Assignment I - Error Diffusion Dotting Techniques
Introduction to Digital Image Processing

Renata Falguera Gonçalves - RA: 262889

[1]Computer Institution – Campinas State University (Unicamp)

Abstract. *The purpose of this assigment is to build an algorithm of Error Diffusion Dotting Techniques to reduce the amount of color of an RGB or a monochromatic image.*

1. Assignment Structure

All archives used for this assignment have been saved to a zip file named "***Assignment_I***", which contains:

- Main, Error Diffusion Dotting Techniques for RGB and Monochromatic scripts;
- Images for test and evaluation of the algorithm;
- Output Folder;

2. The Program

Due to the large number of libraries provided, the program was developed with the newest version of Python, the 3.6.8. For better image processing and easy data manipulation, it was decided to use the Numpy (version: 1.17.1), Math, ArgParse and OpenCV (version: 4.1.1) libraries.

2.1. How to Execute

The user must download the ***Assigment_I.zip*** file and extract all the archives in a folder of their choice. As said in Section 1, there are three diffent scripts: ***main.py*** (Main Program), ***monochromatic.py*** (Error Diffusion Dotting Techiniques for Monochromatic Images) and ***RGB.py*** (Error Diffusion Dotting Techiniques for RGB images) respectively.

The program can be run through the ***main.py*** script.

The script takes as its argument the algorithm to be executed and the name of the input image. An example execution is shown below:

python3 main.py -img monalisa.png

2.2. Input and Output

The program inputs are RGB images informed in the statement of the assignmnet (*https://www.ic.unicamp.br/h̃elio/imagens_coloridas/*). The outputs are RGB or monochromatic images, after the quantization process done at the input image, which are saved into the Output folder (as shown in section 1).

3. The Problem Solution

As said in section 2.1, the main script is the base of execution of the program. It will be imported the other two scripts: *mochromatic.py* script, with the nominator "*mono*" and the *RGB.py* script, with the nominator "*colorscript*" (Figure 1).

The *main* program has 3 execution steps:

1. Passing and obtaining the image path argument given at the command as shown in Section 2.1;
2. Opening and reading of the image in the RGB and Monochromatic forms;
3. Accessing the others scripts to apply each technique;
4. Save the output images in folders;

```
1    import monochromatic as mono
2    import RGB as colorscript
```

Figure 1. Importing the *mochromatic.py* and *RGB.py* scripts.

There are four Error Diffusion Dotting Techniques in total (each one with different weights):

- Floyd and Steinberg;
- Burkes;
- Sierra;
- Stuki;
- Jarvis, Judice and Ninke;
- Stivenson and Arce;

3.1. Passing and obtaining the image path argument

In this first part, it was used all the classes and functions contained at the ArgParse library. As seen in figure 2, the first step when using *argparse* is to create a parser object and tell it what arguments to expect (Figure 3). The parser can then be used to process the command line arguments when your program runs.

```
1    parser = argparse.ArgumentParser()
```

Figure 2. Setting up a Parser.

Once all of the arguments are defined, you can parse the command line by passing a sequence of argument strings to *parse_args()* (Figure 3). By default, the arguments are taken from *sys.argv[1:]*, but you can also pass your own list. The options are processed using the GNU/POSIX syntax, so option and argument values can be mixed in the sequence.

The return value from *parse_args()* is a namespace containing the arguments to the command. The object holds the argument values as attributes, so if your argument destination is "myoption", you access the value as *args.myoption*.

In this specific problem, the argument is given by "img_adress", so we acess it using *args.img_adress* and save at the *filename_input* variable (Figure 3).

```
#Input
parser.add_argument('-img',
                    '--img_adress',
                    required = True,
                    help = 'Selected Image')
args = parser.parse_args()
filename_input = args.img_adress
```

Figure 3. Setting up a Parser, defining the arguments and saving the passing argument.

