State University of Campinas
Institute of Computer Science
MO443 - Professor: Hélio Pedrini

Project 1

April 14, 2021

Pedro Henrique Vaz Valois
p265676@dac.unicamp.br

***Abstract*—Digital images are abundant in the current age. Be it as means of modern art or source of data, their importance grew in parallel to the dependence humanity has on it. Thus, the need for processing, transforming and understanding such images has risen and a multitude of techniques were developed for these tasks. In this project, we perform two linear transformations on RGB images and apply 10 kernel filters on monochromatic figures.**

I. Background

All code for this project has been developed and tested with `Python 3.7.9`, `numpy 1.20.2` and `matplotlib 3.4.1`.

We work the solution using tensor notation on most equations. A monochromatic image is defined as a rank 2 tensor (matrix)

$$C = (C_m n), 0 \leq m \leq M-1, 0 \leq n \leq N-1$$

where M is the image pixel width, N is the image pixel height and $C_m n$ indicates the level of luminance in the $[0, 255]$ range for the pixel in position (m, n).

Moreover, a RGB image is defined as a rank 3 tensor

$$C = (C_i jk), 0 \leq i \leq M-1, 0 \leq j \leq N-1, 0 \leq k \leq 2$$

where k indicates the channel index (red, green or blue) and all considerations for monochromatic image are equally valid for RGB ones.

II. Solutions

1.1 Full Color Images

a: The first task asks the following transformation to be performed onto a RGB image:

$$\begin{aligned} R' &= 0.393R + 0.769G + 0.189B \\ G' &= 0.349R + 0.686G + 0.168B \\ B' &= 0.272R + 0.534G + 0.131B \end{aligned}$$

We can restructure this linear system as the matrix-vector multiplication

$$\begin{bmatrix} R' \\ G' \\ B' \end{bmatrix} = \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

that can be denoted as $\vec{v'} = S\vec{v}$, where $\vec{v'} = (R', G', B')$, $\vec{v} = (R, G, B)$ and S is the transformation matrix, where

$$S = \begin{bmatrix} 0.393 & 0.769 & 0.189 \\ 0.349 & 0.686 & 0.168 \\ 0.272 & 0.534 & 0.131 \end{bmatrix}$$

Therefore, for a RGB image $C = C_{ijk}$, this transformation can be denoted as a matrix multiplication on each channel vector

$$C'_{ijk} = \sum_{m=0}^{2} S_{km} C_{ijm}$$

For implementation, we use `pyplot.imread` channels last configuration, that outputs the image array as

$$C_{i \times j \times m}$$

where m represents the channels axis.

Thus, we must align the right axes being multiplied when performing the matrix multiplication. In short, this means we must use the transpose of S as our final transformation.

$$C' = C_{i \times j \times m} S_{m \times k} = CS^T$$

In listing 1, we write the transpose of the transformation, apply the matrix multiplication and trim values beyond 1.

```python
import numpy as np

transformation = np.array([
    [.393, .349, .272],  # R
    [.769, .686, .534],  # G
    [.189, .168, .131],  # B
])

result = img @ transformation
result[result > 1] = 1  # apply threshold
```

Listing 1: image linear combination in Python

b: The transformation equation is now

$$I = 0.2989R + 0.5870G + 0.1140B$$

Similarly, we can see this transformation I as a matrix multiplication with the take that I is now a horizontal vector I_m. Thus, the equation for the final image C' is

$$C'_{ij} = \sum_{m=0}^{2} I_m C_{ijm} \implies C' = CI$$

In listing 2, we write the transformation as row vector and apply the matrix multiplication. Notice there is no need for trimming as $\sum I_m < 1$.

```
import numpy as np

transformation = np.array([.2989, .5870, .114])

result = img @ transformation
```

Listing 2: single channel image transformation in Python

1.2 Monochromatic Images

The filtering process of a monochromatic image $C = (C_{ij}) = C_{W \times H}$ with a matrix kernel $h = h_{M \times N}$ can be seen as the 2D convolution

$$(C * h)_{ij} = \sum_{p=i}^{i+M} \sum_{q=j}^{j+N} C_{pq} h_{pq}$$

where the pair i, j denote the pixel in the output image.

In the following sections, we will show how to implement a vectorized version of this sum in Python with `numpy`.

