Project 1 - Halftoning

Thales Mateus Rodrigues Oliveira (RA 148051)

***Abstract*—In this project, it was given the task of creating a halftoning algorithm to generate images in grayscale and color. To fulfill the requirements, the solution applies error diffusion for neighbor pixels on image sweeping, using approaches listed in the literature. It was able to generate the halftoning images in grayscale and color, and the differences of each error diffusion approach, the pros and cons of the implementation are explained in this report.**

I. Introduction

Halftoning is a technique used for reducing the number of colors used to represent an image, while is desired to keep a good visual perception of its contents for the user. In this work, the implementation of halftoning techniques with error diffusion was realized. For each technique, tests were executed to analyze the quality of the output image, sweeping the image in two different ways. The next sections explains the implemented algorithm, the experiments realized and the output analysis.
The code, along with the input files and the report is delivered in the compressed file THALES_MATEUS_RODRIGUES_OLIVEIRA_148051.tar, in the Google Classroom.

II. The Program

The program was implemented with Python 3.7.3. The libraries used and their respective versions are OpenCV 4.1.0 and Numpy 1.16.4.

A. How to execute it

The project has a Makefile available to help performing some actions on it. The Makefile has 3 basic instructions: clean, build and exec. Clean instruction removes generated images stored in the **output** folder, the execution code in **bin** folder and the folders itself. The Build instruction creates the **output** and **bin** folders, and moves the source code to **bin**. The Exec instruction executes the code with images in the **input** folder. Listing 1 provides examples of how to execute the three instructions in a terminal.

```
#clean environment, deletes output and bin folders and their content
make clean

#prepare the environment for code execution
make build

#executes code
make exec
```

Listing 1. Makefile usage example

B. Input

The program does not have an input argument by default, the input images are listed in code, and they are expected to be stored in the **input** folder. Listing 2 shows how images are listed to be executed in code. The $images$ tuple is implemented in $src/main.py$

```
# for inserting other images, add tem to /input folder and list them here
images = (
    'baboon',
    'monalisa',
    'peppers',
    'watch'
)
```

Listing 2. Input images inside code

C. Output

The output of the program is a series of halftoning images based on the input ones, changing the error diffusion methods and the sweeping directions. The output images are stored in the **output** folder, and they are labeled by concatenating the image name, whether is colored or grayscale, the error diffusion method and the sweep order (*e.g.: baboon_colored_sierra_left-to-right.png*)

D. Implementation

The function which implements the halftoning operation is defined in the *src/halftoning.py* file. The file contains a dictionary that lists the approaches used for error diffusion, as mentioned in Figure 1 in the project proposal. Listing 3 shows the implemented dictionary.

```
# Masks used for error propagation
MASKS = {
    "floyd-steinberg": np.array([
      [0, 0, 7/16],
      [3/16, 5/16, 1/16]]),
    "stevenson-arce": np.array([
      [0, 0, 0, 0, 0, 32/200, 0],
      [12/200, 0, 26/200, 0, 30/200, 0, 16/200],
      [0, 12/200, 0, 26/200, 0, 12/200, 0],
      [5/200, 0, 12/200, 0, 12/200, 0, 5/200]]),
    "burkes": np.array([
      [0, 0, 0, 8/32, 4/32],
      [2/32, 4/32, 8/32, 4/32, 2/32]]),
    "sierra": np.array([
      [0, 0, 0, 5/32, 3/32],
      [2/32, 4/32, 5/32, 4/32, 2/32],
      [0, 2/32, 3/32, 2/32, 0]]),
    "stucki": np.array(
      [[0, 0, 0, 8/42, 4/42],
      [2/42, 4/42, 8/42, 4/42, 2/42],
      [1/42, 2/42, 4/42, 2/42, 1/42]]),
    "jarvis-judice-ninke": np.array([
      [0, 0, 0, 7/48, 5/48],
      [3/48, 5/48, 7/48, 5/48, 3/48],
      [1/48, 3/48, 5/48, 3/48, 1/48]])
```

