Processamento Digital de Imagens

Table of Contents

1. Manipulando pixels em uma imagem.

A manipulação de pixels em uma imagem refere-se ao processo de alterar as propriedades dos pixels individuais que compõe a imagem, como, por exemplo, modificar o valor da cor do pixel, alterar a sua posição na imagem, aplicar filtros ou efeitos especiais, entre outros. Desta forma, é possível realizar uma infinidade de tarefas, como redimensionar uma imagem, remover objetos indesejados, corrigir imperfeições, aplicar efeitos artísticos, criar animações, entre outras aplicações criativas e práticas.

1.1. Filtro Negativo (regions.cpp)

Um filtro negativo em uma imagem é uma técnica de manipulação de pixels que inverte as cores da imagem original. Nesse filtro, cada pixel da imagem é transformado em seu complemento, resultando em uma imagem com cores invertidas. Além de criar um efeito estético interessante, o

filtro negativo também pode ser útil em certas aplicações, como melhorar a visualização de detalhes em imagens com alto contraste ou realçar certos elementos. No entanto, é importante notar que a aplicação de um filtro negativo em uma imagem não é uma técnica que preserva informações importantes da imagem original, mas sim uma transformação visual que pode ser usada para efeitos artísticos ou estilísticos.

1.1.1. Código & Resultado.

Implemente um programa regions.cpp. Esse programa deverá solicitar ao usuário as coordenadas de dois pontos P1 e P2 localizados dentro dos limites do tamanho da imagem e exibir que lhe for fornecida. Entretanto, a região definida pelo retângulo de vértices opostos definidos pelos pontos P1 e P2 será exibida com o negativo da imagem na região correspondente.

regions.cpp

```
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main() {
    cout << "INICIANDO..." << endl;

    Mat image;
    int P1X, P1Y, P2X, P2Y;
    char diretorio[1000];

    cout << "Digite a loc da imagem: " << endl;
    cout << "EX.: C:\\User\\Desktop\\<nomedaimagem>.png" << endl;
    cin >> diretorio;

    auto imageO = imread(diretorio);
    image = imread(diretorio, IMREAD_GRAYSCALE);

    int rows = image.rows;
    int cols = image.cols;

    cout << "Digite o ponto P1 da imagem" << endl;
    cout << "Entre Linha 0 e" << " " << (rows - 1) << " " << "e Coluna 0 e" << " " <<
(cols - 1) << endl;
    cin >> P1X >> P1Y;
    cout << "(" << P1X << "," << P1Y << ")" << endl;
    cout << "Digite 0 para cancelar e 1 para confirmar" << endl;

    int confirmaP1;

    do {
        cin >> confirmaP1;
        if (confirmaP1 == 0) {
```

```
            cout << "Digite o ponto P1 da imagem" << endl;
            cout << "Entre Linha 0 e" << " " << (rows - 1) << " " << "e Coluna 0 e" <<
" " << (cols - 1) << endl;
            cin >> P1X >> P1Y;
            cout << "(" << P1X << "," << P1Y << ")" << endl;

            cout << "Digite 0 para cancelar e 1 para confirmar" << endl;