1) Performing one convolution: Listing 3 shows how to perform the convolution for one 3×3 square on the original image. This process must be repeated for all i, j pairs in order to produce the final output image.

```
import numpy as np

h = np.array([
    [-1, 0, 1],
    [-2, 0, 2],
    [-1, 0, 1],
])

M, N = h.shape
i = j = 0

result = np.dot(
    img[i:i + M, j:j + N].ravel(),
    h.ravel()
)
```

Listing 3: kernel filtering to produce one pixel

A. Padding

The convolution can reduce the size of the image on both width and height if the image dimensions are not divisible by the kernel dimensions. This can be seen as an unexpected side effect and the simplest solution is to add a null border around the image.

This process is denoted padding and the function `numpy.pad` will be used to build the border. The padding dimensions p_k can be determined by

$$\begin{aligned} dim(C_{pad})_k \mod dim(h)_k = 0 \implies \\ (dim(C)_k + 2p_k) \mod dim(h)_k = 0 \implies \\ p_k = dim(C)_k \mod 2dim(h)_k \end{aligned}$$

where $dim(T)$ is the dimension vector of array T and the factor 2 comes from the fact the padding is done on both sides.

```
import numpy as np

p = np.array(img.shape) % (2 * np.array(h.shape))
img_pad = np.pad(img, p)
```

Listing 4: image padding

B. Indexing

In sequence, we must reshape the image into an array of dimension $width * height \times M * N$ in order to perform the dot product in a vectorized manner.

The approach we take is to use multidimensional indexing on the `numpy.array`. `numpy` indexing enables mapping an array into a new shape by referencing multiple indices on each axis [1].

For instance, it is possible to flatten the first 3×3 block $\vec{a}$ of an array A with two array of indices

$$\begin{aligned} i = [0, 0, 0, 1, 1, 1, 2, 2, 2] \\ j = [0, 1, 2, 0, 1, 2, 0, 1, 2] \\ \implies \vec{a} = A[i, j] \end{aligned}$$

In this case, notice that `numpy` performs the cartesian product $i \times j$ to build $\vec{a}$. Moreover, i, j have each distinct types of repetitions, where i can be built with `numpy.repeat(3, 3)` and j with `numpy.tile(3, 3)`.

Listing 5 uses `numpy.repeat` and `numpy.tile` to build the indices i, j. The final indices both have dimensions $W * H \times M * N$. The result is a 2D-array where each row is a flattened version of each $M \times N$ contiguous block.

```
import numpy as np

M, N = h.shape
W, H = img.shape

i0 = np.repeat(np.arange(M), M)
i1 = np.repeat(np.arange(W), W)
i = i0.reshape(-1, 1) + i1.reshape(1, -1)

j0 = np.tile(np.arange(N), N)
j1 = np.tile(np.arange(H), H)
j = j0.reshape(-1, 1) + j1.reshape(1, -1)

result = img_pad[i, j]
```

Listing 5: reshaping image with multidimensional indexing

1) Matrix multiplication and reshaping: The last step is to perform the convolution. We originally have shown this step as a dot product, but the indexing from the previous step restructure it as a matrix multiplication followed by a reshape to the original dimension.

$$C * h = hC_{pad}[i, j]$$

Listing 6 shows the python code that actually performs the kernel convolution.

```
conv = kernel.ravel() @ img_pad[i, j]
result = conv.reshape(img.shape)
```

Listing 6: 2D convolution with padding and indexing and multiplication and reshaping

III. Implementation

The software that implements the solutions discussed in the las section is contained in the `script.py` file. It depends on `numpy` to perform the vectorized operations and `matplotlib` for reading and saving image files. Moreover, it uses python standard library's `os` module for checking if the output file already exists and the `argparse` module for parsing command line arguments.

The solutions were implemented in the functions `color_matrix_transformation`, `color_vector_transformation` and `grayscale_kernel_convolution`. The kernels are defined inside the function `get_kernels`. Furthermore, other helper functions were made to simplify the process.

The `argparse` module provides a help text for using the script. By typing `python script.py --help`, a helper message is shown in the console.

If the image is grayscale, `--kernel` specifies the kernel and combinations of kernels can be made with a + sign. Listing 7 shows an example.

```
python script.py gray.png out.png --kernel h1+h2
```

Listing 7: out.png is resulted from the kernel convolution of h1 and h2 filters combined on gray.png

On the other hand, if the image is RGB, the transformation can be specified with `--transformation matrix` (solution 1.1a) or `--transformation vector` (solution 1.1b). An example can be seen on Listing 8

```
python script.py fullcolor.png out.png --overwrite
    --transformation vector
```

Listing 8: out.png is overwritten with the result of the vector transformation on fullcolor.png

IV. Results and Discussion

1.1 Full Color Images

We test the implemented transformations on 4 different RGB images, shown in figure 1.