ra148051@students.ic.unicamp.br

```
  }
```

Listing 3. Masks used for error diffusion

The function *apply_halftoning* implements the desired operation. It receives the original image, the name of the error diffusion approach, the sweep method (left to right or alternated), and a benchmarking flag to monitor execution time. Listing 4 shows keyparts of implementation

```
def apply_halftoning(img, err_method="floyd-steinberg", sweep_method=1, benchmarking=False):
  # initialize result array
  result = np.zeros_like(img)
  # separates the masks that could be used
  # (it needs the flip version of mask for alternated sweep)
  m = (0, MASKS[err_method], np.flip(MASKS[err_method], 1))
  # saves mask dimensions to be used when needed
  mask_h, mask_w = m[1].shape
  # saves image dimensions to be used when needed
  img_h, img_w = img.shape
  # it holds the offset of manipulated pixel related to mask
  offset = mask_w//2
  # applies padding to image to make it easier the operations with mask
  img_padded = np.pad(img, ((0, mask_h - 1), (mask_w//2, mask_w//2)), 'constant')

  # default starting direction (left to right)
  direction = 1
  # default index values for sweeping from left to right and right to left
  sweep_options = (0, (0, img_w), (img_w - 1, 0))

  # it sweeps the image from top to bottom
  for j in range(img_h):
      # it sweeps the image from left to right or right to left depending on direction
      beginning, end = sweep_options[direction]
      # do the horizontal sweeping
      for i in range(beginning, end, direction):
          # depending on analyzed pixel, set its result value according to threshold
          if img_padded[j][(i + offset)] < 128:
              result[j][i] = 0
          else:
              result[j][i] = 1

          # calculates associated error
          error = img_padded[j][(i + offset)] - result[j][i]*255
          # propagates error according to mask
          img_padded[j:j+mask_h, i:i+mask_w] = (img_padded[j:j+mask_h, i:i+mask_w]
                                                + (error*m[direction])).astype(np.uint8)
      # it changes from left to right to right to left (vice-versa) depending on the sweep method
      direction *= sweep_method

  # scale the result
  return result*255
```

Listing 4. Implementation of halftoning solution

The decisions made in the implementation are explained in the following part.

1) Dealing with sweeping directions: In this solution, it was implemented two approaches: sweeping every line from left to right, and alternating directions when changing from one line to the other. Figure 1 shows the approaches. If the second mentioned method is chosen, the error diffusion mask has to be mirroed in the vertical direction. In order to avoid *if-else* statements when deciding which mask to use for the specific line (the regular or the mirroed one), the tuple *m* holds both masks, and decides which one to use based on the *direction* flag (if *direction* is 1, use the regular mask. If it is -1, use the mirroed one). To sweep the image from right to left, it is also considered the indexes in the loop to decrease. Following the same idea of tuple *m*, the tuple *sweep_options* is used to set the starting index, the stop index, and the *direction* flag as step.

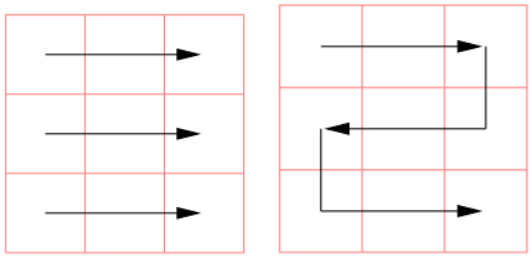

Fig. 1. Left: Sweeping from left to right. Right: Sweeping alternated

2) Image Padding to deal with borders: In order to deal with pixels in the border of the image, to diffuse the error, the image is padded as the mask propagates the error to its neighbors, as they may not exist for pixels in the border. The padding is done based on the mask properties, such as not diffusing errors to already visited pixels and the dimensions of each mask. The values padded do not matter, as they are not visited.

