Universidade Federal do Rio Grande do Sul
INF01046 - Fundamentos de Processamento de Imagens - Turma B - Trabalho 2

Gabriela Copetti Maccagnan

04/12/2024

1 Transformações Lineares, Equalização e Matching de Histograma, Convolução e Filtragem no Domínio Espacial

Este trabalho tem como objetivo implementar transformações lineares, operações de equalização, matching de histograma, convolução e filtragem sobre imagens digitais, utilizando a linguagem C++, juntamente com as bibliotecas *OpenCV* para auxílio nas operações de processamento de imagens e a biblioteca *cvui* (https://fernandobevilacqua.com/cvui/) para a construção de uma interface para o programa.

O programa foi construído utilizando CMake, portanto possui um arquivo *CMakeLists.txt* e pode ser rodado a partir dos comandos na pasta root:

- cmake -B build (para fazer o build do programa)
- cmake –build build –config Release (para criar o executável na pasta \build\Release)
- .\build\Release\FPI _T2.exe (para rodar o executável)

O programa foi desenvolvido em cima do Trabalho 1, portanto sua estrutura é bastante similar, contando com um arquivo main.cpp, onde foi desenhada a interface e de onde são chamadas as funções de processamento a partir do clique nos botões da tela. Foram adicionadas algumas pastas para arquivos .cpp e .h para melhor separar as funções criadas de acordo com seu objetivo.

No novo programa foram mantidas as imagens teste apresentadas na Figura 1.

Figure 1: Imagens teste

Para fechar o programa, basta clicar na tecla *esc*.

1.1 Parte I

1.1.1 Calcular e exibir o histograma de uma imagem em tons de cinza

O histograma das imagens foi calculado a partir do código apresentado na Figura 3, no qual é percorrida a imagem identificando a frequência de aparecição de cada tom de cor de pixel e atribuindo a frequência à posição correspondente no vetor "histogram" criado de tamanho 256 (0 a 255). No caso de imagens coloridas, a imagem foi primeiro convertida para tons de cinza (luminância) e somente após foi calculado o histograma.

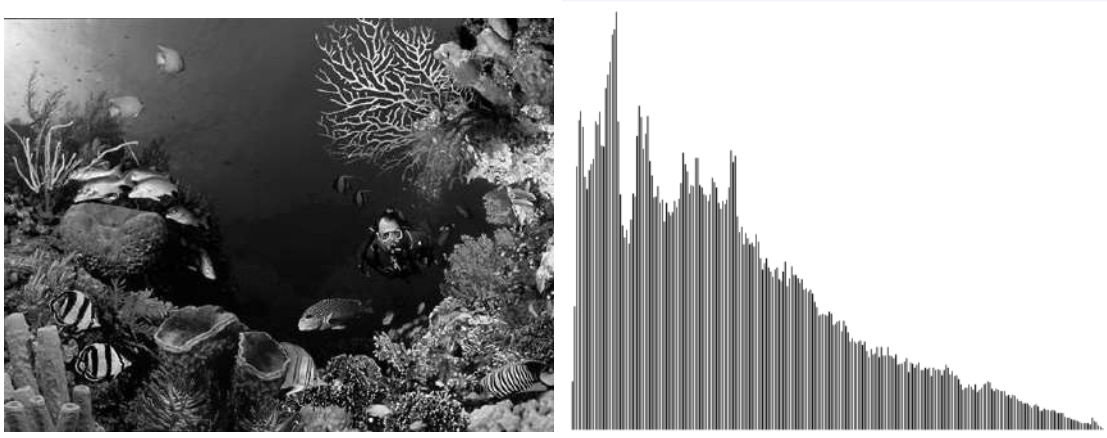

Figure 2: Comparação da imagem original convertida para grayscale com o histograma obtido a partir da imagem de luminância

```cpp
vector<int> makeHistogram(const Mat& img) {
    int hue = 256;

    vector<int> histogram(hue, 0);

    Mat grayImage(img.size(), img.type());

    if (img.channels() != 3) {
        grayImage = img.clone();
    } else {
        grayImage = convertToGS(img);
    }

    for (int y = 0; y < grayImage.rows; y++) {
        for (int x = 0; x < grayImage.cols; x++) {
            int pixel = grayImage.at<uchar>(y, x);
            histogram[pixel]++;
        }
    }

    return histogram;
}
```

Figure 3: Função para o cálculo do histograma

O histograma obtido para a imagem "Underwater" pode ser visto na Figura 2, onde cada coluna da histograma representa um tom de cinza.

1.1.2 Ajustar o brilho de uma imagem

O ajuste de brilho da imagem foi realizado percorrendo a imagem e somando um valor *bn* definido pelo usuário ao valor de cada pixel. No caso de imagens em escalas de cinza existe somente um canal de cor, mas para imagens coloridas a adição do valor é feito em cada um dos três canais.

```cpp
Mat adjustLight(const Mat& img, const int& bn) {
    Mat ajustedImage(img.size(), img.type());

    if (img.channels() != 3) {
        for (int y = 0; y < img.rows; y++) {
            for (int x = 0; x < img.cols; x++) {
                int pixel = img.at<uchar>(y, x) + bn;
                int newValue = verifyRange(pixel);
                ajustedImage.at<uchar>(y, x) = newValue;
            }
        }
    } else {
        for (int y = 0; y < img.rows; y++) {
            for (int x = 0; x < img.cols; x++) {
                Vec3b pixel = img.at<Vec3b>(y, x);
                for (int c = 0; c < img.channels(); c++) {
                    ajustedImage.at<Vec3b>(y, x)[c] = verifyRange(pixel[c] + bn);
                }
            }
        }
    }

    return ajustedImage;
}
```

