

# **Trabalho Prático I:**

## **Processamento digital de imagens**

Jhonata Ezequiel Alves de Miranda, Mateus Batista Honório, Victoria Monteiro Pontes



CENTRO DE INFORMÁTICA  
UNIVERSIDADE FEDERAL DA PARAÍBA

João Pessoa, 2023



Jhonata Ezequiel Alves de Miranda, Mateus Batista Honório, Victoria Monteiro Pontes

## Trabalho Prático I

Projeto apresentado ao curso Engenharia de Computação  
do Centro de Informática, da Universidade Federal da Paraíba,  
como requisito para a obtenção de nota em Processamento Digital de Imagens

Professor: Leonardo Vidal

Abril de 2023



## LISTA DE FIGURAS

1	Imagen ao longo de seus componentes Y, I e Q . . . . .	12
2	Esquema de cores que auxilia na visualização da conversão para negativo . . . . .	13
3	Como funciona a aplicação de uma máscara . . . . .	14
4	Imagen original . . . . .	16
5	Imagen com filtro Sobel . . . . .	16
6	Imagen original . . . . .	17
7	Imagen original . . . . .	17
8	. . . . .	18
9	Função que faz a conversão de RGB para YIQ . . . . .	19
10	Função auxiliar na transformação de YIQ para RGB . . . . .	19
11	Função que transforma de forma efetiva a imagem YIQ para RGB . . . . .	20
12	Função que retorna cada pixel em sua forma negativa . . . . .	20
13	Função que transforma imagem original em sua versão em negativo chamando a função da figura 11 . . . . .	21
14	Função que transforma de imagem YIQ para negativo Y . . . . .	21
15	Função que aplica filtro de Correlação . . . . .	22
16	Função que checa se o tamanho do filtro é válido . . . . .	22
17	Função que lê filtros de um arquivo . . . . .	23
18	Função de Expansão de Histograma . . . . .	24
19	Função do filtro da Mediana . . . . .	24
20	. . . . .	25
21	Lena com Conversão RGB-YIQ-RGB, a imagem volta ao que era originalmente . . . . .	26
22	Lena em negativo nos seus canais RGB . . . . .	27
23	Lena com negativo na componente Y da imagem YIQ . . . . .	27
24	Lena com Filtro Box_11x1(Box_1x1(image)) . . . . .	28
25	Lena com filtro Box 11x11 . . . . .	29
26	Lena com detecção de bordas verticais . . . . .	30

27	Lena com detecção de bordas horizontais . . . . .	30
28	Lena com filtro Emboss . . . . .	31
29	Lena com filtro Mediana 3x3 . . . . .	32
30	Lena com filtro Mediana 7x7 . . . . .	32

# Sumário

<b>1 INTRODUÇÃO</b>	<b>9</b>
1.1 Descrição do Projeto . . . . .	9
1.2 Ferramentas e bibliotecas utilizadas . . . . .	9
1.2.1 Objetivo geral . . . . .	9
<b>2 Fundamentação Teórica</b>	<b>11</b>
2.1 Sistema RGB em imagens . . . . .	11
2.2 Sistema YIQ . . . . .	11
2.3 Conversão de RGB para YIQ . . . . .	11
2.4 Imagens em negativo . . . . .	12
2.5 Filtros . . . . .	13
2.5.1 A operação de correlação . . . . .	13
2.5.2 Máscaras e a extensão por zeros . . . . .	14
2.5.3 Filtro média ou Box . . . . .	14
2.5.4 Filtro Soma . . . . .	15
2.5.5 Filtro Gradiente de Sobel . . . . .	15
2.5.6 Filtro Emboss . . . . .	17
2.6 Expansão de histograma . . . . .	18
<b>3 METODOLOGIA</b>	<b>19</b>
3.1 Conversão RGB-YIQ-RGB . . . . .	19
3.2 Negativo em: . . . . .	20
3.2.1 RGB . . . . .	20
3.2.2 YIQ . . . . .	21
3.3 Correlação e filtros . . . . .	21
3.3.1 Correlação . . . . .	21
3.3.2 Expansão de histograma . . . . .	24
3.4 Filtro mediana . . . . .	24
3.4.1 Filtro mediana convecional . . . . .	24

3.4.2	Filtro mediana MxN, com M e N ímpares, sobre R, G e B . . . . .	25
<b>4</b>	<b>APRESENTAÇÃO E ANÁLISE DOS RESULTADOS</b>	<b>26</b>
4.1	Conversão RGB-YIQ-RGB . . . . .	26
4.2	Negativo . . . . .	27
4.2.1	Negativo em RGB . . . . .	27
4.2.2	Negativo em YIQ . . . . .	27
4.3	Correlação e filtros . . . . .	28
4.3.1	Filtro box_11x1( box_1x11(image)) . . . . .	28
4.3.2	Filtro box_11x11 . . . . .	29
4.3.3	Sobel vertical . . . . .	30
4.3.4	Sobel horizontal . . . . .	30
4.3.5	Filtro Emboss . . . . .	31
4.4	Mediana . . . . .	31
4.4.1	Filtro 3x3 . . . . .	31
4.4.2	Filtro 7x7 . . . . .	32
<b>5</b>	<b>CONCLUSÕES E TRABALHOS FUTUROS</b>	<b>33</b>
<b>REFERÊNCIAS</b>		<b>33</b>

# 1 INTRODUÇÃO

O processamento digital de imagens é uma área essencial para diferentes contextos de utilização. Ou seja, o PDI (Processamento Digital de Imagens) é base e pré-requisito para áreas como a visão computacional e a Inteligência Artificial como fase preparatória de imagens de entrada para a extração de informações nelas embutidas.

## 1.1 Descrição do Projeto

O projeto consiste em implementar 4 exigências principais:

1. Conversão RGB-YIQ-RGB com atenção especial aos valores RGB
2. Negativo. Duas formas de aplicação devem ser testadas: em RGB (banda a banda) e na banda Y, com posterior conversão para RGB
3. Correlação m x n (inteiros não negativos), com extensão por zeros, sobre R, G e B, com offset (inteiro) e filtro definidos em um arquivo (txt) à parte. Testar com filtros Soma, Box, —Sobel— e Emboss, e explicar os resultados. Compare Box11x1(Box1x11(Image)) com Box(11x11), em termos de resultado e tempo de processamento. Para o Sobel, aplique expansão de histograma para [0, 255]. Para o filtro de Emboss, aplique valor absoluto ao resultado da correlação, e então some o offset
4. Filtro mediana m x n, com m e n ímpares, sobre R, G e B

## 1.2 Ferramentas e bibliotecas utilizadas

Foram utilizadas como IDE e editores de texto o PyCharm e VSCode, com o auxílio da extensão Live Server que permite a colaboração do grupo em uma mesma instância do código em tempo real e de forma remota. Foram também utilizadas as bibliotecas Numpy e Pillow. A biblioteca Numpy foi utilizada para a implementação das exigências de forma otimizada enquanto a biblioteca Pillow foi utilizada para instanciar a imagem como objeto.