3.2. Opening and Reading

In this second part, it was used the function *cv2.imread()* from the OpenCv library. This function has two parameters: the first one is the image path; the second one is a flag that can assume these 3 values:

- -1 : For loading the image as such including alpha channel;
- 0 : For loading the image in gray scale mode;
- 1 : For loading a color image. Any transparency of image will be neglected. It is the default flag.

Since we want to access the image in its original (RGB) and in monochromatic forms, it is necessary to open the image in different ways, and saving them in different variables.

Thus, to obtain the monochromatic form of the image, it was applied the 0 flag, and saved the image file in the variable called *imageMono*. For the RGB form, by default, the flag 1 was applied, and the image was saved at the *imageRGB*, as seen in the Figure 4.

```
imageMono = cv2.imread(filename_input,0)
imageRGB = cv2.imread(filename_input)
```

Figure 4. Opening and reading of the image in RGB and monochromatic form.

3.3. Script Accessing

Since the images will be saved in a folder, we use the *cv2.imwrite()* function. It has two parameters: the first is the path where the image will be saved; the second is the image after the quantization process (Figure 5 and 6).

```
cv2.imwrite("output/Burkes/Monochromatic/
    Normal_Order-" + filename_input, mono.Burkes(
    imageMono,imageMono.shape[0],imageMono.shape
    [1],0))
```

Figure 5. A example of the main script accessing the *monochromatic.py* script and saving the output image.

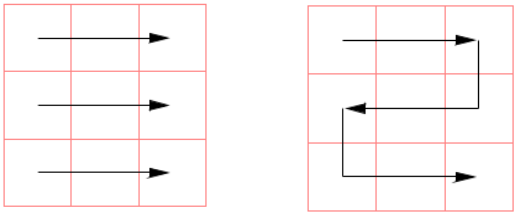
In this third step, it was applied each Error Diffusion Dotting Technique to the images (RGB and Monochromatic), by accessing each function from the *monochromatic.py* and *RGB.py* scripts. For each technique it was created the following functions:

- Floyd_Steinberg (*image, m, n, activator*);
- Burkes (*image, m, n, activator*);
- Sierra (*image, m, n, activator*);
- Stuki (*image, m, n, activator*);
- Jarvis_Judice_Ninke (*image, m, n, activator*);
- Stivenson_Arce (*image, m, n, activator*);

```
1       cv2.imwrite('output/Burkes/RGB/Normal_Order-' +
            filename_input, colorscript.Burkes(imageRGB,
            imageRGB.shape[0],imageRGB.shape[1],0))
```

Figure 6. A example of the main script accessing the *RGB.py* script and saving the output image.

As seen above, each one of the functions have the same parameters (*image, m, n and activator*). The *image* parameter receive the image file, the *m* and *n* receive the line and column size of the image, and the *activator* receive the information that select the scan direction, being 0 or 1 for the directions shown on the Figure 7(a) and 7(b) respectively.



(a) Normal Direction. (b) ZizZag Direction.

Figure 7. Scan Directions

The logic of each function for working with the **RGB** images (Figure 8) can be described by:

1. Create a error zero numpy array;
2. Create a zero integer matrix **S** of shape **(A,B, k)**;
3. Copy the pixels values from the original image to the matrix **S**;
4. Create a coefficient mask, with all the weights;
5. Obtains the lower and upper limits, and the flip mask activation;
6. Access each point (x,y) of the matrix **S**;
7. Save the old value of each channel (R,G,B) of the pixel (x,y);
8. Generates the new pixel value for each channel (R,G,B): 0 or 255 (White or Black);
9. Attaches the new pixel value of each channel (R,G,B) to the pixel (x,y);
10. Calculates the error of each channel (R,G,B) between the new and the old pixel value;
11. Verifies if the scan direction is ZigZag, if it's then flip the coefficient mask;
12. Apply the Error Diffusion Dotting Technique for each channel (R,G,B) of the pixel;

The logic of each function for working with the **Monochromatic** images (Figure 9) can be described by:

