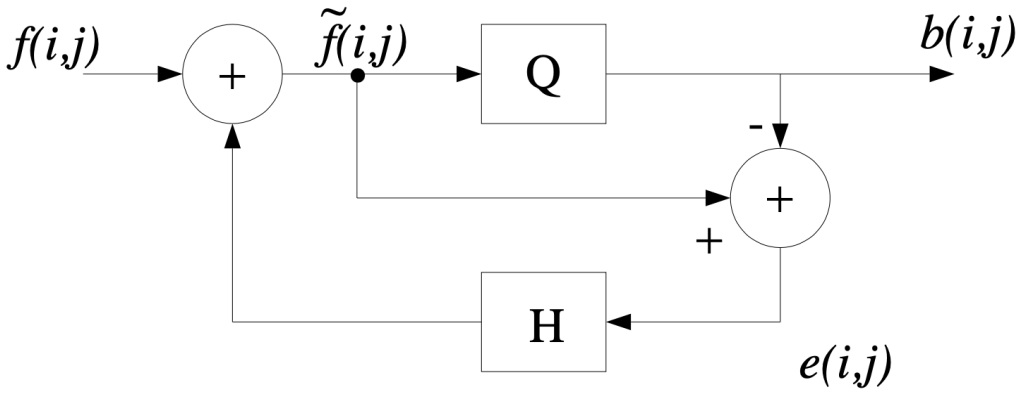

Figure 7: Block diagram of the error diffusion method.

Figure 7 is a block diagram that illustrates the method of error diffusion. The current input pixel $f[i,j]$ is modified by means of past quantization errors to give a modified input $\tilde{f}[i,j]$. This pixel is then quantized to a binary value by Q, using some threshold T. The error $e[i,j]$ is defined as

$$e[i,j] = \tilde{f}[i,j] - b[i,j] \tag{7}$$

where $b[i,j]$ is the quantized binary image.

The error $e[i,j]$ of quantizing the current pixel is is diffused to “future” pixels by means of a two-dimensional weighting filter $h[i,j]$, known as the diffusion filter. The process of modifying an input pixel by past errors can be represented by the following recursive relationship.

$$\tilde{f}[i,j] = f[i,j] + \sum_{k,l \in S} h[k,l] e[i-k, j-l] \tag{8}$$

The most popular error diffusion method, proposed by Floyd and Steinberg, uses the diffusion filter shown in Figure 8. Since the filter coefficients sum to one, the local average value of the quantized image is equal to the local average gray scale value. Figure 9 shows the halftone image produced by Floyd and Steinberg error diffusion. Compared to the ordered dither halftoning, the error diffusion method can be seen to have better contrast performance. However, it can be seen in Figure 9 that error diffusion tends to create “streaking” artifacts, known as worm patterns.

	•	7/16
3/16	5/16	1/16

Figure 8: The error diffusion filter proposed by Floyd and Steinberg.

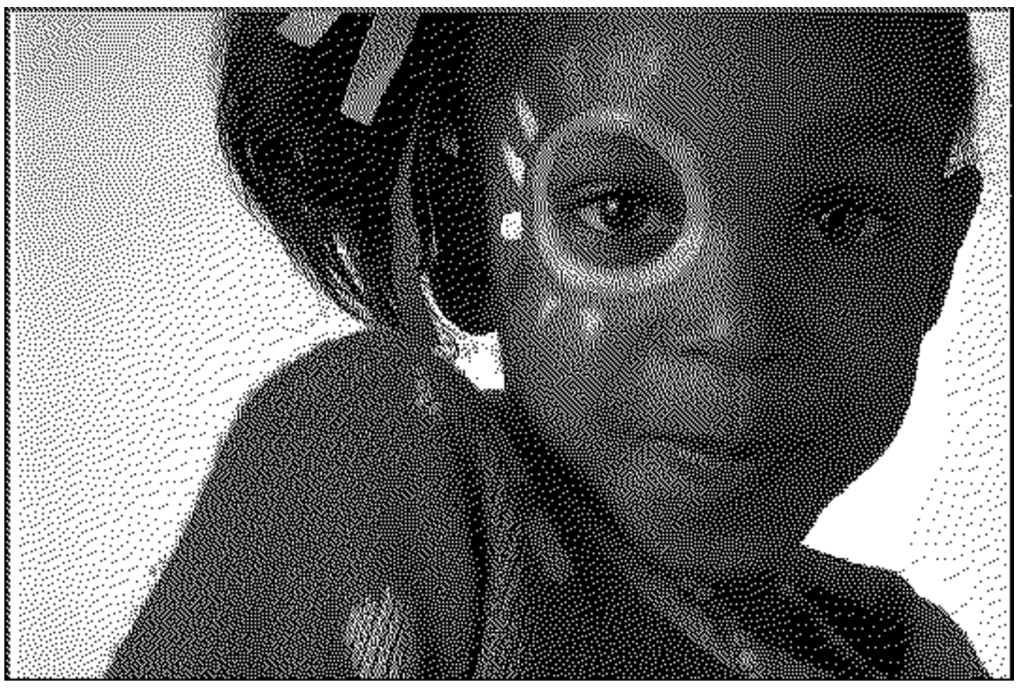

Figure 9: A halftone image produced by the Floyd and Steinberg error diffusion method.

Exercise 3.6: Halftoning - Error Diffusion

Now try halftoning via the error diffusion technique, using a threshold $T = 108$ and the diffusion filter in Figure 8. It is most straightforward to implement this by performing the following steps on each pixel in raster order:

1. Initialize an output image matrix with zeros.
2. Quantize the current pixel using using the threshold T, and place the result in the output matrix.
3. Compute the quantization error by subtracting the binary pixel from the gray scale pixel.
4. Add scaled versions of this error to “future” pixels of the original image, as depicted by the diffusion filter of Figure 8.
5. Move on to the next pixel.

You do not have to quantize the outer border of the image.

1. Use the algorithm to create the halftoned image and display it.

In [22]:
```
# insert your code here
```

2. Compute the error image and display it.

In [23]:
```
# insert your code here
```

3. Compute the MSE of the error image.

In [24]:
```
# insert your code here
```

4. By comparing three MSE values, is the MSE consistent with the visual quality?

insert your answer here

5. By looking at the error images, determine which method appears to be the least signal dependent? Does the signal dependence seem to be correlated with the visual quality?

insert your answer here

Exercise 3.7: Halftoning - Filtered Halftone

1. The human visual system naturally lowpass filters halftone images. To analyze this phenomenon, filter each of the halftone images with the Gaussian lowpass filter `h` that you loaded from `h.npy`, and measure the MSE of the filtered versions.

In [25]:
```
# insert your code here
```

2. Use the following template to make a table.

Halftone Method	Filtered	Not filtered
Simple Thresholding		
Ordered Dithering		
Error Diffusion		

3. Compare the MSE’s of the filtered versions with the nonfiltered versions for each method. What is the implication of these observations with respect to how we perceive halftone images.

insert your answer here

4. References

[1] J.M. Kasson and W. Plouffe, “An analysis of selected computer interchange color spaces,” ACM Trans. Graphics, vol. 11, no. 4, pp. 373–405, 1992.