            cin >> confirmaP1;
        }

        if (confirmaP1 == 1 && (P1X < 0 || P1X > rows)) {
            cout << "Valor de X do ponto P1 localiza-se fora da imagem, digite
novamente P1X: " << endl;
            cin >> P1X;

            cout << "(" << P1X << "," << P1Y << ")" << endl;
            cout << "Digite 0 para cancelar e 1 para confirmar" << endl;
            cin >> confirmaP1;
        }

        if (confirmaP1 == 1 && (P1Y< 0 || P1Y > cols)) {
            cout << "Valor de Y do ponto P1 localiza-se fora da imagem, digite
novamente P1Y: " << endl;
            cin >> P1Y;

            cout << "(" << P1X << "," << P1Y << ")" << endl;
            cout << "Digite 0 para cancelar e 1 para confirmar" << endl;
            cin >> confirmaP1;
        }

    } while (confirmaP1 == 0);

    cout << "Digite o ponto P2 da imagem" << endl;
    cout << "Entre Linha 0 e" << " " << (rows - 1) << " " << "e Coluna 0 e" << " " <<
(cols - 1) << endl;
    cin >> P2X >> P2Y;
    cout << "(" << P2X << "," << P2Y << ")" << endl;
    cout << "Digite 0 para cancelar e 1 para confirmar" << endl;

    int confirmaP2;
    do {
        cin >> confirmaP2;
        if (confirmaP2 == 0) {
            cout << "Digite o ponto P2 da imagem" << endl;
            cout << "Entre Linha 0 e" << " " << (rows - 1) << " " << "e Coluna 0 e" <<
" " << (cols - 1) << endl;
            cin >> P2X >> P2Y;
            cout << "(" << P2X << "," << P2Y << ")" << endl;

            cout << "Digite 0 para cancelar e 1 para confirmar" << endl;
            cin >> confirmaP2;
```

```
        }

        if (confirmaP2 == 1 && (P2X < 0 || P2X > rows || P2X < P1X)) {
            cout << "Valor de X do ponto P2 localiza-se fora da imagem ou eh menor que
P1X, digite novamente P2X: " << endl;
            cin >> P2X;

            cout << "(" << P2X << "," << P2Y << ")" << endl;
            cout << "Digite 0 para cancelar e 1 para confirmar" << endl;
            cin >> confirmaP2;
        }

        if (confirmaP2 == 1 && (P2Y< 0 || P2Y > cols || P2Y < P1Y)) {
            cout << "Valor de Y do ponto P2 localiza-se fora da imagem ou eh menor que
P1Y, digite novamente P2Y: " << endl;
            cin >> P2Y;

            cout << "(" << P2X << "," << P2Y << ")" << endl;
            cout << "Digite 0 para cancelar e 1 para confirmar" << endl;
            cin >> confirmaP2;
        }
    } while (confirmaP2 == 0);

    if (!image.data) {
        cout << "Imagem nao encontrada!" << endl;
    }

    for (int i = P1X; i < P2X; i++) {
        for (int j = P1Y; j < P2Y; j++) {

            image.at<uchar>(i, j) = 255 - image.at<uchar>(i, j);

        }
    }

    imwrite("janelaNegativo.png", image);
    namedWindow("janelaOriginal", WINDOW_AUTOSIZE);
    imshow("janelaOriginal", imageO);
    namedWindow("janelaNegativo", WINDOW_AUTOSIZE);
    imshow("janelaNegativo", image);

    waitKey();

    return 0;

}
```