### 1.2.1 Objetivo geral

O objetivo geral deste projeto é explorar diferentes técnicas de filtros e correlação em Processamento Digital de Imagens, utilizando a conversão RGB para YIQ e voltando para RGB. Serão testados os efeitos do filtro negativo de imagem da forma banda-a-banda e na banda Y, além de um filtro M x N com M e N ímpares sobre RGB.

Também serão explorados diferentes tipos de correlação m x n (inteiros não negativos) sobre as bandas R, G e B, com extensão por zeros, utilizando um filtro definido em um arquivo txt à parte. Serão testados os filtros de soma, box, Sobel e emboss, e os resultados serão analisados e comparados.

Para o filtro Box11x1(Box1x11(Image)), será comparado com o filtro Box(11x11) em termos de resultado e tempo de processamento. Para o filtro Sobel, será aplicada a expansão do histograma para [0, 255]. Já para o filtro Emboss, será aplicado o valor absoluto ao resultado da correlação e, em seguida, somado o offset.

O objetivo final é obter um entendimento mais profundo das técnicas de filtros e correlação em Processamento Digital de Imagens e como elas podem ser aplicadas em diferentes contextos, com diferentes filtros e parâmetros.

## 2 Fundamentação Teórica

### 2.1 Sistema RGB em imagens

O sistema RGB é um sistema em que as três cores primárias (Red, Green, Blue) dão origem às outras cores visíveis ao olho humano. Ao utilizar duas cores do sistema RGB e retirando uma terceira podemos criar as cores Cyan(Blue + Green), Yellow(Red + Green), Magenta(Blue + Red), que são chamadas de cores secundárias e formam o sistema CMY (Cyan, Magenta, Yellow), um sistema complementar ao sistema RGB utilizado na composição de pigmentos de objetos.

Em uma imagem digital, composta por pixels, é possível perceber a utilização do sistema RGB, uma vez que cada pixel é representado por 3 canais. Cada canal é responsável por uma cor do sistema RGB.

### 2.2 Sistema YIQ

O sistema YIQ é um sistema utilizado na transmissão de sinais pela TV em um sistema de cores. Da sua sigla temos a sua composição, ou seja, Y representa a luminância, I representa a matiz e Q é a componente de saturação.

Observação: A matiz é o comprimento de onda dominante de cor. Usada para dar um nome a uma cor. Já a saturação mede a pureza relativa da cor ou quantidade de luz branca misturada com uma matiz.

### 2.3 Conversão de RGB para YIQ

A conversão de uma imagem RGB para YIQ se dá pela fórmula:

$$Y = 0.299R + 0.587G + 0.114B \quad (1)$$

$$I = 0.596R - 0.274G - 0.322B \quad (2)$$

$$Q = 0.211R - 0.523G + 0.312B \quad (3)$$

Já a conversão de uma imagem YIQ para RGB se dá pela fórmula:

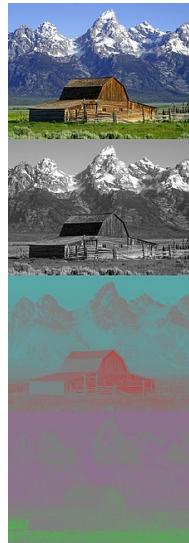
$$R = 1.000Y + 0.956I + 0.621Q \quad (4)$$

$$G = 1.000Y - 0.272I - 0.647Q \quad (5)$$

$$B = 1.000Y - 1.106I + 1.703Q \quad (6)$$

Entretanto, na prática, essas equações podem resultar em valores negativos na conversão RGB-YIQ-RGB exigida no primeiro item da descrição do projeto. Portanto, deve haver um tratamento de valores do pixel.

A seguir temos uma imagem que ilustra como se apresenta um sistema YIQ:



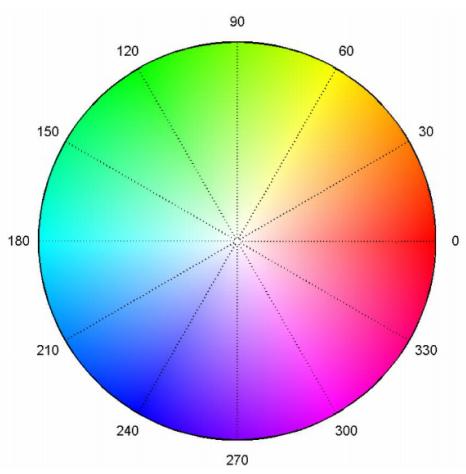
**Figura 1: Imagem ao longo de seus componentes Y, I e Q**

## 2.4 Imagens em negativo

Uma imagem em negativo consiste em uma imagem cujos canais estão com valores complementares aos valores originais. Por exemplo, uma imagem com os canais RGB que tenham valores 255, terão estes valores convertidos para 0, enquanto pixels com valores 0 terão seus valores modificados para 255. Ou seja, para que haja a conversão para negativo, utilizamos o valor original do pixel e subtraímos este valor de 255.

O efeito visual dessa operação é representada na imagem abaixo. Ao termos uma imagem completamente amarela (mistura das cores vermelho e verde, cada uma com valor 255 e azul com valor 0) e a convertermos para negativo, temos que os canais vermelho e verde passam a ter valor zero e o canal azul passa a ter valor 255. A figura 2 mostra que a cor complementar ao amarelo, ou que é sua "oposta" é o azul em seu tom mais forte (valor 255).

Uma imagem com canais YIQ também pode ser convertida para negativo em Y (canal de luminância) utilizando a mesma operação.



**Figura 2:** Esquema de cores que auxilia na visualização da conversão para negativo

## 2.5 Filtros

Filtros são ferramentas importantes para aplicar diferentes efeitos em uma imagem. Por exemplo, filtros de média, mediana, gaussiana são utilizados para suavização de traços em uma imagem, ou seja, torná-la mais uniforme. Já o filtro gradiente de Sobel é utilizado para detecção de bordas. O filtro Emboss é utilizado para dar destaque a bordas de forma a criar uma aparência tridimensional à uma imagem bidimensional, ou seja, cria-se a ilusão de um alto-relevo ou baixo-relevo.

### 2.5.1 A operação de correlação

A correlação em processamento digital de imagens é uma operação matemática que é amplamente utilizada para medir a similaridade entre duas imagens ou entre uma imagem e um filtro, também conhecido como kernel. É uma técnica comum em várias aplicações de processamento de imagens, como reconhecimento de padrões, detecção de bordas, filtragem de imagens e rastreamento de objetos.

A correlação é geralmente aplicada comparando-se duas imagens pixel a pixel. Para cada pixel na imagem de origem, uma janela de convolução é aplicada à imagem de destino, e uma medida de similaridade é calculada multiplicando-se os valores dos pixels correspondentes nas duas imagens e somando os resultados. O resultado é um valor numérico que indica o grau de correspondência entre as duas imagens ou entre a imagem e o filtro.