3) Defining the value of the pixel and the error diffusion: The main ideia of the halftoning algorithm is the following: rather than a simple binarization, in order to create the ”feeling” of having more than 2 intensities of a color in the image, an analyzed pixel propagates its information to specific close neighbors. This propagation of information is what we define as error diffusion. Based on a threshold which defines whether a pixel has the maximum or minimum intensity, the resulting pixel value is defined and the error is calculated and passed through specific neighbors with specific weights. That weight is what is defined in the error diffusion masks. The error diffused for the neighbor pixels is proportional to the mentioned weight. After performing these steps to all pixels in the image, the process is done. The steps are the ones inside the two *for* statements in listing 4. They were implemented based on Algorithm 5, presented on the *Enhancement* class of MO443: Introduction to Digital Image Processing course[1].

4) Sweeping the image: To perform the visit of every pixel, as seen in listing 4, it is used two nested *for* statements. As the author was unable to vectorize the error propagation, the execution time occurs on the second scale. To define the value of the analyzed pixel in the iteration, an offset is used, as the pixel is in the middle column related to mask.

III. Experiments

The *src/main.py* file executes the test pipeline. The idea is the following: for each input image, for each error propagation

method, for each sweep order, generate the grayscale and colored halftoning output images. The input images are stored in **input** folder as mentioned before, and their names and dimensions are listed in table I. The input images are shown in figure 2

Image Names	Dimensions (width x height)
baboon_colored.png	512 x 512
monalisa_colored.png	256 x 256
peppers_colored.png	512 x 512
watch_colored.png	Vertical 1024 x 768

TABLE I
INPUT IMAGES USED IN EXPERIMENTS

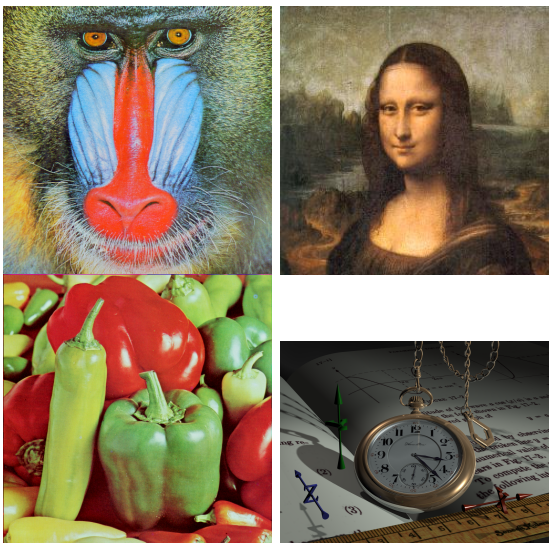

Fig. 2. Input images used in experiments. a) Baboon. b) Monalisa c) Peppers d) Watch

The error diffusion approaches are taken from the literature[1]. The listing 3 indicates their dimensions and weights. Table II lists the approaches names and the masks dimensions.

Error diffusion method	Mask Dimensions (width x height)
Floyd and Steinberg	3 x 2
Stevenson and Arce	7 x 4
Burkes	5 x 2
Sierra	5 x 3
Stucki	5 x 3
Jarvis, Judice and Ninke	5 x 3

TABLE II
ERROR DIFFUSION METHODS USED IN EXPERIMENTS

As we have 4 input images, 6 error diffusion methos, 2 sweep modes and it generates 2 output images (grayscale and colored), we have 96 images of output. The output images are stored in the **output** folder.

IV. DISCUSSION

This section is organized in four parts. The first part analyses the output images when compared to the original input. The second part takes into consideration the differences of sweeping modes. The third one does the comparisons between the error propagation methods. To sum up, the pros and cons of the implementation.

A. *Effectiveness of the technique*

In order to check if the premise of the implementation was achieved, the comparison between input and output images is needed. The figures 3, 4, 5, 6 places input and output images together, for all the input images used in the experiments, showing a variety of methods and sweeping orders.