Figure 4: Função para a aplicação do filtro de iluminação

Figure 5: Imagens resultante da aplicação da operação de iluminação na imagem original com valores positivos e negativos, respectivamente

1.1.3 Ajustar o contraste de uma imagem

Para ajustar o contraste da imagem, de maneira similar ao ajuste de brilho é percorrida a imagem, com a intenção de multiplicar o valor de cada pixel da imagem por um valor *cn* definido pelo usuário. No caso de imagens em escalas de cinza a multiplicação é feita no único canal, enquanto em imagens coloridas, a multiplicação dos valores é feita em cada canal.

```
Mat adjustContrast(const Mat& img, const int& cn) {
    Mat ajustedImage(img.size(), img.type());

    if (img.channels() != 3) {
        for (int y = 0; y < img.rows; y++) {
            for (int x = 0; x < img.cols; x++) {
                int pixel = img.at<uchar>(y, x) * cn;
                int newValue = verifyRange(pixel);
                ajustedImage.at<uchar>(y, x) = newValue;
            }
        }
    } else {
        for (int y = 0; y < img.rows; y++) {
            for (int x = 0; x < img.cols; x++) {
                Vec3b pixel = img.at<Vec3b>(y, x);
                for (int c = 0; c < img.channels(); c++) {
                    ajustedImage.at<Vec3b>(y, x)[c] = verifyRange(pixel[c] * cn);
                }
            }
        }
    }

    return ajustedImage;
}
```

Figure 6: Função para a aplicação do filtro de contraste

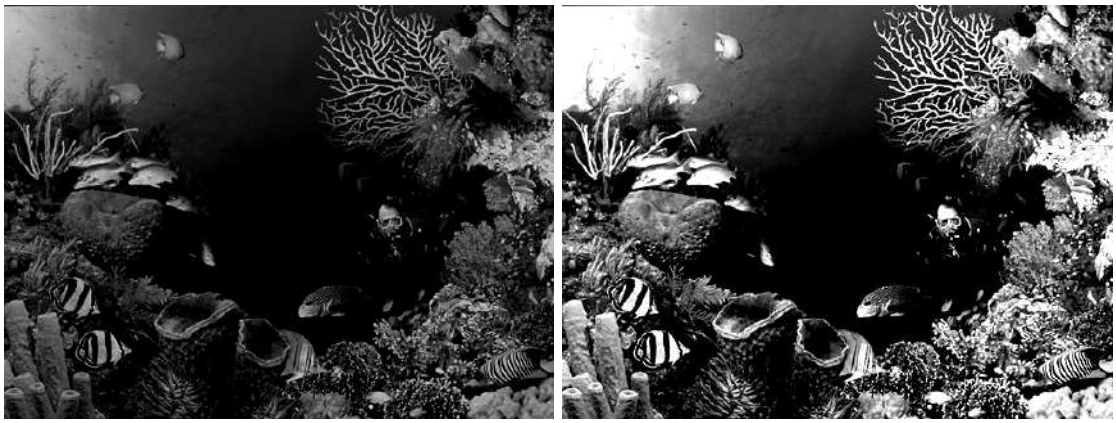

Figure 7: Comparação da imagem original em escala de cinza com a imagem resultante da aplicação da operação de contraste

1.1.4 Calcular e exibir o negativo de uma imagem

O filtro negativo de uma imagem é calculado percorrendo a imagem fonte e alterando o valor de cada pixel seguindo o formato pixel_novo = 255 - pixel_atual. Essa operação pode ser feita tanto em imagens de luminância quando em imagens coloridas, porém no segundo caso é necessário fazer a alteração dos valores de pixel nos três canais da imagem.

```cpp
Mat convertToNegative(const Mat& img) {
    Mat negativeImage(img.size(), img.type());

    if (img.channels() != 3) {
        for (int y = 0; y < img.rows; y++) {
            for (int x = 0; x < img.cols; x++) {
                int pixelHue = img.at<uchar>(y, x);
                negativeImage.at<uchar>(y, x) = 255 - pixelHue;
            }
        }
    } else {
        for (int y = 0; y < img.rows; y++) {
            for (int x = 0; x < img.cols; x++) {
                Vec3b pixel = img.at<Vec3b>(y, x);
                for (int c = 0; c < img.channels(); c++) {
                    negativeImage.at<Vec3b>(y, x)[c] = 255 - pixel[c];
                }
            }
        }
    }

    return negativeImage;
}
```

Figure 8: Função para a aplicação do filtro negativo

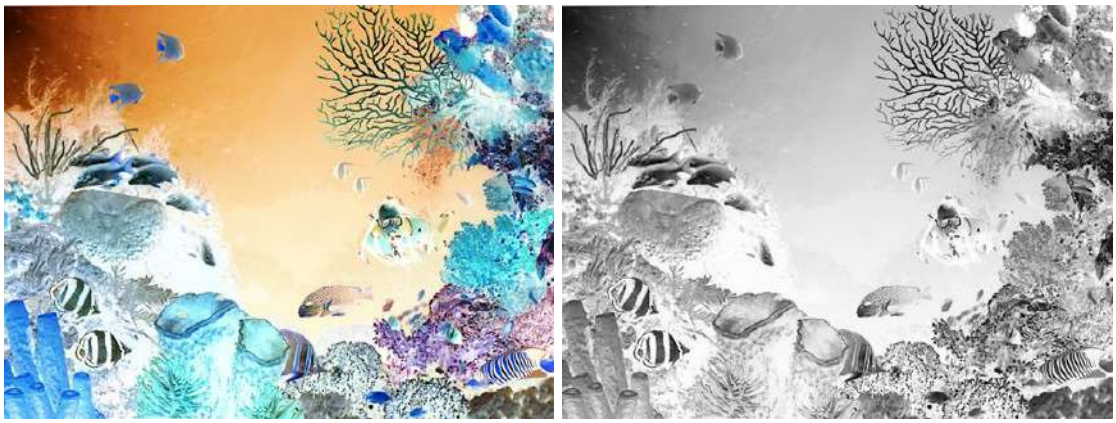

Figure 9: Comparação das imagens resultantes do filtro negativo em uma imagem colorida e uma imagem em escala de cinza, respectivamente