A correlação é usada em várias aplicações de processamento de imagens. Por exemplo, em detecção de bordas, um filtro de borda é aplicado a uma imagem usando a correlação para realçar áreas de transição de intensidade, como bordas de objetos. Em

reconhecimento de padrões, a correlação pode ser usada para comparar uma imagem de consulta com imagens de referência a fim de encontrar uma correspondência adequada.

É importante notar que a correlação é sensível a deslocamentos e variações de escala na imagem, uma vez que compara os valores dos pixels diretamente. Por essa razão, técnicas como a correlação cruzada normalizada, que normaliza as imagens antes da correlação, são frequentemente utilizadas para lidar com essas variações e obter resultados mais robustos

### 2.5.2 Máscaras e a extensão por zeros

Uma máscara é uma matriz bidimensional de valores numéricos que é usada como um filtro ou uma janela de convolução para processar uma imagem. Ao aplicar a máscara em uma imagem, a operação é feita pixel a pixel, em que cada valor numérico é multiplicado pelo valor correspondente do pixel na imagem, e então os resultados são somados para obter o valor final do pixel da imagem processada.

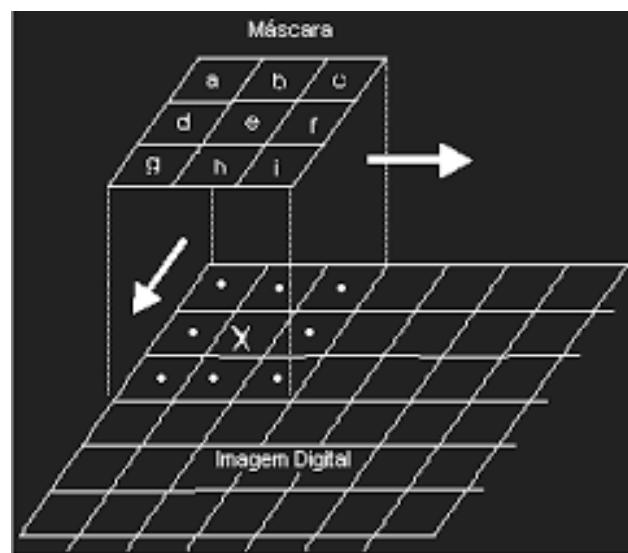


Figura 3: Como funciona a aplicação de uma máscara

A extensão por zeros é uma técnica usada para garantir que a máscara seja aplicada em todos os pixels da imagem sem deixar os pixels da borda de fora. Esta técnica envolve a adição de zeros aos limites de uma imagem original para simular pixels fictícios com valor de intensidade zero para que a máscara possa ser aplicada.

### 2.5.3 Filtro média ou Box

Um filtro média é uma técnica de processamento digital de imagens que é usada para suavizar uma imagem, reduzindo o ruído e as variações de intensidade dos pixels.

Ele é um tipo de filtro linear que substitui o valor de um pixel pelo valor médio dos pixels em sua vizinhança.

O filtro média funciona deslizando uma janela retangular ou quadrada, também conhecida como janela de convolução, sobre a imagem de origem. Para cada posição da janela, o filtro calcula a média dos valores dos pixels dentro da janela e atribui esse valor ao pixel correspondente na imagem de destino.

O tamanho da janela de convolução afeta a quantidade de suavização aplicada à imagem. Janelas maiores têm um efeito de suavização mais intenso, pois incorporam mais pixels na média, resultando em uma redução maior do ruído e das variações de intensidade. No entanto, janelas maiores também podem causar uma perda de detalhes finos na imagem, resultando em uma imagem suavizada demais.

O filtro média é amplamente utilizado em várias aplicações de processamento de imagens, como redução de ruído, pré-processamento de imagens para algoritmos de visão computacional e suavização de imagens para melhorar a qualidade visual. É uma técnica simples e computacionalmente eficiente, mas pode não ser adequada para todos os tipos de imagens ou para situações em que a preservação de detalhes finos é importante. Existem também variantes do filtro média, como o filtro média ponderado, que atribui pesos diferentes aos pixels dentro da janela de convolução com base em sua proximidade ao pixel central, oferecendo mais flexibilidade na aplicação do efeito de suavização.

#### 2.5.4 Filtro Soma

O filtro Soma funciona de forma simples. A máscara do filtro é aplicada à imagem original deslizando por ela pixel a pixel. Para cada pixel na imagem, a máscara é multiplicada pelo valor do pixel correspondente na imagem, e os resultados são somados para gerar o valor resultante do pixel na imagem filtrada. Esse valor resultante é então atribuído ao pixel correspondente na imagem filtrada.

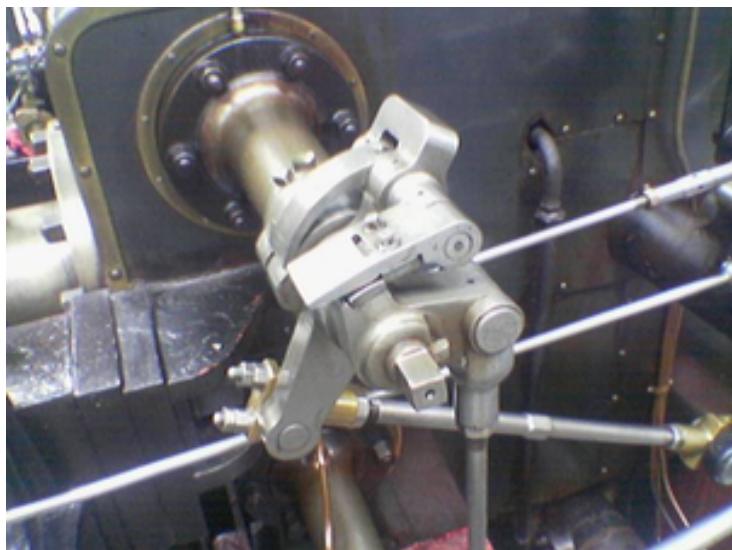
A máscara do filtro Soma possui valores iguais em suas entradas, que normalmente é 1, divididos pelo número total de elementos da máscara. A aplicação do filtro Soma envolve a soma ponderada dos valores dos pixels na vizinhança do pixel analisado. Esse tipo de filtro causa um efeito de suavização.