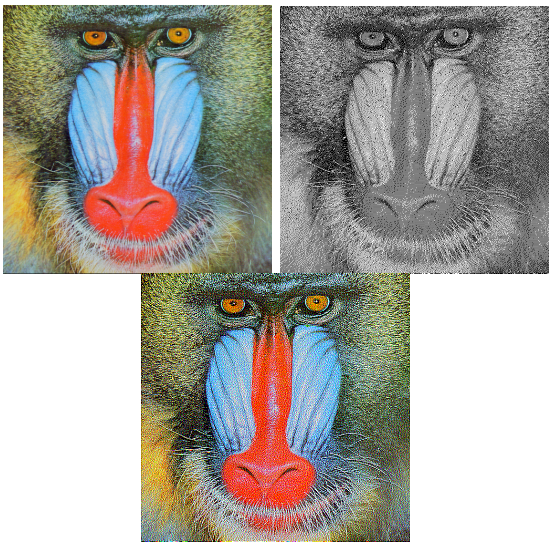

Fig. 3. Baboon images manipulated in experiments. a) Input. b) grayscale by using Floyd and Steinberg, left to right c) colored by using Jarvis, Judice and Ninke, alternate

It is interesting to notice, from those images, that the output images are closely related to the input images, even though it is used only the maximum and minimum values of grayscale and color channels. The error diffusion works well in applying the feeling of having multiple intensities of colors, notwithstanding we can still see points or regions with points in the output (*e.g.:* monalisa (grayscale and colored), baboon (grayscale), peppers (colored)).

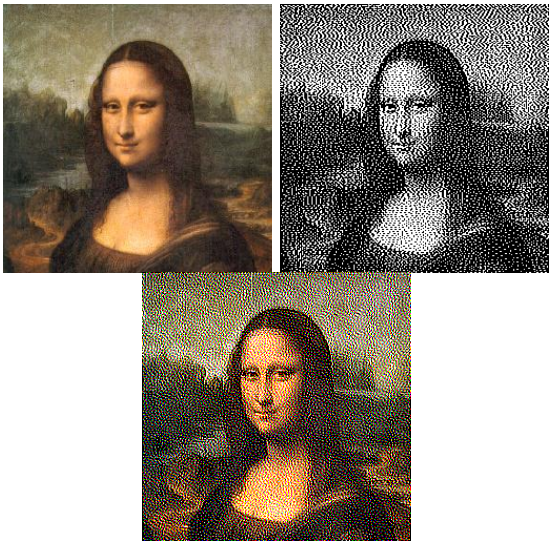

Fig. 4. Monalisa images manipulated in experiments. a) Input. b) grayscale by using Stevenson and Arce, alternate c) Colored by using Sierra, left to right

Fig. 5. Peppers images manipulated in experiments. a) Input. b) Grayscale by using Stucki, left to right c) Colored by using Burkes, alternate

Furthermore, the output images maintain the contrast seen in the input ones. It is clear that the error diffusion generates outputs which differs a lot of the process of binarization, in which necessary information of image is lost from the quantization according to a specific threshold.

Fig. 6. Watch images manipulated in experiments. a) Input. b) Grayscale by using Jarvis, Judice and Ninke, left to right. c) Colored by using Floyd and Steinberg, alternate.

B. Sweeping mode differences

For sweeping mode comparisons, an ideia is to place side by side output images related to the same input and same error diffusion method, and comparing them visually. As the report does not include the images in its real dimensions, it is encouraged to the reader to access the images in the **output** folder for further inspection. Figures 7, 8, 9 and 10 shows 4 output images for every input image, created using the two sweep methods, in grayscale and color.