Figure 1. Biel original

Figure 2. Biel com filtro negativo

1.2 Troca de regiões

A transposição de quadrante envolve a troca desses quadrantes, de modo que as baixas frequências fiquem no quadrante inferior direito e as altas frequências no quadrante superior esquerdo. Essa operação é frequentemente realizada para fins de visualização ou processamento de imagens, uma vez que a transposição pode melhorar a interpretação visual ou permitir a aplicação de determinadas técnicas de filtragem ou análise. Após a transposição de quadrante, é possível realizar operações de filtragem ou análise no domínio da frequência e, em seguida, reverter a imagem para o domínio espacial, se necessário.

1.2.1 Código & Resultado.

Implemente um programa trocaregioes.cpp. Seu programa deverá trocar os quadrantes em diagonal na imagem.

trocaderegioes.cpp

```
// Código realizado para rodar em WINDOWS sem makeFile, atente-se as intruções!
#include <iostream>
```

```
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main() {
    cout << "INICIANDO..." << endl;

    //DEFININDO VARIAVEIS ...
    char diretorio[10000];
    Mat image, imageT;


    //RECEBENDO LOCALIZAÇÃO DA IMAGEM...
    do {
        cout << "Digite a localizacao da imagem: " << endl;
        cout << "EX.: C:\\User\\Desktop\\<nomedaimagem>.<png ou jpeg>" << endl;
        cin >> diretorio;

        image = imread(diretorio);

        if (!image.data) {
            cout << "Imagem nao encontrada!" << endl;
            cout << "Verifique se a escrita esta correta." << endl;
        }
    } while (!image.data);

    image.copyTo(imageT);

    int rows4q, cols4q;

    rows4q = image.rows / 2;
    cols4q = image.cols / 2;

    // DIVIDINDO OS QUADRANTES ...
    Mat q1, q2, q3, q4;
    q1 = image(Rect(0, 0, rows4q, cols4q)); // Esquerdo superior
    q2 = image(Rect(0, cols4q, rows4q, cols4q)); // Direito superior
    q3 = image(Rect(rows4q, 0, rows4q, cols4q)); // Esquerdo inferior
    q4 = image(Rect(rows4q, cols4q, rows4q, cols4q)); // Direito inferior

    cout << rows4q << " e " << cols4q << endl;

    //TROCANDO OS QUADRANTES EM DIAGONAL ...
    q1.copyTo(imageT(Rect(rows4q, cols4q, rows4q, cols4q)));
    q2.copyTo(imageT(Rect(rows4q, 0, rows4q, cols4q)));
    q3.copyTo(imageT(Rect(0, cols4q, rows4q, cols4q)));
    q4.copyTo(imageT(Rect(0, 0, rows4q, cols4q)));

    namedWindow("janelaOriginal", WINDOW_AUTOSIZE);
    imshow("janelaOriginal", image);
```

```
    namedWindow("janelaTrocado", WINDOW_AUTOSIZE);
    imshow("janelaTrocado", imageT);
    imwrite("janelaTrocado.png", imageT);
    waitKey();

    return 0;
}
```

Figure 3. Biel original

Figure 4. Biel com quadrantes trocados

2. Serialização via FileStore

A serialização refere-se ao processo de converter dados em uma representação que possa ser armazenada ou transmitida, permitindo sua recuperação posterior. No contexto da programação, a serialização é comumente usada para salvar dados em um formato persistente, como um arquivo, para que possam ser recuperados posteriormente e usados novamente.

O ponto flutuante é um formato numérico usado para representar números reais em computadores. Ele permite representar uma ampla gama de valores, incluindo números fracionários e números muito grandes ou muito pequenos. A serialização de dados em ponto flutuante via FileStorage é especialmente útil quando se lida com grandes conjuntos de dados

numéricos, como matrizes ou imagens, que precisam ser armazenados e recuperados sem perda de precisão.

2.1. FileStore

O FileStorage é uma funcionalidade oferecida por algumas bibliotecas de programação, como OpenCV, que permite armazenar dados em um arquivo com uma estrutura organizada. Essa estrutura pode incluir seções, como grupos ou tags, que ajudam a organizar os dados serializados. Além disso, o FileStorage fornece métodos para escrever e ler dados em vários formatos, incluindo números de ponto flutuante.

2.1.1 Código & Resultado.

Crie um programa que gere uma imagem de dimensões 256x256 pixels contendo uma senóide de 4 períodos com amplitude de 127 desenhada na horizontal. Grave a imagem no formato PNG e no formato YML. Compare os arquivos gerados, extraindo uma linha de cada imagem gravada e comparando a diferença entre elas. Trace um gráfico da diferença calculada ao longo da linha correspondente extraída nas imagens. O que você observa?

filestorage.cpp

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <sstream>
#include <string>

using namespace std;
using namespace cv;

int SIDE = 256;
int PERIODOS = 4;
int M_PI = 3.141516;


int main(int argc, char** argv) {
    cout << "INICIANDO..." << endl;

    stringstream ss_img, ss_yml;
    Mat image;

    ss_yml << "senoide-" << SIDE << ".yml";
    image = Mat::zeros(SIDE, SIDE, CV_32FC1);

    FileStorage fs(ss_yml.str(), FileStorage::WRITE);

    for (int i = 0; i < SIDE; i++) {
        for (int j = 0; j < SIDE; j++) {
            image.at<float>(i, j) = 127 * sin(2 * M_PI * PERIODOS * j / SIDE) + 128;
        }
    }
```