#### 2.5.5 Filtro Gradiente de Sobel

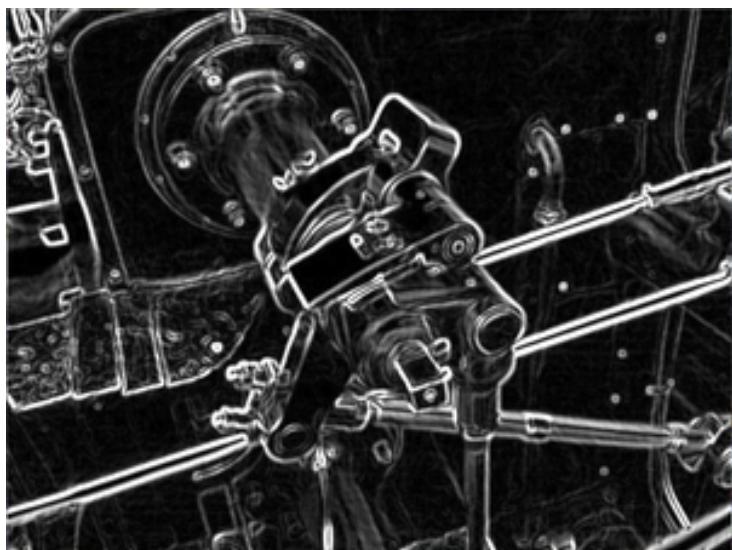
O filtro de Sobel é uma operação de detecção de bordas horizontais e verticais e por isso possui duas máscaras. Cada máscara é uma matriz quadrada de tamanho 3x3 ou 5x5, com pesos específicos atribuídos a cada elemento da matriz. Esses pesos

são projetados para calcular a derivada de primeira ordem da intensidade dos pixels em relação às coordenadas horizontal (x) e vertical (y) da imagem.

A convolução é realizada multiplicando os valores dos pixels vizinhos pela correspondente máscara Sobel e somando os resultados para obter o valor resultante do pixel na imagem filtrada. Isso resulta em duas imagens filtradas, uma para detecção de bordas horizontais e outra para detecção de bordas verticais. As imagens filtradas resultantes do filtro Sobel podem ser combinadas usando técnicas de combinação de imagens, como a magnitude do gradiente ou o ângulo do gradiente, para produzir uma imagem final que destaque as bordas na imagem original. Podemos conferir o resultado da aplicação do filtro nas figuras abaixo:



**Figura 4: Imagem original**



**Figura 5: Imagem com filtro Sobel**

### 2.5.6 Filtro Emboss

O filtro Emboss é um filtro utilizado para realçar o efeito de relevo em uma imagem, criando uma aparência tridimensional. Ele é comumente utilizado para fins de visualização ou estilização de imagens, adicionando um efeito de profundidade. Para cada pixel na imagem, a máscara é multiplicada pelo valor do pixel correspondente na imagem original, e os resultados são somados para gerar o valor resultante do pixel na imagem filtrada. Esse valor resultante é então atribuído ao pixel correspondente na imagem filtrada. Assim fica uma imagem com filtro Emboss:

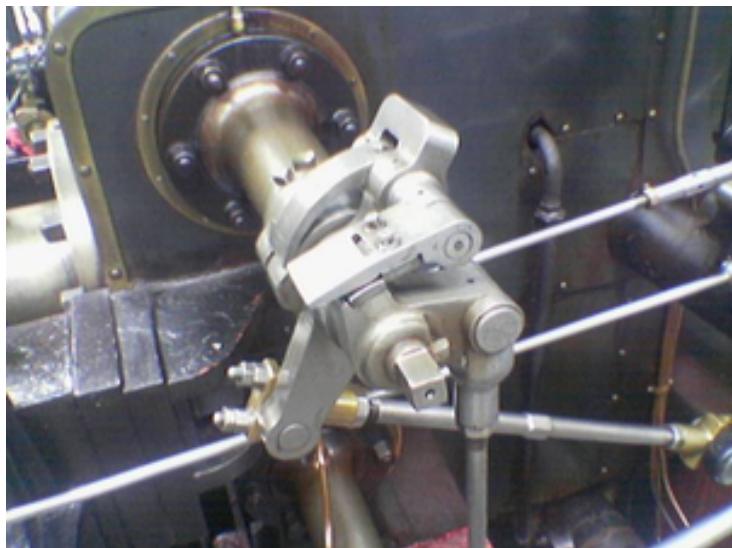


Figura 6: Imagem original



Figura 7: Imagem original

## 2.6 Expansão de histograma

A expansão de histograma é uma técnica utilizada para aumentar o contraste de uma imagem. O histograma de uma imagem é uma representação gráfica da distribuição de intensidades de pixels na imagem. A expansão de histograma ajusta a faixa de intensidades de pixels em uma imagem para abranger todo o intervalo disponível, a fim de ampliar o contraste.

O processo de expansão de histograma envolve a redistribuição dos valores de intensidade de pixel na imagem original para um intervalo maior, normalmente de 0 a 255, que é o intervalo de intensidade típico de uma imagem de 8 bits. Isso é feito por meio de uma transformação de intensidade, em que os valores de intensidade de pixel são mapeados de sua faixa original para a nova faixa de intensidades desejada.

A expansão de histograma é aplicada pixel a pixel, usando uma função de mapeamento que determina como os valores de intensidade de pixel originais são mapeados para a nova faixa de intensidades. Essa função de mapeamento pode ser determinada de várias maneiras, como usando uma função linear, uma função logarítmica, uma função exponencial ou uma função não linear personalizada.

Para a aplicação desta técnica, podemos utilizar esta fórmula:

$$s = T(r) = \text{round} \left( \frac{r - r_{\min}}{r_{\max} - r_{\min}} (L - 1) \right)$$

**Figura 8**

### 3 METODOLOGIA

#### 3.1 Conversão RGB-YIQ-RGB

Para implementar a solução, o time desenvolveu três funções relativas às conversões necessárias:

```
def rgb_to_yiq(image: Image) -> List[List[List[float]]]:
    """
    Receives an RGB image and makes the necessary adjustments to convert each pixel to its YIQ counter-part utilizing the
    correct formula for it.
    :param image: Original image in RGB
    :return: List containing every component of every pixel in YIQ format
    """
    y_pixels = [[_ for _ in range(image.size[1] - 1)]
                for _ in range(image.size[0] - 1)]
    i_pixels = [[_ for _ in range(image.size[1] - 1)]
                for _ in range(image.size[0] - 1)]
    q_pixels = [[_ for _ in range(image.size[1] - 1)]
                for _ in range(image.size[0] - 1)]
    for r in range(image.size[0] - 1):
        for c in range(image.size[1] - 1):
            colors = image.getpixel((r, c))
            y = 0.299 * colors[0] + 0.587 * colors[1] + 0.114 * colors[2]
            i = 0.596 * colors[0] - 0.274 * colors[1] - 0.322 * colors[2]
            q = 0.211 * colors[0] - 0.523 * colors[1] + 0.312 * colors[2]
            y_pixels[r][c] = y
            i_pixels[r][c] = i
            q_pixels[r][c] = q
    return [y_pixels, i_pixels, q_pixels]
```