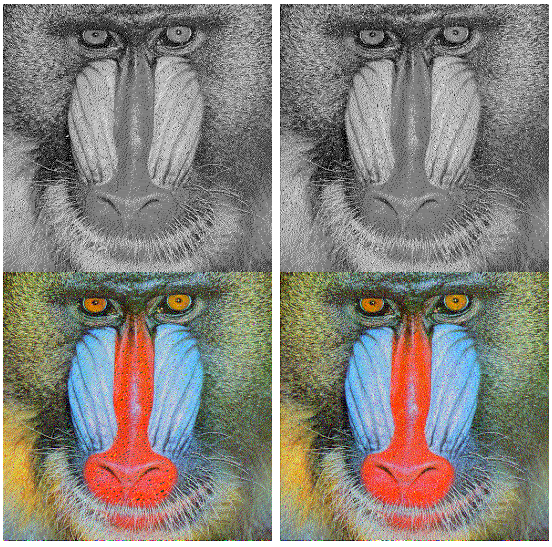

Fig. 7. Baboon images manipulated in experiments. a) Grayscale by using Floyd and Steinberg, left to right. b) Grayscale by using Floyd and Steinberg, alternate. c) Colored by using Floyd and Steinberg, left to right. d) Colored by using Floyd and Steinberg, alternate.

The differences are subtle, which makes it difficult to notice in the report, but by inspecting the original ones it is possible to see, for the baboon image input, that the sweep order from left to right creates more visible error points (*e.g.:* yellow and green-*ish* points are bigger in the nose of the baboon) than the alternate sweep. For their grayscale versions, the differences are harder to notice but the same effect is observable (if image is *zoomed-in*).

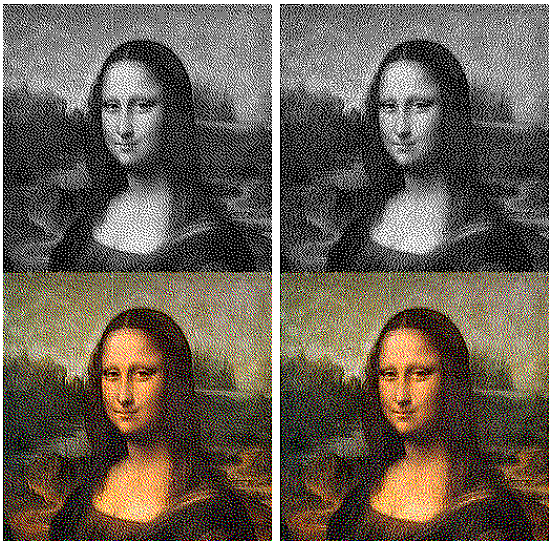

Fig. 8. Monalisa images manipulated in experiments. a) Grayscale by using Stucki, left to right. b) Grayscale by using Stucki, alternate. c) Colored by using Stucki, left to right. d) Colored by using Stucki, alternate.

As the monalisa image input is smaller compared to the other ones, it is difficult to spot any differences between the sweep methods.

Fig. 9. Peppers images manipulated in experiments. a) Grayscale by using Stevenson and Arce, left to right. b) Grayscale by using Stevenson and Arce, alternate. c) Colored by using Stevenson and Arce, left to right. d) Colored by using Stevenson and Arce, alternate.

In the peppers case, the differences are more visible in the colored examples. As the background is black, the error diffusion is clearly spotted in the bottom of the image (green blur). For the left to right sweep, the blur are is bigger than in the alternate sweep, and it has the ”bottom-to-right” direction, as expected from the mask characteristics and the sweep order. For the alternate sweep, the blur area is more ”self-contained”, as the propagation happens from left to right and right to left, so the error is diffused to the boundaries of the pepper itself, giving the feeling of more correctude. The error propagation is also clearer in this image for the grayscale cases, as we can see from the white spots in the black background.