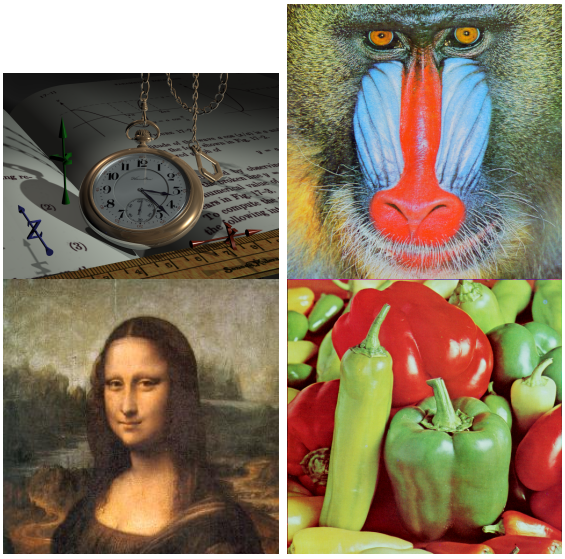

Figure 1: 4 RGB images used for testing the Full Color transformations

a: The matrix transformation can be performed on an image following the pattern shown in listing 9. The results are presented in figure 2 and, from them, we can see the transformation is the sepia toning transformation, that gives the images this warmer appearance.

```
python script.py color/monalisa.png out.png -w --
    transformation matrix
```

Listing 9: Applying the matrix transformation on the RGB image of monalisa.png

Figure 2: Matrix transformation applied on each RGB image. The resulting sepia effect gives the image an warmer tone.

b: The vector transformation can be performed on an image following the pattern shown in listing 10. The results are

presented in figure 3 and, from them, we can see the transformation is responsible for turning the image into grayscale.

More specifically, the vector transformation is the RGB luminance equation, that maps the RGB colors to RGB luminance, which is brightness adjusted to indicate appropriately what we really see [2]. This transformation gives darker tones to blue and red, while making green clearer.

```
python script.py color/monalisa.png out.png -w --
    transformation vector
```

Listing 10: Applying the vector transformation on the RGB image of monalisa.png

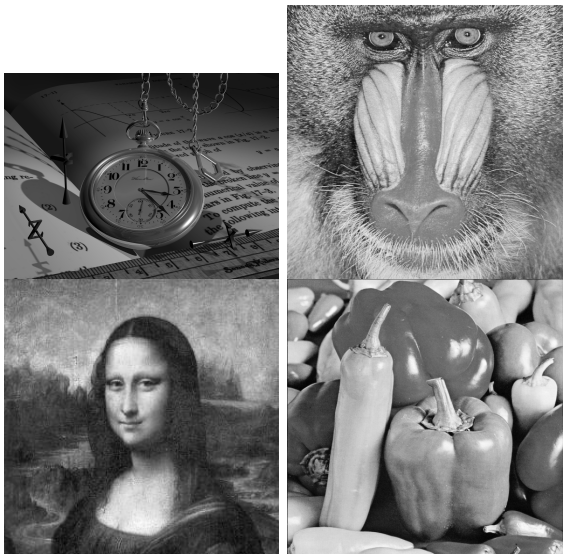

Figure 3: Vector transformation applied on each RGB image, turning them monochromatic

1.2 Monochromatic Images

We test the 8 implemented kernels on 4 different monochromatic images, shown in figure 4. Below, we explain the effect of each kernel for one of these images.

h1 & h2: Kernel $h1$ has a null column, thus highlighting vertical edges whereas kernel $h2$ has a null row, highlighting horizontal edges. Figure 5 shows this effect visually and uses the matplotlib seismic colormap to make it more visible.

h1 + h2: There are two ways to perform the the combinations of two filters: the first is to compute the final filter and use it in the image and the other is to compute intermediate images and then compute the final one. Each of these approaches have different results and we will discuss the in the follwoing paragraphs.