Figura 9: Função que faz a conversão de RGB para YIQ

Na figura 8, podemos observar que a função recebe como parâmetro um objeto Image (da biblioteca PIL) e retorna uma lista contendo cada componente Y, I e Q em uma lista de uma lista de listas, contendo valores do tipo Float. Seguindo o raciocínio da subseção 2.3 da seção 2 - Fundamentação Teórica, temos então a utilização das equações descritas para a conversão de RGB para YIQ.

```
def get_rgb_pixels_from_yiq(pixels: List[List[List[float]]) -> ndarray:
    """
    Similar to the RGB-YIQ function, but this time it does the opposite. It receives a list containing every pixel in
    YIQ format, and returns them in RGB format after conversion.
    :param pixels: List containing the pixels in YIQ format
    :return: np array containing every pixel in RGB format
    """
    new_pixels = np.empty([
        len(pixels[0]), len(pixels[0][0]), 3], dtype=np.uint8)
    for i in range(len(pixels[0])):
        for c in range(len(pixels[0][0])):
            r = pixels[0][i][c] + 0.956 * \
                pixels[1][i][c] + 0.621 * pixels[2][i][c]
            g = pixels[0][i][c] - 0.272 * \
                pixels[1][i][c] - 0.547 * pixels[2][i][c]
            b = pixels[0][i][c] - 1.166 * \
                pixels[1][i][c] + 1.705 * pixels[2][i][c]
            r = round(r)
            g = round(g)
            b = round(b)
            r = r if r > 0 else 0
            g = g if g > 0 else 0
            b = b if b > 0 else 0
            r = r if r < 256 else 255
            g = g if g < 256 else 255
            b = b if b < 256 else 255
            new_pixels[i][c] = np.array([r, g, b], dtype=np.uint8)
    return new_pixels
```

Figura 10: Função auxiliar na transformação de YIQ para RGB

Na figura 9, temos a implementação de uma função auxiliar que utiliza as equações de transformação de YIQ para RGB. Ao iterar por todos os pixels, o código demonstrado pode fazer a transformação de YIQ para RGB de forma rápida e simples. Já nas linhas 87 a 98, há o tratamento de valores inesperados como valores maiores que 255 ou zero para valores negativos. Essa função retorna uma lista de lista com os novos pixels já modificados.

```

100 def yiq_to_rgb(pixels: List[List[List[float]]]) -> Image:
101     """
102         Transforms an image (in this case, it is a list of values that characterize a pixel) in YIQ into RGB calling an
103         auxiliary function to make the calculations. It rotates the pixels as well to adjust the image correctly after the
104         conversion.
105         :param pixels: YIQ pixels list
106         :return: RGB image from the YIQ pixels
107     """
108     pixels = get_rgb_pixels_from_yiq(pixels)
109     rotated_pixels = np.rot90(pixels, axes=(1, 0))
110     rotated_pixels = np.flip(rotated_pixels, axis=1)
111     new_image = Image.fromarray(rotated_pixels)
112     return new_image
113

```

**Figura 11:** Função que transforma de forma efetiva a imagem YIQ para RGB

A figura 10 ilustra a função que recebe a lista de uma lista de uma lista do tipo Float como parâmetro e retorna um objeto do tipo Image. Na linha 108, é chamada a função da figura 9 para obter as listas com os canais RGB da imagem. Até este ponto, a conversão de RGB-YIQ-RGB está quase completa com apenas uma exceção: a imagem aparece rotacionada para a esquerda ou mesmo de cabeça para baixo. As linhas 108 e 109 do código são responsáveis por rotacionar a imagem à sua posição original.

### 3.2 Negativo em:

#### 3.2.1 RGB

```

7 def get_negative_pixels(image: Image) -> List:
8     """
9         This function returns every pixel in its negative counter-part.
10        :param image: Original Image in RGB
11        :return: List with the negative RGB pixels
12    """
13    red_pixels = [[] for _ in range(image.size[0] - 1)]
14    green_pixels = [[] for _ in range(image.size[0] - 1)]
15    blue_pixels = [[] for _ in range(image.size[0] - 1)]
16    for r in range(image.size[0] - 1):
17        for c in range(image.size[1] - 1):
18            colors = image.getpixel((r, c))
19            red_pixels[r].append(255 - colors[0])
20            green_pixels[r].append(255 - colors[1])
21            blue_pixels[r].append(255 - colors[2])
22
23    return [red_pixels, green_pixels, blue_pixels]
24

```

**Figura 12:** Função que retorna cada pixel em sua forma negativa

Nesta função podemos observar que é feita uma operação pixel a pixel nos três canais para que sejam retornadas as suas contra-partes negativas

```

26     def turn_negative(image: Image) -> Image:
27         """
28             Transforms the image in its negative RGB counter-part
29             :param image: Original image in RGB
30             :return: New negative RGB image
31         """
32         pixels = get_negative_pixels(image)
33         for r in range(image.size[0] - 1):
34             for c in range(image.size[1] - 1):
35                 image.putpixel(
36                     (r, c), (pixels[0][r][c], pixels[1][r][c], pixels[2][r][c]))
37         return image
38
39

```

**Figura 13:** Função que transforma imagem original em sua versão em negativo chamando a função da figura 11

Esta função chama a função da Figura 11 e recebe os pixels já negativos, a partir daí, ocorre a junção dos canais e a imagem é retornada

### 3.2.2 YIQ

```

115     def negative_on_y(image: Image) -> Image:
116         """
117             Turns the Y from the YIQ color scheme into negative and returns an image of the result.
118             :param image: Original image in RGB format
119             :return: RGB image with the Y coordinate of YIQ negative.
120         """
121         yiq_pixels = rgb_to_yiq(image)
122         for i in range(len(yiq_pixels[0])):
123             for r in range(len(yiq_pixels[0][i])):
124                 yiq_pixels[0][i][r] = 255 - yiq_pixels[0][i][r]
125         return yiq_to_rgb(yiq_pixels)
126
127

```

**Figura 14:** Função que transforma de imagem YIQ para negativo Y

Nesta função, é feita a contra-parté negativa da componente Y seguindo de forma análoga o procedimento descrito na subseção 2.3 da seção 2 - Fundamentação Teórica. Uma observação interessante a ser feita é que, tradicionalmente a composição do Y varia apenas de 0 a 1. Porém, a biblioteca da PILL fornece a componente Y como se variasse de 0 a 255, como se fosse um canal como o do RGB.