1.1.5 Equalizar o histograma de uma imagem

A operação de equalização das imagens foi desenvolvida a partir do pseudo-código oferecido em aula, onde primeiro é calculado o histograma cumulativo através do histograma da imagem (obtido na primeira questão) e um fator de escala. Em um segundo passo é percorrida a imagem, atribuindo o valor de cada pixel da nova imagem segundo o histograma cumulativo calculado. Para

imagens coloridas, o histograma cumulativo utilizado para cada um dos canais (R, G, B) foi obtido a partir da imagem de luminância.

```
Mat equalize(const Mat &img) {
    if (img.channels() == 3) {
        return equalizeColor(img);
    }

    Mat equalizedImage(img.size(), img.type());

    vector<int> histogram = makeHistogram(img);

    int hue = 256;
    vector<int> histogram_cum(hue, 0);

    int num_pixels = img.total();
    float scaling_factor = 255.0 / num_pixels;

    histogram_cum[0] = static_cast<int>(scaling_factor * histogram[0]);
    for (int i = 1; i < hue; i++) {
        histogram_cum[i] = histogram_cum[i - 1] + static_cast<int>(scaling_factor * histogram[i]);
    }

    for (int y = 0; y < img.rows; y++) {
        for (int x = 0; x < img.cols; x++) {
            equalizedImage.at<uchar>(y, x) = histogram_cum[img.at<uchar>(y,x)];
        }
    }

    return equalizedImage;
}
```

Figure 10: Função para a aplicação da equaliação em imagens em escala de cinza

```
Mat equalizeColor(const Mat &img) {
    Mat equalizedImage(img.size(), img.type());

    int hue = 256;
    vector<int> histogram_cum(hue, 0);

    int num_pixels = img.total();
    float scaling_factor = 255.0 / num_pixels;

    vector<int> histogram = makeHistogram(img);
    histogram_cum[0] = static_cast<int>(scaling_factor * histogram[0]);
    for (int i = 1; i < hue; i++) {
        histogram_cum[i] = histogram_cum[i - 1] + static_cast<int>(scaling_factor * histogram[i]);
    }

    for (int y = 0; y < img.rows; y++) {
        for (int x = 0; x < img.cols; x++) {
            for (int c = 0; c < img.channels(); c++) {
                equalizedImage.at<Vec3b>(y, x)[c] = histogram_cum[img.at<Vec3b>(y, x)[c]];
            }
        }
    }

    return equalizedImage;
}
```

Figure 11: Função para a aplicação da equalização em imagens coloridas

Figure 12: Comparação das imagens resultantes da equalização de uma imagem colorida e uma imagem em escala de cinza, respectivamente

1.1.6 Realizar Histogram Matching de pares de imagens em tons de cinza

Para adquirir o histogram matching de duas imagens também foi utilizado o pseudo-código oferecido em aula. Aqui são calculados os histogramas cumulativos das duas imagens, a original e a alvo, caso elas sejam imagens coloridas é primeiro feita uma conversão das duas imagens. Após,realiza-se um mapeamento para encontrar o valor de cor mais próximo com relação à imagem alvo.

```
Mat histogramMatching(const Mat &img_src, const Mat &img_target) {
    Mat matchedImage(img_src.size(), CV_8UC1);

    Mat grayImage_src(img_src.size(), CV_8UC1);
    Mat grayImage_target(img_target.size(), CV_8UC1);

    if (img_src.channels() != 3) {
        grayImage_src = img_src.clone();
    } else {
        grayImage_src = convertToGS(img_src);
    }

    if (img_target.channels() != 3) {
        grayImage_target = img_target.clone();
    } else {
        grayImage_target = convertToGS(img_target);
    }

    int hue = 256;
    vector<int> histogram_src_cum(hue, 0);
    vector<int> histogram_target_cum(hue, 0);
    vector<int> histogram_matched(hue, 0);

    vector<int> histogram_src = makeHistogram(grayImage_src);
    vector<int> histogram_target = makeHistogram(grayImage_target);
```

```cpp
int num_pixels_src = grayImage_src.total();
float scaling_factor_src = 255.0 / num_pixels_src;

int num_pixels_target = grayImage_target.total();
float scaling_factor_target = 255.0 / num_pixels_target;


histogram_src_cum[0] = static_cast<int>(scaling_factor_src * histogram_src[0]);
histogram_target_cum[0] = static_cast<int>(scaling_factor_target * histogram_target[0]);
for (int i = 1; i < hue; i++) {
    histogram_src_cum[i] = histogram_src_cum[i - 1] + static_cast<int>(scaling_factor_src * histogram_src[i]);
    histogram_target_cum[i] = histogram_target_cum[i - 1] + static_cast<int>(scaling_factor_target * histogram_target[i]);
}

for (int i = 0; i < hue; i++) {
    int min_diff = abs(histogram_src_cum[i] - histogram_target_cum[0]);
    int best_match = 0;

    for (int j = 1; j < hue; j++) {
        int diff = abs(histogram_src_cum[i] - histogram_target_cum[j]);
        if (diff < min_diff) {
            min_diff = diff;
            best_match = j;
        }
    }
    histogram_matched[i] = best_match;
}
```

```cpp
 for (int y = 0; y < grayImage_src.rows; y++) {
     for (int x = 0; x < grayImage_src.cols; x++) {
         matchedImage.at<uchar>(y, x) = histogram_matched[grayImage_src.at<uchar>(y, x)];
     }
 }

 return matchedImage;
}
```