1. Create a zero integer matrix **S** of shape **(A,B)**;
2. Copy the pixels values from the original image to the matrix **S**;
3. Create a coefficient mask, with all the weights;
4. Obtains the lower and upper limits, and the flip mask activation;
5. Access each point (x,y) of the matrix **S**;
6. Save the old value of the pixel (x,y);
7. Generates the new pixel value: 0 or 255 (White or Black);
8. Attaches the new pixel value to the pixel (x,y);
9. Calculates the error between the new and the old pixel value;
10. Verifies if the scan direction is ZigZag, if it's then flip the coefficient mask;
11. Apply the Error Diffusion Dotting Technique;

```
1  error = np.zeros(3)
2  imgf = np.zeros(shape=(m+4,n+4,3)).astype(np.uint8)
3  copy_element(imgf,2,image)
4
5  mask_coef = np.array[[a, b, c, d],[e, w, z, k]]
6
7  for x in range(padding,m-padding):
8    lmax,lmin, final_mask = scan_direction(x,n,padding,
         activator)
9    for y in range(lmin,lmax):
10     old_pixel = imgf[x,y]
11     new_pixel = 255 * mp.floor(old_pixel/128.0)
12
13     imgf[x,y] = new_pixel
14
15     error = old_pixel - new_pixel
16
17     if(final_mask == 1):
18       mask_coef = np.fliplr(mask_coef)
19
20     #Apply error diffusion to adjacent pixels...
```

Figure 8. Logic used in the six Error Diffusion Dotting Techniques for RGB Images.

In the RGB image case, as said above, the first step was to create a error zero numpy array for saving all the errors information of each channel (Red, Green and Blue) when we accessed each pixel with the **for** loop (Figure 10).

Due to the image boundary, care must be taken with its edges. Passing a technique (also called "*masks*") above the edges, depending of the mask size, it can be a problem. Parts of the mask can go beyond the boundary giving an error.

To solve this problem, one of the solutions was to create a new matrix full of zeros of size **A x B** (N times bigger than the original image) and copy the values of the image to it. By doing this, we will have the same image with just a few extra *K* rows and/or columns with zero value. So when the mask goes over the edge there will be no errors. This "extra number" of rows and columns full of zeros is called *padding*.

```
imgf = np.zeros(shape=(a,b)).astype(np.uint8)

change_matrix_size(imgf,padding,image)

mask_coef = np.array[[a, b, c, d],[e, w, z, k]]

for x in range(padding,m-padding):
  lmax,lmin, final_mask = scan_direction(x,n,padding,
      activator)
  for y in range(lmin,lmax):
    old_pixel = imgf[x,y]
    new_pixel = 255 * mp.floor(old_pixel/128.0)

    imgf[x,y] = new_pixel
    error = old_pixel - new_pixel

    if(final_mask == 1):
      mask_coef = np.fliplr(mask_coef)

    #Apply error diffusion to adjacent pixels...
```

Figure 9. Logic used in the six Error Diffusion Dotting Techniques for Monochromatic Images.

```
        error = np.zeros(3)
```

Figure 10. Error Zero numpy array variable for the RGB case.

If for example, there is a **(512 x 512)** size image, and a **(3 x 3)** size mask. This mask when crossing the border will extrapolate the image boundary with 2 rows and columns. So if you create a zero full matrix of size (M + 4, N + 4), after copying the values from the original image to the matrix, you will have the same image with 2 rows and columns left.

Following this idea, we have for each technique:

- Floyd and Steinberg: the new matrix size is **(M + 2) X (N + 2)**, and the *padding* is equal to 1;
- Burkes: the new matrix size is **(M + 4) X (N + 4)**, and the *padding* is equal to 2;
- Sierra: the new matrix size is **(M + 4) X (N + 4)**, and the *padding* is equal to 2;
- Stuki: the new matrix size is **(M + 4) X (N + 4)**, and the *padding* is equal to 2;
- Jarvis, Judice and Ninke: the new matrix size is **(M + 4) X (N + 4)**, and the *padding* is equal to 2;
- Stivenson and Arce: the new matrix size is **(M + 6) X (N + 6)**, and the *padding* is equal to 3;

In the RGB case, the zero numpy matrix created, called *imgf*, has the shape **(A,B,k)**, i.e., it's a three dimensional matrix, each element as a zero numpy array of size 3 (each element for each channel (R,G,B)) (Figure 11).