## 3.3 Correlação e filtros

### 3.3.1 Correlação

Esse filtro é utilizado com a extensão por zeros, como indicado na linha 215 e efetivado na linha 224. O loop for irá recuperar todos os valores de R, G e B na linha 222. A operação começa a partir da linha 227

```

201 def correlational(image: Image, size: Tuple[int, int], correlational_filter: ndarray, offset: Tuple[int, int]) -> Image:
202     """
203         This function will apply a correlational filter
204         :param image: image with the filter will be applied
205         :param size: size of the window of the filter (m,n)
206         :param correlational_filter: array containing the filter and its values
207         :param offset: offset to be considered by the function
208         :return:
209         """
210
211     im = np.array(image)
212     m, n = size
213     window = correlational_filter
214
215     # how many rows and columns will be expanded by 0
216     mxn_extended = (m//2, n//2)
217
218     # final image will be the 'result'
219     result = np.zeros_like(im)
220
221     for i in range(im.shape[2]):
222         # getting all the values of R, then G, then B
223         channel = im[:, :, i]
224
225         # expanding the array with 0s according to the needs of the filter
226         padded_channel = np.pad(channel, mxn_extended, mode='constant')
227         for x in range(im.shape[0]):
228             for y in range(im.shape[1]):
229                 # getting a sub_image that starts at position x and finishes at position x+m
230                 # and starts at position y and finishes at position y+n
231                 sub_image = padded_channel[x: x + m, y: y + n]
232
233                 # if the sub_image has not the same size of the window, it means that the filter is already applied
234                 # to the image
235                 if sub_image.shape == window.shape:
236                     # the new value will be the absolute value because some filters can return negative numbers
237                     new_value = abs(np.sum(sub_image * window))
238
239                     """
240                         if the image is ixi, and x+offset[0] is larger than i or y+offset[1] is larger than j, it means
241                         that the filter is already applied to all the image, because the offset will be after the end of the
242                         image
243
244                         if x + offset[0] <= im.shape[0] - 1 and y + offset[1] <= im.shape[1] - 1:
245                             result[x + offset[0], y + offset[1], i] = new_value
246
247             result = np.uint8(result)
248     return Image.fromarray(result)

```

**Figura 15:** Função que aplica filtro de Correlação

```

184 def call_correlation_mxn(image: Image, correlational_filter: ndarray, offset: Tuple[int, int]) -> Image:
185     """
186         This function checks if the size of the filter is valid and then calls the correlational filter
187         :param image: image with the filter will be applied
188         :param correlational_filter: array containing the filter and its values
189         :param offset: offset to be considered by the function
190         :return: image with the filter applied
191         """
192
193     size = correlational_filter.shape[0], correlational_filter.shape[1]
194
195     m, n = size
196     if m < 0 or n < 0 or type(m) != int or type(n) != int:
197         raise ValueError("m and n must be a positive integer!")
198     return correlational(image, size, correlational_filter, offset)
199

```

**Figura 16:** Função que checa se o tamanho do filtro é válido

Essa função trabalha em conjunto com a função `read_correlational_filters()`. Ela é responsável por checar se a matriz lida do arquivo é de fato válida para a aplicação do filtro. O core de sua funcionalidade se encontra na linha 196 em diante, que checa se os valores de `m` e `n` de uma matriz são positivas e se o tamanho da matriz é inteiro.

```

245     def read_correlational_filters(file_name: str) -> None:
246         with open(file_name) as f:
247             lines = f.readlines()
248
249             correlational_filters = [line.strip() for line in lines]
250
251             offsets = [offset for offset in correlational_filters if ',' in offset]
252             offsets = [(int(offset.split(',')[0]), int(offset.split(',')[1]))]
253
254             filters = [[] for _ in correlational_filters if ',' in _]
255
256             j = 0
257             for i in range(len(correlational_filters)):
258                 if correlational_filters[i] != '':
259                     filters[j].append(correlational_filters[i])
260                 if correlational_filters[i] == '':
261                     j += 1
262
263             finished_arrays = [np.array(_) for _ in range(len(filters))]
264             for j in range(len(filters)):
265                 filters[j].pop(0)
266
267                 for i in range(len(filters[j])):
268                     filters[j][i] = filters[j][i].split(' ')
269
270                     for h in range(len(filters[j][i])):
271                         treated_float = filters[j][i][h] if '/' in filters[j][i][h] else None
272
273                         if treated_float:
274
275                             split_float = filters[j][i][h].split('/')
276                             split_float = int(split_float[0]) / int(split_float[1])
277                             filters[j][i][h] = split_float
278
279                         filters[j][i] = [float(_) for _ in filters[j][i]]
280
281             finished_arrays[j] = np.array(filters[j])
282
283             im = Image.open("tests/image.png")
284             if file_name == "tests/box_1x1(box_1x1(image)).txt":
285                 begin_time = timeit.default_timer()
286                 for i in range(len(finished_arrays)):
287                     im = call_correlation_mxn(im, finished_arrays[i], offsets[i])
288
289                 end_time = timeit.default_timer()
290                 print("Time of box_1x1(box_1x1(image)): ", end_time - begin_time)
291
292             elif file_name == "tests/box_11x11(box_1x11(image)).txt":
293                 begin_time = timeit.default_timer()
294                 im = call.correlation_mx11(im, finished_arrays[0], offsets[0])
295                 end_time = timeit.default_timer()
296                 print("Time of box_11x11 ", end_time - begin_time)
297
298             else:
299                 for i in range(len(finished_arrays)):
300                     im = call.correlation_mx11(im, finished_arrays[i], offsets[i])
301
302             if file_name in ["tests/sobel_horizontal.txt", "tests/sobel_vertical.txt"]:
303                 im = histogram_expansion(im)
304
305             im.show()

```

**Figura 17:** Função que lê filtros de um arquivo

Esta função é responsável por ler de uma arquivo as informações necessárias para extrair um filtro e aplicá-lo a uma certa imagem. Caso o filtro lido seja um filtro de Sobel vertical/horizontal, a função expansão de histograma é chamada.

### 3.3.2 Expansão de histograma

```
248 def histogram_expansion(image: Image) -> Image:  
249     """  
250         the histogram expansion will be from [0, 255]. It will apply the formula to every pixel in the image.  
251         :param image: image that will receive the expansion  
252         :return: output image with the expansion applied  
253     """  
254     im = np.array(image)  
255     min_intensity = np.min(image)  
256     max_intensity = np.max(image)  
257     im_expanded = (im - min_intensity) * \  
258     | (255 / (max_intensity - min_intensity))  
259     output = Image.fromarray(np.uint8(im_expanded))  
260     return output  
261
```

Figura 18: Função de Expansão de Histograma

A função de histograma é aumentar o contraste de uma imagem e utilizamos no projeto para aumentar o contraste da imagem após a aplicação do filtro de Sobel. Portanto, seguimos uma pequena fórmula de expansão citada na subseção 2.6 da seção 2 - Fundamentação Teórica.

## 3.4 Filtro mediana

### 3.4.1 Filtro mediana convecional

```
544 def median_filter(image: Image, size: Tuple[int, int]) -> Image:  
545     """  
546         Applies the median filter to an image.  
547         :param image: original image in RGB format  
548         :param size: size of the median filter (ixj or mxn)  
549         :return: new image with the filter applied.  
550     """  
551     im = np.array(image)  
552     m, n = size  
553  
554     # window will be our kernel or 'mask', containing all values equals to 'one' and being m x n  
555     # because it has only 'ones', it can serve as a mask for the function np.median, used later when applying the filter  
556     window = np.ones((m, n))  
557  
558     # the result variable will be initialized with zeros in the size of the original image  
559     result = np.zeros_like(im)  
560  
561     # this tuple will contain how many rows and columns must be extended to apply the filter  
562     mxn_extended = (m // 2, n // 2)  
563     for i in range(im.shape[2]):  
564         # channel will have all the values of RGB in the image  
565         channel = im[:, :, i]  
566         # padded_im will be the extended image with the necessary number of zeros. If it is not given a value to the  
567         # param 'constant_values', it will be zero, so the image will be extended by zeros  
568         padded_channel = np.pad(channel, mxn_extended, mode='constant')  
569         for x in range(im.shape[0]):  
570             for y in range(im.shape[1]):  
571                 """  
572                     the sub_image will be the used to determine which part of the image will be the one used to  
573                     calculate the median with the np.median function. The sub_image will be multiplied by the 'mask', in  
574                     this case, 'window', and passed as a parameter.  
575                 """  
576                 sub_image = padded_channel[x: x + m, y: y + n]  
577                 if sub_image.shape == window.shape:  
578                     result[x, y, i] = np.median(sub_image * window)  
579  
580             # the final array will be converted to uint8 in order to use less memory  
581             result = np.uint8(result)  
582  
583     return Image.fromarray(result)
```