```
    fs << "mat" << image;
    fs.release();

    normalize(image, image, 0, 255, NORM_MINMAX);
    image.convertTo(image, CV_8U);
    ss_img << "senoide-" << SIDE << ".png";
    imwrite(ss_img.str(), image);

    cout << "Matriz da imagem png... " << endl;
    cout << image << endl;

    fs.open(ss_yml.str(), FileStorage::READ);
    fs["mat"] >> image;

    normalize(image, image, 0, 255, NORM_MINMAX);
    image.convertTo(image, CV_8U);

    imshow("image", image);
    waitKey();

    return 0;
}
```

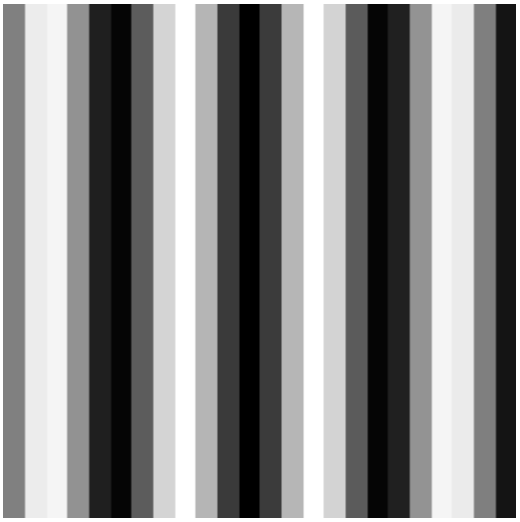


Figure 5. Senoide

3. Decomposição de imagens em planos de bits

A decomposição de imagens em planos de bits é um processo no qual uma imagem digital é dividida em diferentes planos, cada um representando uma determinada quantidade de bits. Essa decomposição permite visualizar a contribuição de cada plano de bits para a formação da imagem final e pode ser útil em várias aplicações, como processamento de imagem, compressão de dados e análise de características visuais. No contexto da decomposição em planos de bits, consideraremos imagens em escala de cinza, onde cada pixel é representado por um único valor de intensidade. O

valor de intensidade de um pixel é geralmente representado por um número binário, que é composto por uma sequência de bits. O número de bits utilizados para representar a intensidade de cada pixel determina a quantidade de níveis de cinza disponíveis na imagem.

3.1 Esteganografia em imagens digitais

A esteganografia em imagens digitais é uma técnica que envolve esconder informações ou dados dentro de uma imagem digital de forma imperceptível aos olhos humanos. É uma maneira de ocultar uma mensagem dentro de outra imagem, conhecida como imagem de cobertura, de modo que a presença da mensagem oculta não seja facilmente detectada. Existem várias abordagens para realizar esteganografia em imagens digitais. Uma das técnicas mais comuns é a substituição do bit menos significativo (LSB - Least Significant Bit) dos pixels da imagem de cobertura pelos bits da mensagem que se deseja ocultar. Como o bit menos significativo tem menos influência na representação visual da imagem, a substituição desse bit por informações ocultas geralmente não causa alterações perceptíveis na imagem.

3.1.1 Código & Resultado.

Escreva um programa que recupere a imagem codificada de uma imagem resultante de esteganografia. Lembre-se que os bits menos significativos dos pixels da imagem fornecida deverão compor os bits mais significativos dos pixels da imagem recuperada. O programa deve receber como parâmetros de linha de comando o nome da imagem resultante da esteganografia.

decode.cpp

```
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;

int main() {
    Mat imagemCodificada, imagemPortadora, imagemEscondida;
    Vec3b valCod, valPort, valEsc;
    int nbits = 3;
    char diretorio[1000];

    // Recebendo a imagem;
    do {
        cout << "Digite a localizacao da imagem codificada: " << endl;
        cout << "EX.: C:\\User\\Desktop\\<nomedaimagem>.<png ou jpeg>" << endl;
        cin >> diretorio;

        imagemCodificada = imread(diretorio, IMREAD_COLOR);

        if (!imagemCodificada.data) {
            cout << "Imagem nao encontrada!" << endl;
            cout << "Verifique o endereco digitado." << endl;
        }
```

```
    } while (!imagemCodificada.data);

    // Clonando;
    imagemEscondida = imagemCodificada.clone();
    imagemPortadora = imagemCodificada.clone();