Figure 13: Função para a aplicação do histogram matching

Figure 14: Comparação da imagem usada como objetivo do matching e a imagem resultante da operação

1.2 Parte II

1.2.1 Reduzir uma imagem (zoom out) utilizando fatores de redução s_x e s_y

A operação de redução da imagem foi implementada de forma que o usuário pode definir valores de s_x e s_y que são usados como novas dimensões para a imagem, podendo essas duas variáveis serem iguais ou não. Após, é percorrida a imagem original e, para cada pixel, é percorrido o retângulo de tamanho s_x e s_y calculando a média dos valores de pixel no retângulo. Essa média é aplicada ao valor do pixel correspondente ao retângulo reduzido.

```
for (int y = 0; y < zoomedImage.rows; y++) {
    for (int x = 0; x < zoomedImage.cols; x++) {
        int sum = 0;
        int count = 0;

        for (int j = 0; j < sy; j++) {
            for (int i = 0; i < sx; i++) {
                int oldY = y * sy + j;
                int oldX = x * sx + i;

                if (oldY < img.rows && oldX < img.cols) {
                    uchar pixel = img.at<uchar>(oldY, oldX);
                    sum += pixel;
                    count++;
                }
            }
        }

        zoomedImage.at<uchar>(y, x) = uchar(static_cast<uchar>(sum / count));
    }
}
```

Figure 15: Função para a aplicação do zoom out em imagens em escala de cinza

Na Figura 15 está apresentada apenas uma parte da implementação, correspondente a redução de uma imagem en tons de cinza. Para a redução de imagens coloridas é necessário calcular a média de pixels para os três canais de cor.

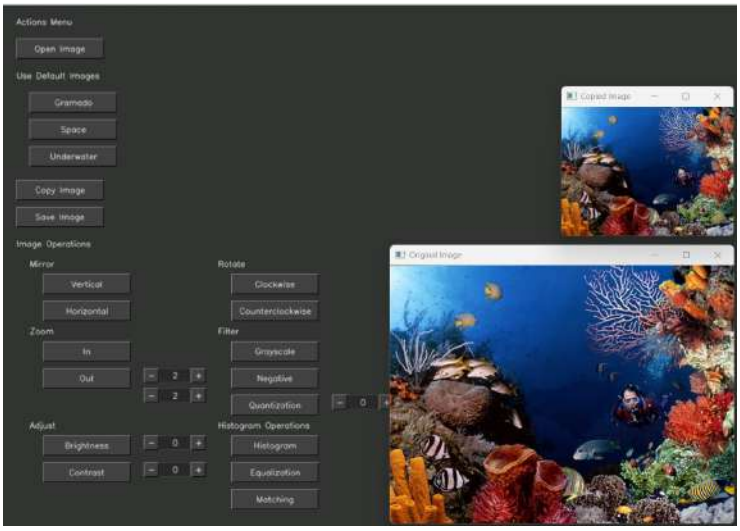

Figure 16: Imagem resultante da redução da imagem original com valores iguais para s_x e s_y

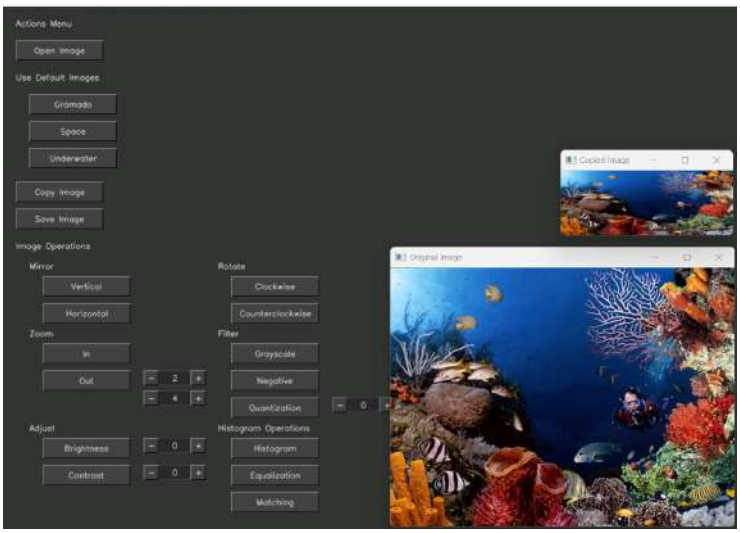

Figure 17: Imagem resultante da redução da imagem original com valores diferentes para s_x e s_y

1.2.2 Ampliar a imagem (zoom in) utilizando um fator de 2x2 a cada vez

A operação de ampliação da imagem foi implementada conforme especificado no trabalho. Primeiramente foi alocado um tamanho para nova imagem 4 vezes maior que a original e após foi implementado o processo de ampliação utilizando operações 1D em dois passos. No primeiro passo foram inseridas uma linha e coluna em branco a cada duas linhas e colunas originais e no segundo passo foram interpolados os valores existentes com objetivo de preencher os espaços vazios.