In the monochromatic case, a two dimensional zero numpy array was created, with the same name as the RGB case, of shape **(A,B)** (Figure 12)

```
1   imgf = np.zeros(shape=(m+2,n+2,3)).astype(np.uint8)
```

Figure 11. An example of the zero numpy array matrix for the RGB case, at Floyd and Steinberg method.

```
1   imgf = np.zeros(shape=(m+4,n+4)).astype(np.uint8)
```

Figure 12. An example of the zero numpy array matrix for the monochomatic case, at the Floyd and Steinberg method.

In both cases, the next step was to copy all the elements of the original image to the new matrix created using the *copy_element()* function (Figure 13). In this function, we copied the value from the point *(padding,padding)* until the *(A - padding, B - padding)*. As shown in the figure 13, where *matrix* is the **imgf** matrix created and *copymatrix* is the original image.

```
1   matrix[padding:(matrix.shape[0] - padding),
2       padding:(matrix.shape[1] - padding)] = copymatrix[:]
```

Figure 13. ***copy_element() function***

For an easier manipulation, later it was created a matrix, called *mask_coef* that contains all the coefficients(weights) of the specific technique (Figure 14).

```
1   mask_coef = np.array([[1, 1, 1, 8/32, 4/32],
2       [2/32, 4/32, 8/32, 4/32, 2/32]])
```

Figure 14. Example of the Coefficient mask of Burkes method.

For accessing each point (pixel) of the *imgf* matrix, it was used a **for** loop, where the scan direction is defined by the *scan_direction()* function, as seen at figure 8 and figure 9. The function receives the line position *x*, the number *n* of columns of the *imgf* matrix, the *padding* value and the *activator* (Figure 15).

If the activator is **1** then it means that is a ZigZag scan direction. So, when the line is even (multiple of 2) the scan starts at the end to the begging of the matrix, i.e., the lower limit (*lmin*) is (**A** - *padding*) and the upper limit (*lmax*) is *padding*. If it is not, the scan starts at the beginning to the end of the matrix, i.e., *lmin* is *padding* and *lmax* is (***A*** - *padding*).

But, if the activator is equal to **0**, then activate the Normal direction, where the scan starts at the beginning to the end of the matrix, i.e., *lmin* is *padding* and *lmax* is (***A*** - *padding*).

When we work with ZigZag scan direction it is necessary to flip (reverse) the coefficient mask. For that, the *scan_direction* function (Figure 15), returns also the flip mask activator value, called *flip_mask*. If it is **1**, then flip the mask. If it is **0**, then don't flip.

```
1   def scan_direction(x,n,padding,activator):
2
3     if(activator == 1):
4     #ZigZag Order
5       if (x%2) == 0:
6         lmax = padding
7         lmin = n - padding
8       else:
9         lmax = n - padding
10        lmin = padding
11
12      flip_mask = 1
13
14    #Normal Order
15    else:
16      lmax = n-padding
17      lmin = padding
18      flip_mask = 0
19
20    return lmax, lmin, flip_mask
```

Figure 15. ***scan_direction() function***

In the RGB case, after the scan direction process, for each pixel accessed, it was saved each channel (R,G,B) value at the *old_red_pixel, old_green_pixel* and *old_blue_pixel* variables (Figure 16), respectively. Then, it was calculated the new pixel value for each channel and saving them at the *new_red_pixel, new_green_pixel* and *new_blue_pixel* variables.

```
1           old_blue_pixel = imgf[x,y][0]
2           old_green_pixel = imgf[x,y][1]
3           old_red_pixel = imgf[x,y][2]
```

Figure 16. Saving the old value for each channel of the pixel in the RGB case.