    // Realiazando a decodificação da imagem;
    for (int i = 0; i < imagemCodificada.rows; i++) {
        for (int j = 0; j < imagemCodificada.cols; j++) {
            valCod = imagemCodificada.at<Vec3b>(i, j);

            valEsc[0] = valCod[0] << (8 - nbits);
            valEsc[1] = valCod[1] << (8 - nbits);
            valEsc[2] = valCod[2] << (8 - nbits);

            imagemEscondida.at<Vec3b>(i, j) = valEsc;

            valPort[0] = valCod[0] >> nbits << nbits;
            valPort[1] = valCod[1] >> nbits << nbits;
            valPort[2] = valCod[2] >> nbits << nbits;

            imagemPortadora.at<Vec3b>(i, j) = valPort;
        }
    }

    imwrite("imagemEscondida.png", imagemEscondida);
    imwrite("imagemPortadora.png", imagemPortadora);

    return 0;
}
```

Figure 6. Imagem codificada

Figure 7. Imagem escondida

4. Preenchendo regiões

O preenchimento de regiões em processamento digital de imagens refere-se a técnicas utilizadas para preencher áreas vazias ou ausentes em uma imagem, com o objetivo de restaurar ou completar informações perdidas. Essas regiões podem ser buracos, objetos removidos ou áreas

danificadas na imagem original. É importante mencionar que o resultado do preenchimento de regiões depende da natureza da área a ser preenchida e da qualidade dos dados disponíveis na imagem original. Em algumas situações, pode ser necessário usar técnicas mais avançadas ou até mesmo combinar várias abordagens para obter resultados satisfatórios.

Além disso, é importante ressaltar que o preenchimento de regiões em uma imagem pode introduzir informações artificiais ou imprecisas, especialmente em áreas complexas ou com texturas irregulares. Portanto, é essencial avaliar cuidadosamente os resultados e, se necessário, realizar ajustes manuais ou refinamentos adicionais para obter uma restauração adequada da imagem.

4.1 FloodFill

O algoritmo Flood Fill (preenchimento por inundação) é uma técnica utilizada em processamento digital de imagens para preencher uma região contígua com uma cor ou padrão específico. O objetivo é identificar todos os pixels conectados a partir de um ponto inicial e atribuir-lhes a cor desejada.

O algoritmo Flood Fill é amplamente utilizado em aplicações como edição de imagens, remoção de fundo, segmentação de objetos e detecção de contornos. No entanto, é importante considerar que a eficiência do algoritmo pode variar dependendo do tamanho da região a ser preenchida e da complexidade da imagem. Em casos de regiões muito grandes ou com muitos detalhes, outras técnicas mais avançadas podem ser necessárias para obter resultados precisos e eficientes.

4.1.1 Código & Resultado.

É possível verificar que caso existam mais de 255 objetos na cena, o processo de rotulação poderá ficar comprometido. Identifique a situação em que isso ocorre e proponha uma solução para este problema. Aprimore o algoritmo de contagem apresentado para identificar regiões com ou sem buracos internos que existam na cena. Assuma que objetos com mais de um buraco podem existir. Inclua suporte no seu algoritmo para não contar bolhas que tocam as bordas da imagem. Não se pode presumir, a priori, que elas tenham buracos ou não.

labeling.cpp

```
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main() {
    cout << "INICIANDO O PROGRAMA..." << endl;
    // INICIALIZANDO VÁRIAVEIS ...
    char diretorio[1000];
    Mat image;
    int cols, rows, bburacos = 0, bolhas=0;
    Point p;
```

```
    // RECEBENDO IMAGEM ...
    do {
        cout << "Digite a localizacao da imagem: " << endl;
        cout << "EX.: C:\\User\\Desktop\\<nomedaimagem>.<png ou jpeg>" << endl;
        cin >> diretorio;

        image = imread(diretorio, IMREAD_GRAYSCALE);

        if (!image.data) {
            cout << "Imagem nao encontrada!" << endl;
            cout << "Verifique se a escrita esta correta." << endl;
        }
    } while (!image.data);

    imshow("janelaOriginal", image);

    cols = image.cols;
    rows = image.rows;
    p.x = 0;
    p.y = 0;

    // REMOVENDO AS BOLHAS LOCALIZADAS NAS BORDAS ...
    cout << "Removendo as bolhas localizadas nas bordas..." << endl;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (image.at<uchar>(i, j) == 255) {
                if (i == 0 || j == 0 || i == (rows - 1) || j == (cols - 1)) {
                    p.x = j;
                    p.y = i;
                    floodFill(image, p, 0);
                }
            }
        }
    }

    p.x = 0;
    p.y = 0;
    floodFill(image, p, 200);