Na figura abaixo está apresentada apenas uma parte da implementação, que seria a parte correspondente a ampliação de uma imagem en tons de cinza, mas imagens coloridas também podem ser ampliadas, com a única diferença sendo que em imagens coloridas é necessário inserir valores nos três canais de cor.

```
for (int y = 0; y < zoomedImage.rows; y += 2) {
    for (int x = 1; x < zoomedImage.cols - 1; x += 2) {
        int previousPixel = zoomedImage.at<uchar>(y, x - 1);
        int nextPixel = zoomedImage.at<uchar>(y, x + 1);
        int currentPixel = (previousPixel + nextPixel) / 2;
        zoomedImage.at<uchar>(y, x) = currentPixel;
    }
}

for (int x = 0; x < zoomedImage.cols; x ++) {
    for (int y = 1; y < zoomedImage.rows - 1; y += 2) {
        int previousPixel = zoomedImage.at<uchar>(y - 1, x);
        int nextPixel = zoomedImage.at<uchar>(y + 1, x);
        int currentPixel = (previousPixel + nextPixel) / 2;
        zoomedImage.at<uchar>(y, x) = currentPixel;
    }
}
```

Figure 18: Função para a aplicação do zoom in em imagens em escala de cinza

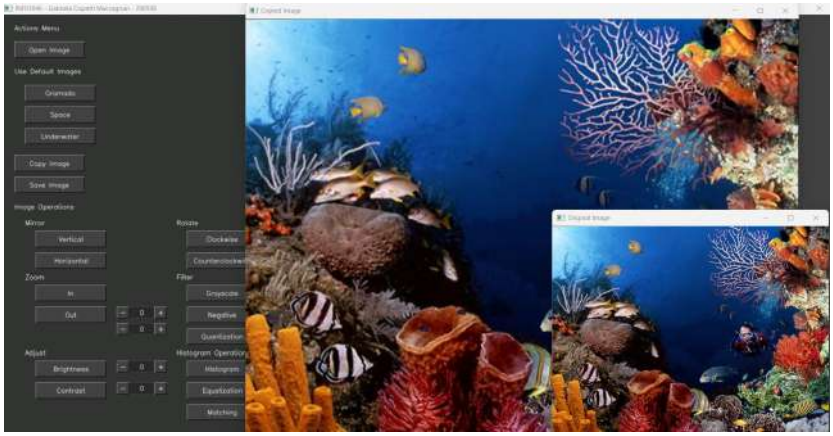

Figure 19: Imagem resultante da ampliação da imagem original

1.2.3 Rotacionar imagem de 90° (tanto no sentido horário como no sentido anti-horário

A operação de rotação foi implementada utilizando a função de memcpy, alterando o valor de cada pixel da imagem de acordo com o sentido da rotação. A operação pode ser aplicada múltiplas vezes em qualquer sentido.

```
Mat rotateImage(const Mat& img, boolean clockwise) {
    Mat rotatedImage(img.cols, img.rows, img.type());
    int channels = img.channels();

    if (clockwise) {
        for (int y = 0; y < img.rows; y++) {
            for (int x = 0; x < img.cols; x++) {
                const uchar* oldRow = img.ptr<uchar>(y) + x * channels;
                uchar* newColumn = rotatedImage.ptr<uchar>(x) + (img.rows - y - 1) * channels;
                memcpy(newColumn, oldRow, channels);
            }
        }
    } else {
        for (int y = 0; y < img.rows; y++) {
            for (int x = 0; x < img.cols; x++) {
                const uchar* oldRow = img.ptr<uchar>(y) + x * channels;
                uchar* newColumn = rotatedImage.ptr<uchar>(img.cols - x - 1) + y * channels;
                memcpy(newColumn, oldRow, channels);
            }
        }
    }

    return rotatedImage;
}
```

Figure 20: Função para a aplicação da rotação

Figure 21: Imagem resultante da rotação em 90° da imagem original

1.2.4 Implementar um procedimento para realizar convolução entre uma imagem e um filtro 3x3 arbitrário

A função de convolução tem como objetivo percorrer a imagem aplicando a operação a um pixel E central e utilizando um kernel 3x3. A chamada rotateKernel faz com que o kernel seja rotacionado em 180° antes de ser aplicado. Ignorando as borda, a imagem foi percorrida calculando para cada pixel os valores iA, hB, gC, fD, eE, dF, cG, bH e aI que significam a multiplicação do peso do kernel pelos pixels da vizinhança de E. O pixel E, portanto, tem como resultado a soma desses valores calculados.

O resultado da convolução de cada pixel pode ser maior que 255 ou menor que zero, sendo assim foi definido um valor máximo e mínimo para o pixel antes de aplicá-lo à nova imagem (operação de clamping). Em alguns casos, deve-se somar 127 ao resultado da convolução antes de fazer o clamping dos pixels, isso é controlado pela variável $shouldMove$. A convolução nesse caso só pode ser aplicada sob imagens em tons de cinza.

```
Mat applyConvolution(const Mat& img, float data[], bool shouldMove) {
    Mat convImage(img.size(), img.type());
    Mat kernel = Mat(3, 3, CV_32F, data).clone();
    rotateKernel(kernel);

    if (img.channels() != 3) {
        for (int y = 1; y < img.rows - 1; y++) {
            for (int x = 1; x < img.cols - 1; x++) {
                int iA = img.at<uchar>(y - 1, x - 1) * kernel.at<float>(0, 0);
                int hB = img.at<uchar>(y - 1, x) * kernel.at<float>(0, 1);
                int gC = img.at<uchar>(y - 1, x + 1) * kernel.at<float>(0, 2);
                int fD = img.at<uchar>(y, x - 1) * kernel.at<float>(1, 0);
                int eE = img.at<uchar>(y, x) * kernel.at<float>(1, 1);
                int dF = img.at<uchar>(y, x + 1) * kernel.at<float>(1, 2);
                int cG = img.at<uchar>(y + 1, x - 1) * kernel.at<float>(2, 0);
                int bH = img.at<uchar>(y + 1, x) * kernel.at<float>(2, 1);
                int aI = img.at<uchar>(y + 1, x + 1) * kernel.at<float>(2, 1);
                int pixel = iA + hB + gC + fD + eE + dF + cG + bH + aI;

                if (shouldMove) {
                    pixel += 127;
                }
                pixel = max(0, min(255, pixel));

                convImage.at<uchar>(y, x) = pixel;
            }
        }
    } else {
        cout << GRAYSCALE_ERROR << endl;
        convImage = img.clone();
    }

    return convImage;
}
```