In the monochromatic case, for each pixel accessed, it was saved their values at the *old_pixel* variable. Then, it was calculated the new pixel value and saving it at the *new_pixel* variable (Figure 17).

```
1           old_pixel = imgf[x,y]
```

Figure 17. Saving the olde value for each channel of the pixel in the Monochromatic case.

In both cases, to find the new pixel color, the old pixel value was divided by 128 (the middle value of the scale 0 to 255). The *math floor* function rounds the division result to the closer integer number. Then every value that is lower than 128, the result is zero (White). For every value bigger than 128, the result is 1 (Black). But, since we are working with 0 or 255 values, we multiply by 255. So, at the end the new pixel value will be 0 or 255, exactly (Figure 18).

```
1       new_red_pixel = 255 * mp.floor(old_red_pixel/128.0)
```

Figure 18. Calculation used to generate new value for the pixel in the RGB and the Monochromatic case.

For the last step, the new pixel value was transferred to the exact point of the *imgf* matrix, and it was calculated the error (Figure 19 and 21 for RGB, and Figure 20 and Figure 22).

```
1   imgf[x,y][0] = new_blue_pixel
2   imgf[x,y][1] = new_green_pixel
3   imgf[x,y][2] = new_red_pixel
```

Figure 19. Transferring the new pixel value to the exact point of the *imgf* matrix for the RGB case.

```
1            imgf[x,y] = new_pixel
```

Figure 20. Transferring the new pixel value to the exact point of the *imgf* matrix for the Monochromatic case.

```
1   error[0] = old_blue_pixel - new_blue_pixel
2   error[1] = old_green_pixel - new_green_pixel
3   error[2] = old_red_pixel - new_red_pixel
```

Figure 21. Error Calculation for the RGB case.

```
1            error = old_pixel - new_pixel
```

Figure 22. Error Calculation for the Monochromatic case.

To finish, in both cases, if the scan direction was the ZigZag, than we used the *fliplr* function from the Numpy library to flip the coefficent mask (Figure 23). This function reverse each line of the given matrix.

```
1            if (flip_mask == 1):
2           mask_coef = np.fliplr(mask_coef)
```

Figure 23. *fliplr()* function.

Ending the flip, we just applied the Error Diffusion Technique, verify if the distribution don't extrapolated the 0 or 255 limit and then returned the final image.

For analysing if the new value attributed at the error diffusion was in the boundary, in both cases (RGB and Monochromatic), it was used the function *value_boundary()*. If the new vale was bigger than 255, then change the value to 255, and if it is below than 0, then change the value to 0 (Figure 24).

```
1  def value_boundary(value):
2    if value > 255:
3      value = 255
4    if value < 0:
5      value = 0
6
7    return value
```

Figure 24. *value_boundary()* function.

3.3.1. Error Diffusion Dotting Techniques for Monocromatic Images

For Monochromatic images, it was needed to apply the Error Diffusion to all adjacent pixels stipulated by the technique being used.

The figure bewlow has a example taken from the *monochromatic.py* script:

```
1  imgf[x+1,y] = value_boundary(imgf[x+1,y] + ((mask_coef
       [1,1]) * error))
2  imgf[x,y+1] = value_boundary(imgf[x,y+1] + ((mask_coef
       [0,2]) * error))
3  imgf[x+1,y-1] = value_boundary(imgf[x+1,y-1] + ((
       mask_coef[1,0]) * error))
4  imgf[x+1,y+1] = value_boundary(imgf[x+1,y+1]  + ((
       mask_coef[1,2]) * error))
```

Figure 25. An example of the Floyd and Steinberg method for Monochromatic Images.

3.3.2. Error Diffusion Dotting Techniques for RGB Images

For RGB images, we need to apply the Error Diffusion to all the channels Red, Green and Blue of each adjacent pixels stipulated by the technique being used.

The figure 26 has a example taken from the *monochromatic.py* script:

3.4. Results and Conclusion

After performing all the test using a few amount of RGB and Monochromatic images, it was analysed:

1. The difference between the results images, RGB and monochromatic, generated from each technique;
2. The difference between the results images of each scan direction;

After applying the different techniques to a significant set of images, a certain similarity between the techniques was observed, except for the Stivenson and Arce method. In this last method, there was a great difference in the obtained results, bringing more degraded images than those obtained in other techniques.