Figura 19: Função do filtro da Mediana

O filtro mediana vai fazer com que os valores de RGB da imagem sejam atualizados com a mediana calculada pela função `np.median()`. A quantidade de valores considerados

para o cálculo será igual ao tamanho da janela mxn. Por exemplo, se a janela tiver um tamanho 3x3, 9 valores serão levados em consideração.

### 3.4.2 Filtro mediana MxN, com M e N ímpares, sobre R, G e B

```
128 def median_ixj(i: int, j: int, image: Image) -> Image:  
129     """  
130     Verifies if i and j are odd, and calls the function to apply the median filter. If i or j are odd, the function will  
131     raise an error.  
132     :param i: quantity of lines in the image  
133     :param j: quantity of columns in the image  
134     :param image: Original image in RGB format  
135     :return: new image with the filter applied.  
136     """  
137     if i % 2 == 0 or j % 2 == 0:  
138         raise ValueError(" 'i' and 'j' must be odd")  
139     median_image = median_filter(image, (i, j))  
140  
141     return median_image  
142
```

Figura 20

Função que chama a função principal da mediana. ”median\_ixj” verificará se os valores de i e de j são ímpares.

## 4 APRESENTAÇÃO E ANÁLISE DOS RESULTADOS

### 4.1 Conversão RGB-YIQ-RGB



**Figura 21:** Lena com Conversão RGB-YIQ-RGB, a imagem volta ao que era originalmente

Neste resultado, podemos apenas checar se a imagem está correta ao printarmos cada componente YIQ de formas separadas. E aí então, viríamos o resultado de forma mais clara, entretanto, a operação também pode ser checada quando a imagem original é igual ao resultado final da conversão RGB-YIQ-RGB, como podemos ver na imagem resultante da operação.

## 4.2 Negativo

### 4.2.1 Negativo em RGB



**Figura 22:** Lena em negativo nos seus canais RGB

Utilizando a subseção 2.4 - Imagens em negativo, podemos te

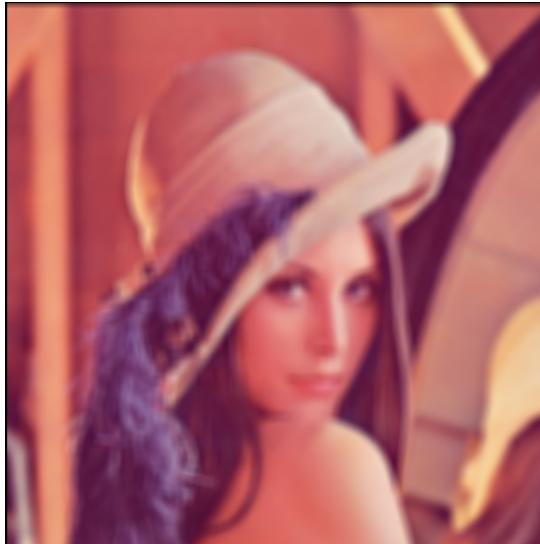
### 4.2.2 Negativo em YIQ



**Figura 23:** Lena com negativo na componente Y da imagem YIQ

## 4.3 Correlação e filtros

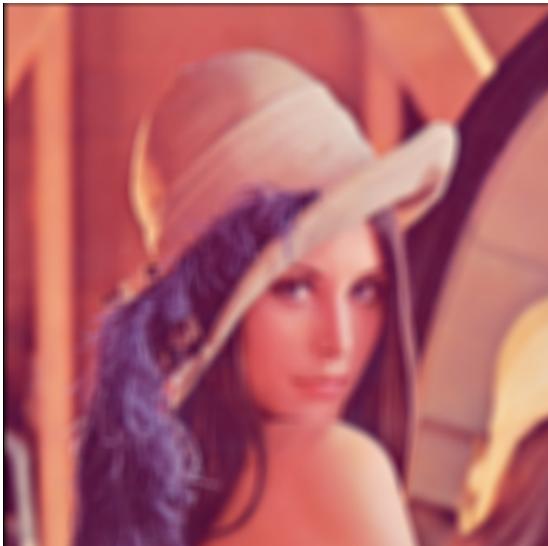
### 4.3.1 Filtro box\_11x1( box\_1x11(image))



**Figura 24:** Lena com Filtro Box\_11x1(Box\_1x11(image))

O filtro box requer muito poder computacional para funcionar, assim como outros filtros correlacionais. Uma técnica utilizada para tentar reduzir a quantidade de cálculos feitos e, consequentemente, reduzir o poder computacional necessário, é dividir o filtro em dois. Primeiro aplicando um filtro 1xn, e depois aplicando um filtro mx1 ao resultado da imagem.

#### 4.3.2 Filtro box\_11x11



**Figura 25: Lena com filtro Box 11x11**

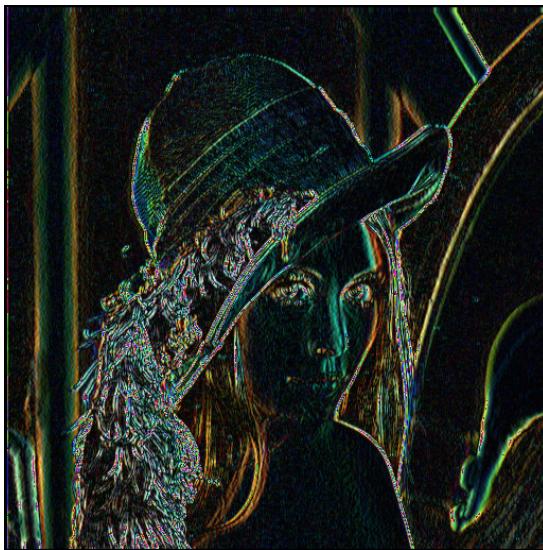
O filtro box 11x11 obteve resultados idênticos ao box 11x1(box 1x11). No entanto, o tempo de processamento dos dois foi diferente de uma maneira inesperada. Como o código utiliza da biblioteca *numpy*, os cálculos feitos são bem rápidos, levando uma média de 5 segundos para aplicar os filtros correlacionais. Quando aplicado o filtro 11x11, apenas uma vez a função é chamada, fazendo com que esse custo médio de 5 segundos de tempo acontecesse apenas uma vez. Já quando a função que aplica o filtro foi chamada duas vezes (uma para o filtro 1x11, depois outra para o filtro 11x1) esse tempo foi aumentado, gerando um tempo médio de 9.5 segundos para aplicar o filtro.