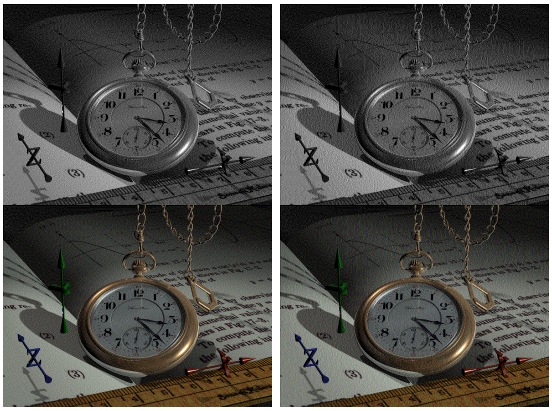

Fig. 10. Watch images manipulated in experiments. a) Grayscale by using Burkes, left to right. b) Grayscale by using Burkes, alternate. c) Colored by using Burkes, left to right. d) Colored by using Burkes, alternate.

The same effects noticed in the peppers image appears here for the colored output. Some error is spotted in the ”Z arrow” in the image for left to right sweep, while the same error does not appear in the alternate one. Surprisingly, for the grayscale output and Burkes mask, the written text in the left-to-right image has better quality than in the alternate one.

In the majority of cases, the alternate sweep obtained greater quality results, as it tries to compensate the error propagation to both directions.

C. Error Diffusion Methods

When analyzing the approaches listed in the literature to solve the problem, it is interesting to analyze them side-by-side. To do so, it is chosen colored or grayscale versions of a specific output, and a fixed sweep method, in this case, alternate. Then all solution proposals are grouped. Figures 11, 12, 13 and 14 shows 6 output images for each input, alternating the error diffusion methods.

Fig. 11. Baboon images manipulated in experiments. a) Colored by using Floyd and Steiberg, alternate. b) Colored by using Stevenson and Arce, alternate. c) Colored by using Burkes, alternate. d) Colored by using Sierra, alternate. e) Colored by using Stucki, alternate. f) Colored by using Jarvis, Judice and Ninke, alternate.

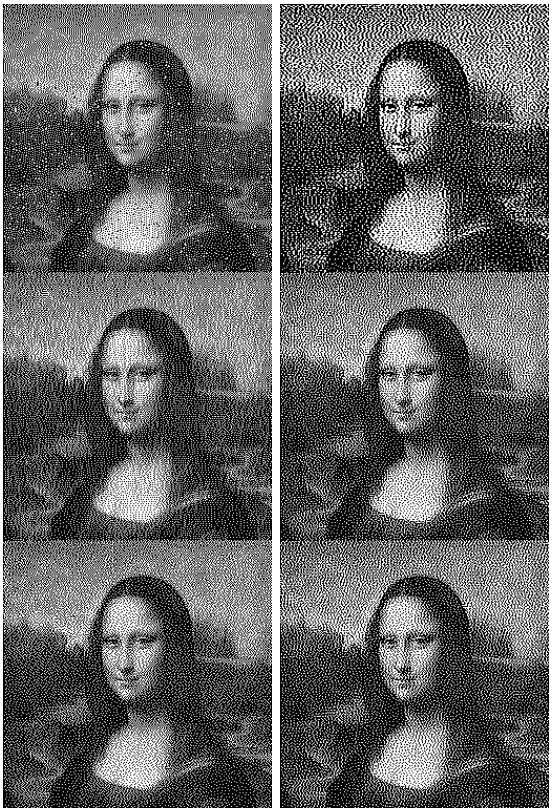

Fig. 12. Monalisa images manipulated in experiments. a) Grayscale by using Floyd and Steiberg, alternate. b) Grayscale by using Stevenson and Arce, alternate. c) Grayscale by using Burkes, alternate. d) Grayscale by using Sierra, alternate. e) Grayscale by using Stucki, alternate. f) Grayscale by using Jarvis, Judice and Ninke, alternate.