This difference can be given by the high value of the mask weights and their error dispersion form, which is made more widely.

```
1  imgf[x+1,y][0] = value_boundary(imgf[x+1,y][0] + ((
       mask_coef[1,1]) * error[0]))
2  imgf[x+1,y][1] = value_boundary(imgf[x+1,y][1] + ((
       mask_coef[1,1]) * error[1]))
3  imgf[x+1,y][2] = value_boundary(imgf[x+1,y][2] + ((
       mask_coef[1,1]) * error[2]))
4
5  imgf[x,y+1][0] = value_boundary(imgf[x,y+1][0] + ((
       mask_coef[0,2]) * error[0]))
6  imgf[x,y+1][1] = value_boundary(imgf[x,y+1][1] + ((
       mask_coef[0,2]) * error[1]))
7  imgf[x,y+1][2] = value_boundary(imgf[x,y+1][2] + ((
       mask_coef[0,2]) * error[2]))
8
9  imgf[x+1,y-1][0] = value_boundary(imgf[x+1,y-1][0] + ((
       mask_coef[1,0]) * error[0]))
10 imgf[x+1,y-1][1] = value_boundary(imgf[x+1,y-1][1] + ((
       mask_coef[1,0]) * error[1]))
11 imgf[x+1,y-1][2] = value_boundary(imgf[x+1,y-1][2] + ((
       mask_coef[1,0]) * error[2]))
12
13 imgf[x+1,y+1][0] = value_boundary(imgf[x+1,y+1][0]  + ((
       mask_coef[1,2]) * error[0]))
14 imgf[x+1,y+1][1] = value_boundary(imgf[x+1,y+1][1]  + ((
       mask_coef[1,2]) * error[1]))
15 imgf[x+1,y+1][2] = value_boundary(imgf[x+1,y+1][2]  + ((
       mask_coef[1,2]) * error[2]))
```

Figure 26. An example of the Floyd and Steinberg method for RGB Images.

Comparing the scanning forms of the image, it was found that scanning in normal order gives a higher quality image than a scan in ZigZag mode.

Among the images tested, here are some examples of the results obtained by applying the Sierra technique whit the ZigZag scan to the Pappers and Monalisa images.