Figure 22: Função para a aplicação da convolução

Segundo o trabalho, foram definidos alguns kernels default que podem ser usados pelo usuário, além da possibilidade do usuário definir o seu próprio kernel.

O primeiro kernel default é o Gaussiano, filtro passa baixa que suaviza as bordas da imagem (efeito de borramento). Sua implementação e os resultados da sua aplicação são encontrados abaixo.

```
Mat applyGaussian(const Mat& img) {
    float data[] = {
        0.0625, 0.125, 0.0625,
        0.125,  0.25,  0.125,
        0.0625, 0.125, 0.0625
    };

    Mat convImage = applyConvolution(img, data, false);

    return convImage;
}
```

Figure 23: Função para a aplicação do kernel Gaussiano

Figure 24: Imagens resultantes da aplicação do kernel Gaussiano 1x e 3x, respectivamente

Após temos o kernel Laplaciano, um filtro passa altas que detecta arestas importantes.

```
Mat applyLaplacian(const Mat& img) {
    float data[] = {
        0, -1,  0,
       -1,  4, -1,
        0, -1,  0
    };

    Mat convImage = applyConvolution(img, data, true);

    return convImage;
}
```

Figure 25: Função para a aplicação do kernel Laplaciano

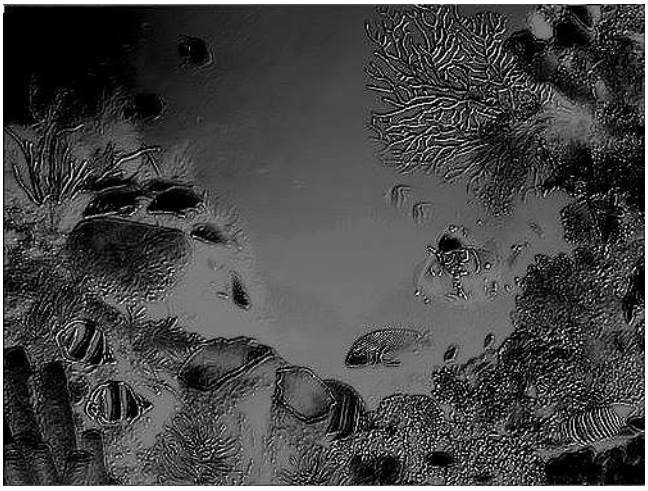

Figure 26: Imagem resultante da aplicação do kernel Laplaciano

O próximo filtro é mais um outro passa alta genérico, usado como um detector de arestas mais sensitivo.

```
Mat applyHighPass(const Mat& img) {
    float data[] = {
        -1, -1, -1,
        -1,  8, -1,
        -1, -1, -1
    };

    Mat convImage = applyConvolution(img, data, false);

    return convImage;
}
```

Figure 27: Função para a aplicação do kernel Passa Alta

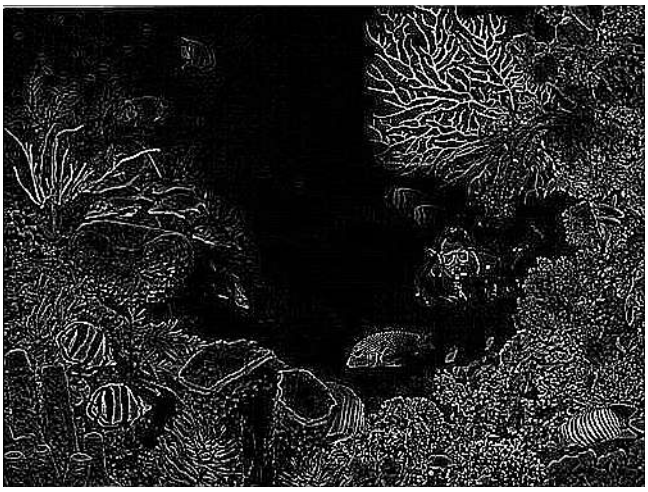

Figure 28: Imagem resultante da aplicação do kernel Passa Alta genérico

Os próximos dois kernels default são referentes ao filtro de Prewitt. O primeiro é Prewitt Hx ou Horizontal que produz um efeito de relevo. Sua implementação e os resultados da sua aplicação são encontrados abaixo.

```
Mat applyPrewittHx(const Mat& img) {
    float data[] = {
        -1, 0, 1,
        -1, 0, 1,
        -1, 0, 1
    };

    Mat convImage = applyConvolution(img, data, true);

    return convImage;
}
```