When kernels $h1$ and $h2$ are combined with $h1,2 = \sqrt{h1^2 + h2^2}$, we end up with a 3×3 matrix with a null center, thus ignoring the original pixel value and making it a combination of its surroundings. Moreover, $h1,2$ does not have negative values and isn't normalized (total sum > 1). Therefore, as visible in Figure 6, the resulting image receives a small blurry effect, more visible on the brick wall on

Figure 4: 4 monochromatic images used for testing the Kernel 2D Convolutions shown with the matplotlib gray colormap

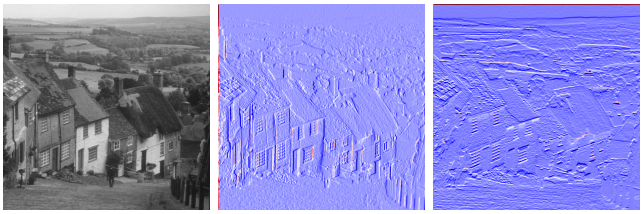

Figure 5: Effect of kernel h1 (middle) and h2 (right) on city.png (left) with the matplotlib seismic colormap

house.png. We emphasize this result is demonstrated here for comparison reasons and **cannot be generated using the code** that follow this text.

On the other hand, it is possible to combine these filters with $R = \sqrt{(C * h1)^2 + (C * h2)^2}$. In turn, this approach gives us a high contrast image with edges highlighted while the rest is much darker, as visible in Figure 7. This formula is the one used for the Sobel filter [3] and clearly gives much more value and information than the other approach, for it can be used as an edge detector. We emphasize that this result is the one **the code that follows this text is able to generate**.

h3 & h8: Kernels $h3$ and $h8$ give a similar effect as they both have a high central value surrounded by a low neighborhood. This leads to lost of contrast on the overall image, leaving only the highest frequencies visible while the rest tends toward an average gray level in the case of $h8$, as shown by Figure 8.

h4 & h9: Kernels $h4$ and $h9$ also result in similar images as they make an average of the surroundings for each pixel. Both filters generate a blur effect on the image, but $h9$ doesn't reduce contrast for it performs a weighted average while $h4$ does not, as shown in Figure 9.

h5 & h6: Kernels $h5$ and $h6$ are quite similar, being different only by the diagonal being highlighted. These kernels

Figure 6: Effect of kernel h1 and h2 (right) combined on city.png and house.png (left) by combining the kernels beforehand

Figure 7: Effect of kernel h1 and h2 (right) combined on city.png and house.png (left) following the Sobel filter formulation, in which the result of each convolution is combined instead of combining their underlying filters.

gray out most of the picture with the exception of clear edges that have sudden change in luminance level. In Figure 10, their effects are shown to be almost identical with the exception of the order of the low tone/high tone position on the border.

h7: Kernel $h7$ gray out all regions that have low spatial contrast, for example, a blurry background. However, edges and shadows become prominent as they mark sudden changes

Figure 8: Effect of kernel h3 (middle) and h8 (right) combined on butterfly.png (left) with the matplotlib gray colormap

Figure 9: Effect of kernel h4 (middle) and h9 (right) combined on butterfly.png (left) with the matplotlib gray colormap

Figure 10: Effect of kernel h5 (middle) and h6 (right) combined on seagull.png (left) with the matplotlib gray colormap

that are highlighted by the kernel, as seen in Figure 11

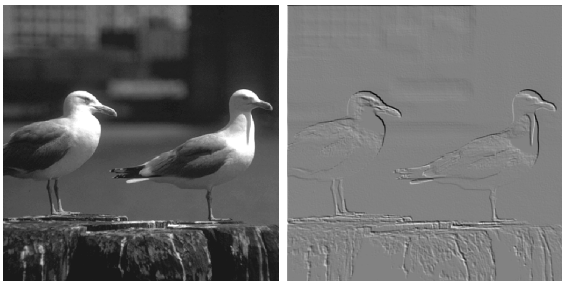

Figure 11: Effect of kernel h7 (right) combined on seagull.png (left) with the matplotlib gray colormap

V. Conclusion

In this project, we saw 2 different kinds of linear transformations applicable on RGB images: the first being the Sepia matrix transformation, that highlight warmer tones in a image, and the second was the RGB luminance equation, that turned all images monochromatic.

Furthermore, we applied 10 different kernel convolutions on monochromatic images and got edge detectors, contrast reducers and blur in different levels for the images tried.

Finally, all transformations tried were developed with a vectorized approach, thus leading to very fast results on all images tested.

REFERENCES

[1] NumPy, "Numpy indexing." [Online]. Available: "https://numpy.org/doc/stable/reference/arrays.indexing.html"

[2] W. Fulton, "A few scanning tips," 2010. [Online]. Available: "https://www.scantips.com/lumin.html"

[3] A. W. Robert Fisher, Simon Perkins and E. Wolfart, "Sobel filter formulation," 2010. [Online]. Available: "https://homepages.inf.ed.ac.uk/rbf/HIPR2/sobel.htm"