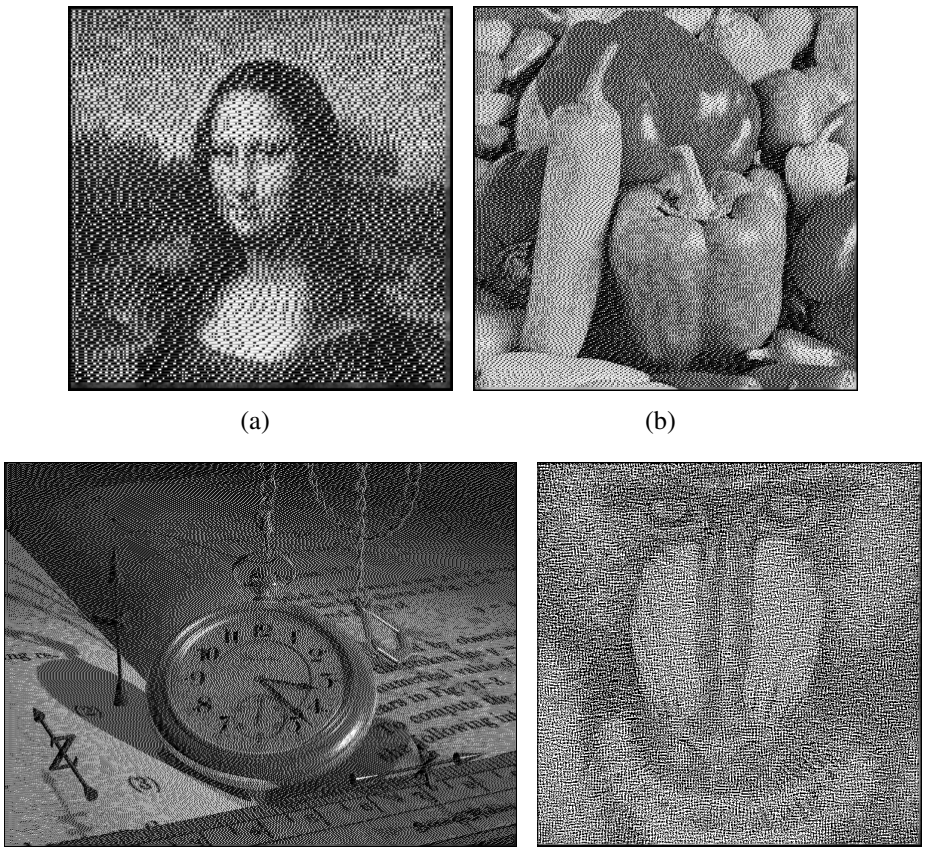


Figure 27. Results given by applying the Sierra Method for Monochromatic images: (a) Monalisa image; (b) Papper image; (c) Watch image; and, (d) Baboon image.

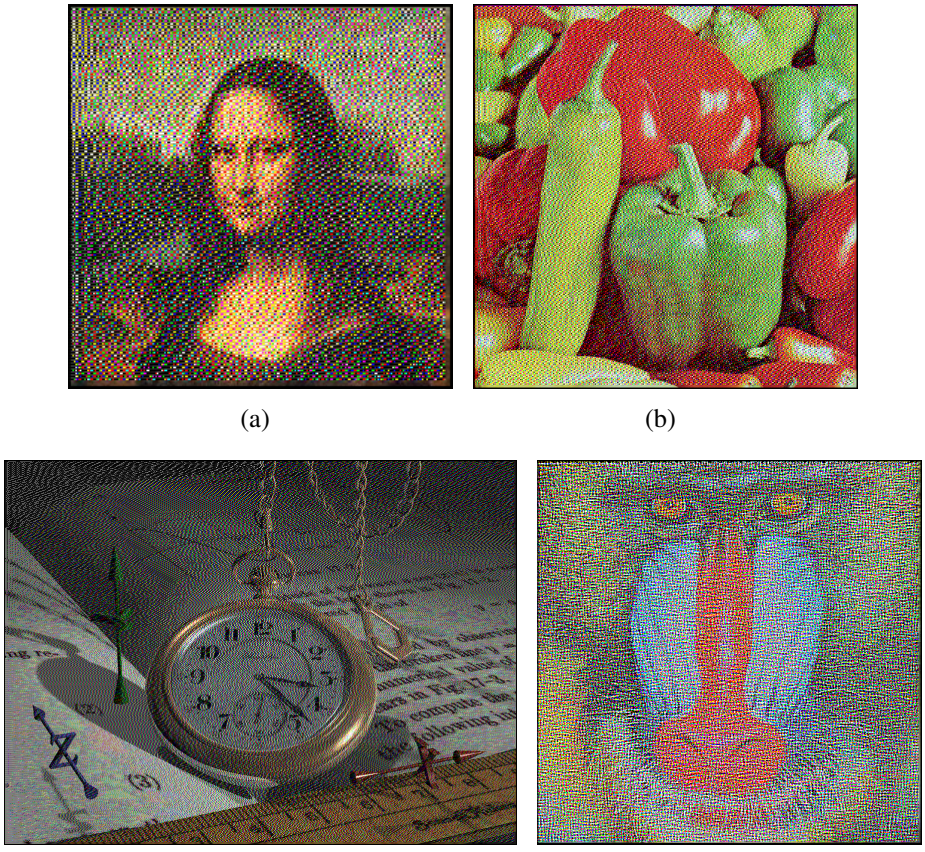


Figure 28. Results given by applying the Sierra Method for Monochromatic images: (a) Monalisa image; (b) Papper image; (c) Watch image; and, (d) Baboon image.