Figure 29: Função para a aplicação do kernel Prewitt Hx

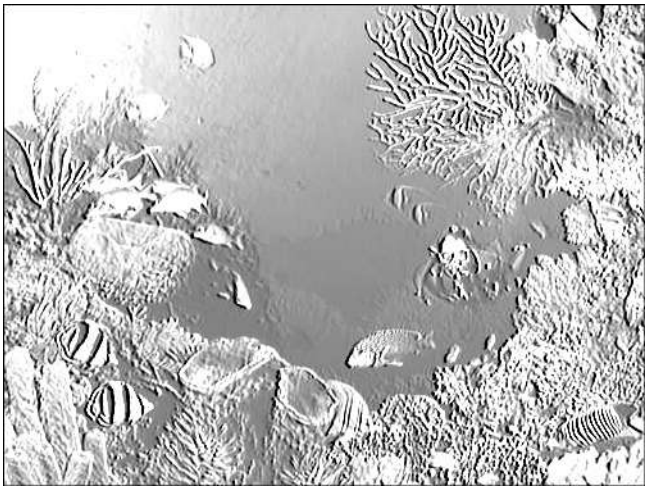

Figure 30: Imagem resultante da aplicação do kernel Prewitt Hx

O segundo é Prewitt Hy, que de maneira similar também produz um efeito de relevo porém no sentido vertical.

```
Mat applyPrewittHy(const Mat& img) {
    float data[] = {
        -1, -1, -1,
         0,  0,  0,
         1,  1,  1
    };

    Mat convImage = applyConvolution(img, data, true);
    return convImage;
}
```

Figure 31: Função para a aplicação do kernel Prewitt Hy

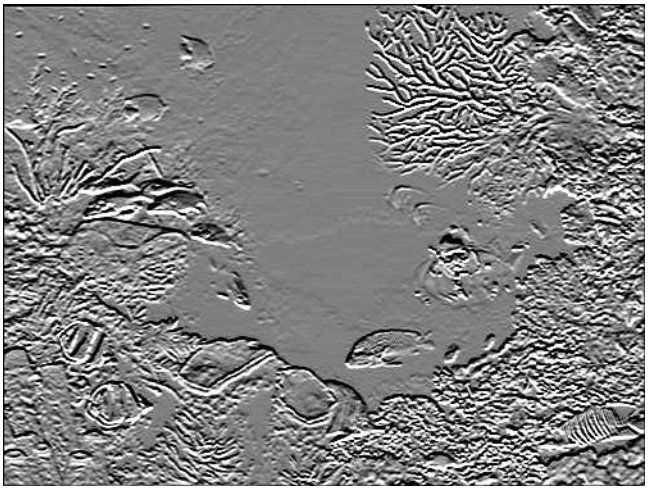

Figure 32: Imagem resultante da aplicação do kernel Prewitt Hy

Os próximos dois filtros são referentes ao filtro de Sobel. Sobel Hx é um filtro mais sensível ao gradiente dos tons de cinza na direção horizontal e também produz um efeito de relevo.

```
Mat applySobelHx(const Mat& img) {
    float data[] = {
        -1, 0, 1,
        -2, 0, 2,
        -1, 0, 1
    };

    Mat convImage = applyConvolution(img, data, true);
    return convImage;
}
```

Figure 33: Função para a aplicação do kernel Sobel Hx

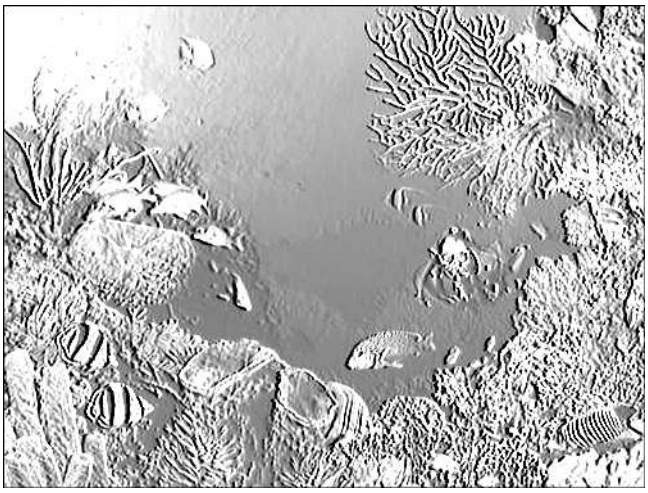

Figure 34: Imagem resultante da aplicação do kernel Sobel Hx

O segundo é o filtro Sobel Hy, que age na direção vertical e também produz um efeito de relevo.

```
Mat applySobelHy(const Mat& img) {
    float data[] = {
        -1, -2, -1,
         0,  0,  0,
         1,  2,  1
    };

    Mat convImage = applyConvolution(img, data, true);
    return convImage;
}
```

Figure 35: Função para a aplicação do kernel Sobel Hy

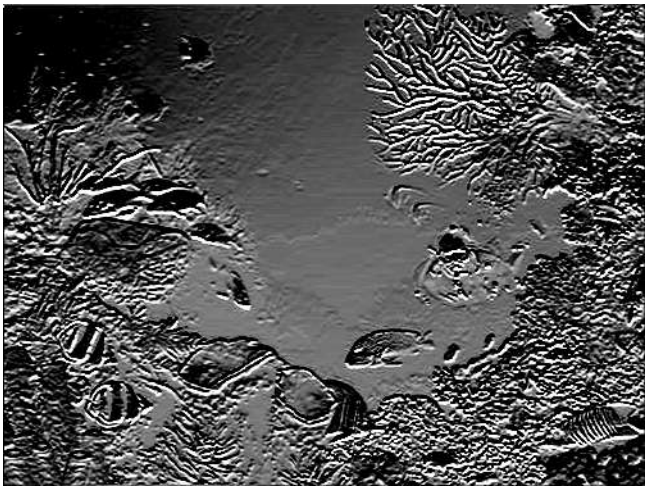

Figure 36: Imagem resultante da aplicação do kernel Sobel Hy

1.3 Interface

A seguir serão apresentadas algumas figuras referentes à interface atualizada, contendo as funções desenvolvidas no Trabalho 1, assim como as novas funções do Trabalho 2. A figura abaixo, por exemplo, apresenta a interface em sua extensão total, na parte superior da tela estão as operações iniciais de abertura, cópia e salvamento da imagem, enquanto na parte inferior estão todas as operações de modificação que podem ser aplicadas às imagens.