Os tempos reais obtidos para que os filtros terminassem de ser aplicados foram, medidos com o auxílio da biblioteca *timeit*:

- 9.09 segundos para o box 11x1(box 1x11);
- 4.76 segundos para o box 11x11.

Como os tempos não foram condizentes com o esperado (o box 11x1(box 1x11) deveria ser mais rápido), o código tem algum problema de execução que desfavorece a aplicação recursiva de filtros, fazendo com que uma só operação, ainda que em teoria mais custosa, consuma menos tempo que duas operações menores.

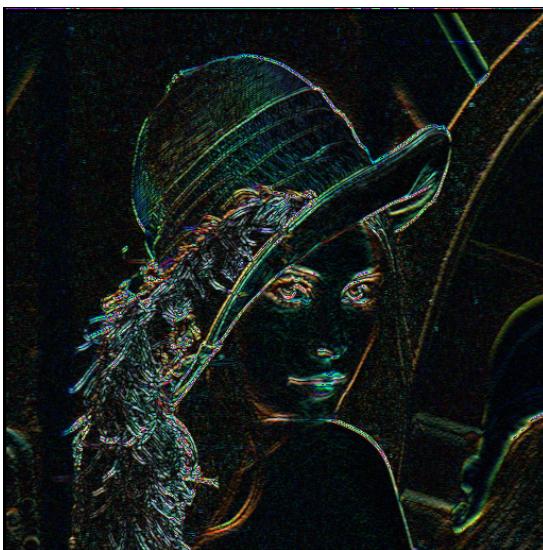
#### 4.3.3 Sobel vertical



**Figura 26:** Lena com detecção de bordas verticais

Na aplicação deste filtro, percebemos que há uma predominância maior de traços verticais. Isso pode ser observado na área da boca de Lena na imagem. A boca por ser formada dde traços horizontais quase não aparece depois da aplicação do filtro. Entretanto, os cabelos de Lena ficam bem desenhados e possuem contraste.

#### 4.3.4 Sobel horizontal



**Figura 27:** Lena com detecção de bordas horizontais

Na aplicação do segundo filtro, dessa vez para bordas horizontais, podemos notar a diferença entre as duas aplicações, pois na figura 28, a área da boca de Lena é bem definida e aparece com mais destaque enquanto seus cabelos quase não aparecem na

imagem. Dessa forma, é possível perceber a diferença entre o destaque que um filtro dá às bordas verticais e o outro dá às horizontais.

#### 4.3.5 Filtro Emboss



**Figura 28:** Lena com filtro Emboss

O filtro Emboss aplicado na immagem de Lena pode ser evidenciado pela nova "textura" dada à imagem. Com atenção, podemos perceber uma certa ilusão de "volume" em uma imagem bidimensional. Dessa forma podemos concluir que o resultado final da aplicação do filtro foi bem sucedida.

### 4.4 Mediana

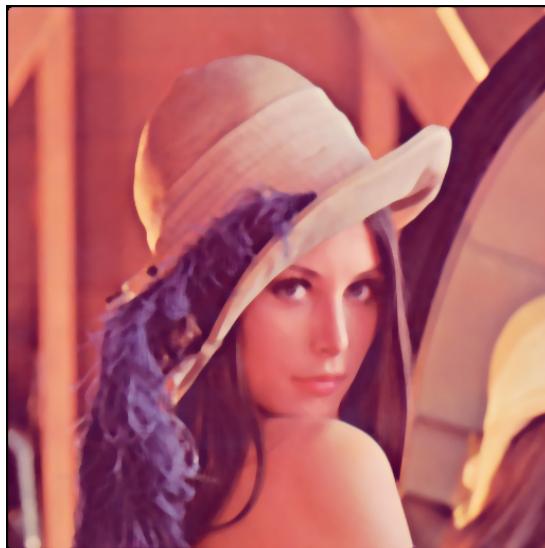
#### 4.4.1 Filtro 3x3

Ao aplicarmos o filtro Mediana 3x3 temos uma variação pouco perceptível nos traços da imaggem. A únnica diferença notória é nos riscos no topo do chapéu de Lena na foto original que, após a aplicação do filtro, desaparecem completamente.



**Figura 29:** Lena com filtro Mediana 3x3

#### 4.4.2 Filtro 7x7



**Figura 30:** Lena com filtro Mediana 7x7

Ao aplicarmos o filtro Mediana 7x7, temos uma evidência clara de suavização dos traços, uma vez que a imagem está um pouco mais "borrada". A máscara maior auxilia nessa mudança, uma vez que o valor do local depende de sua vizinhança. Comparando os dois filtros, se torna evidente qual foi o mais eficaz, o filtro 7x7 é mais eficaz, porém consome mais poder de processamento.

## 5 CONCLUSÕES E TRABALHOS FUTUROS

Ao finalizar este projeto, foi possível explorar várias técnicas de filtros e correlação em Processamento Digital de Imagens, incluindo a conversão RGB para YIQ e voltando para RGB. Os resultados obtidos através da aplicação dessas técnicas foram analisados e comparados, permitindo uma melhor compreensão de como cada filtro e parâmetro afeta a imagem.

O filtro negativo de imagem mostrou-se eficaz em inverter as cores da imagem, tanto na forma banda-a-banda quanto na banda Y. O filtro mediana  $M \times N$  com  $M$  e  $N$  ímpares sobre RGB tornou a imagem mais suave, assim como esperado. Os diferentes filtros de correlação, como soma, box, Sobel e emboss, também foram testados e seus resultados comparados.

Foi interessante observar como o tempo de processamento é afetado pelo tamanho do filtro e pela complexidade da operação de correlação. A comparação entre o filtro Box $11 \times 1$ (Box $1 \times 11$ (Image)) e o filtro Box( $11 \times 11$ ) mostrou como a escolha do filtro pode afetar significativamente o resultado final.

Por fim, a expansão de histograma para a imagem da Lena não apresentou diferença, visto que os valores mínimos e máximos que os pixels assumiram já foram 0 e 255. No entanto, é possível que, a depender da imagem, a expansão melhore a visualização da imagem.

Em suma, este projeto forneceu uma compreensão mais profunda do uso de filtros e correlação em Processamento Digital de Imagens, permitindo a aplicação dessas técnicas em diferentes contextos e ajudando a obter resultados mais precisos e de melhor qualidade.

## **REFERÊNCIAS**

- [1] Gonzalez, R. C. e Woods, R. E.. **Processamento Digital de Imagens.** 3<sup>a</sup>. Pearson Prentice Hall. 2010.
- [2] Disponível em: <<https://sig-arq.ufpb.br/arquivos/20231911672e2d5189739c68d959de66a/PDI2023.pdf>>. Acesso em: 13 de Abril de 2023.