For all error diffusion approaches, the main objective is achieved, but each of them have their peculiarity. With Floyd and Steinberg method, we can see the error more clearly as color points that don't fit to their background (yellow and green points in baboon nose, red points in its blue face). Stevenson and Arce highlights the contours, while we have the similar visibility of errors of Floyd and Steinberg. For Burkes, the error is less noticeable, and the contours of baboon face are atenuated a bit, as the mask is short. For Stucki, Sierra and Jarvis, Judice and Ninke, the visibility of errors is also reduced, and the contours are more visible, as the mask is bigger.

For Monalisa, as the image is smaller, the differences are more difficult to grasp. But is is observable that the Floyd and Steinberg image has more noise than the other ones, and the Stevenson and Arce gives the feeling of having greater contrast.

The pepper output images have the same characteristics of the baboon image, already listed.

For the watch images, all of them reproduces the image well, being able to distinguish their details, even though the noise feeling is greater in the Floyd and Steinberg approach. All of them fail to reproduce the black background correctly.

Fig. 13. Peppers images manipulated in experiments. a) Colored by using Floyd and Steiberg, alternate. b) Colored by using Stevenson and Arce, alternate. c) Colored by using Burkes, alternate. d) Colored by using Sierra, alternate. e) Colored by using Stucki, alternate. f) Colored by using Jarvis, Judice and Ninke, alternate.

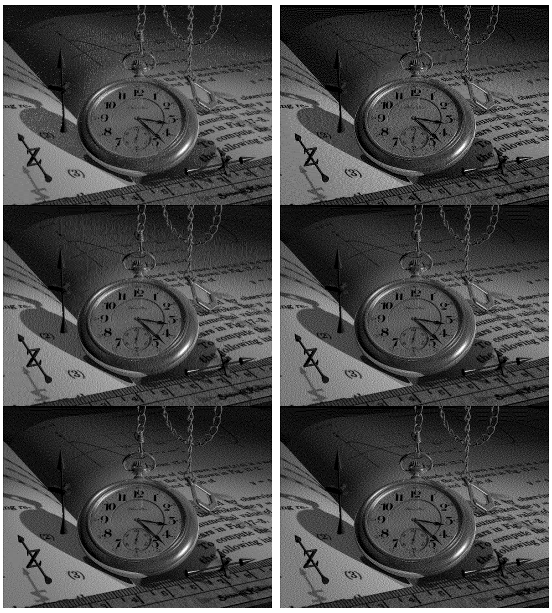

Fig. 14. Watch images manipulated in experiments. a) Grayscale by using Floyd and Steiberg, alternate. b) Grayscale by using Stevenson and Arce, alternate. c) Grayscale by using Burkes, alternate. d) Grayscale by using Sierra, alternate. e) Grayscale by using Stucki, alternate. f) Grayscale by using Jarvis, Judice and Ninke, alternate.

D. Pros and Cons

All methods achieve the objective of representing the original image characteristics with only two intensities per channel, each one having its own peculiarity. Floyd and Steinberg images have the least visible quality, as the propagated error is visible in the output, giving the feeling of being *noisy*. Stevenson and Arce images are very interesting, as they highlights the contours of objects in the image. The other methods proposed generate images with great quality, not interfering a lot in its original contours. In general, the technique performs bad when propagating error to uniform spaces, as the blackbackgrounds in our experiments.

Enabling the benchmarking flag, it was able to check the execution time for every output image. For each image channel, the execution time varied from 0.7 seconds for the monalisa image (smallest dimensions) to 9 seconds for the watch image (biggest dimensions), executing in a Intel(R) Core(TM) i7-4720HQ CPU @ 2.60GHz processor. As the technique sweeps through every pixel to do the error diffusion, the execution time is penalyzed.

V. Conclusion

The application of halftoning techniques to generate output images with similar characteristics of their inputs with only two levels of color intensity per channel was possible thanks to error propagation through image sweeping.

The usage of different input images, error diffusion techniques and sweep modes made it possible to analyze each approach's strength.

References

[1] H. Pedrini, "Introduction to digital image processing, enhancement class," pp. 130–135, 2019.