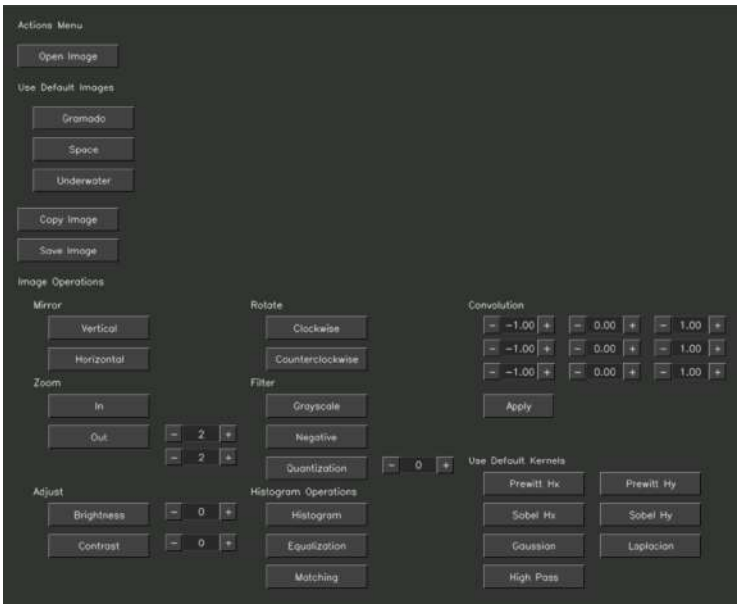

Figure 37: Interface inteira

No menu inicial temos operações já disponibilizadas no primeiro trabalho, com a possibilidade de escolher uma imagem ou utilizar uma imagem default, copiar a imagem para realizar as modificações e, após, salvar a imagem modificada.

Figure 38: Menu inicial com operações para abrir, copiar e salvar a imagem

Na primeira parte do menu de operações temos as funções de espelhamento

vertical e horizontal, de ampliação da imagem, de redução da imagem (nesse caso o usuário deve informar os fatores de redução desejados através dos componentes de contador ao lado), de ajuste de brilho e de contraste. Para as últimas duas operações o usuário também deve informar um valor antes de aplicar o ajuste à imagem.

Figure 39: Parte do menu de operações contendo funções de espelhamento, zoom, iluminação e contraste

Em seguida, temos no menu as operações de rotação em sentido horário e anti-horário, os filtros de escala de cinza, negativo e a operação de quantização com valor definido pelo usuário. Abaixo estão as operações que envolvem histogramas: a própria operação de criação de um histograma da imagem, a operação de equalização que cria um histograma para a imagem original e para a imagem equalizada, e a operação de histogram matching.

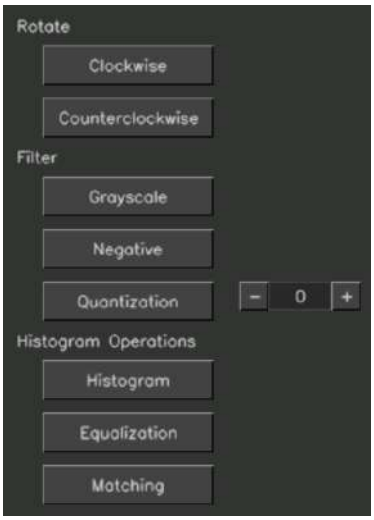

Figure 40: Parte do menu de operações contendo funções de rotação, filtros e histogramas

A última seção do menu conta exclusivamente com a operação de convolução. Nesse caso o usuário pode definir valores para cada uma das posições do kernel 3x3 e aplicar a convolução com o kernel criado à imagem, ou escolher um dos kernels pré-definidos.

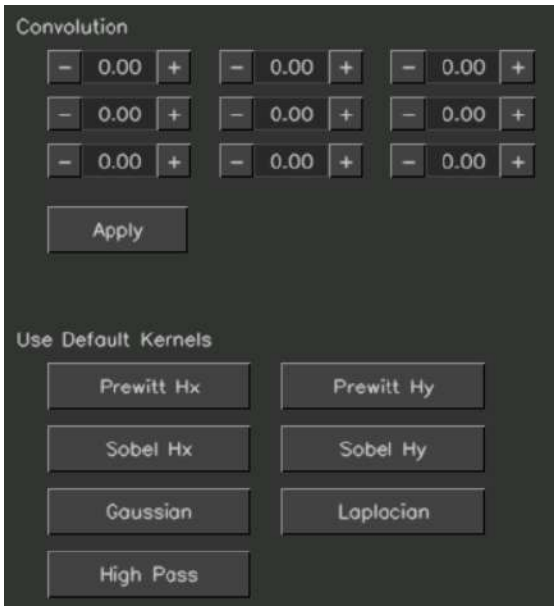

Figure 41: Parte do menu de operações contendo funções de convolução