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (image.at<uchar>(i, j) == 255) {
                p.x = j;
                p.y = i;
                bolhas++;
                floodFill(image, p, 30);
            }
        }
    }
```

```
    cout << "Operacao finalizada ... " << endl;
    imshow("JanelaSBolhas", image);
    imwrite("JanelaSBolhas.png", image);
    waitKey();

    // CONTANDO QUANTAS BOLHAS TEM BURACOS...
    cout << "Contando quantas bolhas tem buraco..." << endl;

    for (int i = 0; i < rows; i++) {
        for (int j = 0; j < cols; j++) {
            if (image.at<uchar>(i, j) == 0) {
                if (image.at<uchar>(i - 1, j) != 200) {
                    bburacos++;
                    p.x = j;
                    p.y = i;
                    floodFill(image, p, 200);
                }
            }
        }
    }

    cout << "Operacao finalizada... " << endl;
    imshow("labeling", image);
    imwrite("labeling.png", image);

    cout << "total de bolhas com buracos: " << bburacos << endl;
    cout << "total de bolhas sem buracos: " << bolhas - bburacos << endl;
    cout << "total de bolhas: " << bolhas << endl;
    waitKey();

    return 0;
}
```

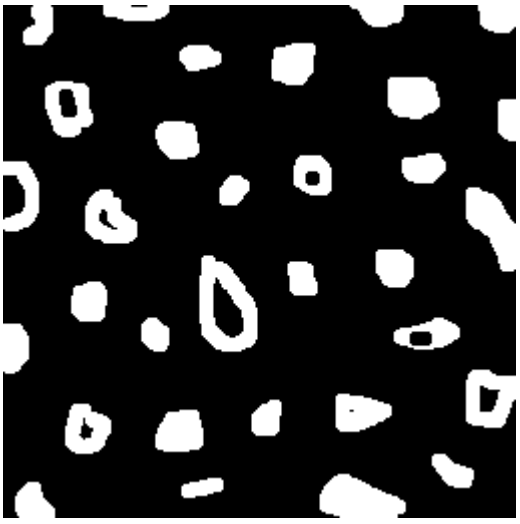


Figure 8. Bolhas

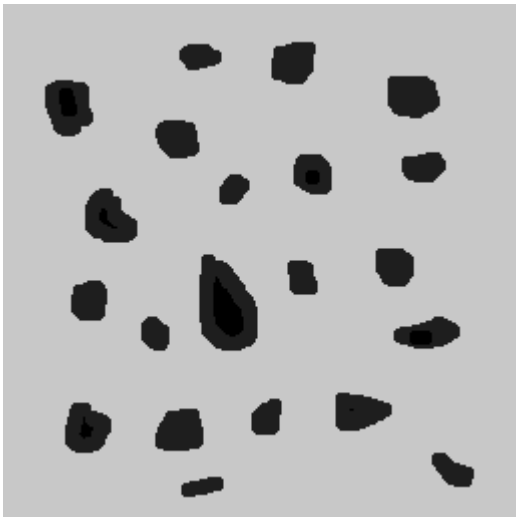


Figure 9. Removendo bolhas das bordas

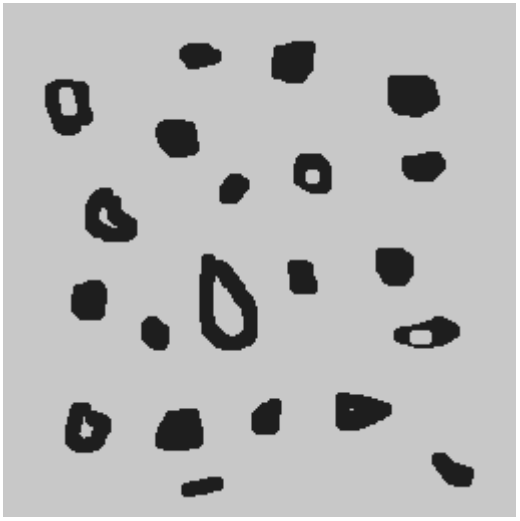


Figure 10. Labeling aplicado

```
total de bolhas com buracos: 7
total de bolhas sem buracos: 14
total de bolhas: 21
```

Figure 11. Resultado pós processamento

5. Manipulação de histogramas

A manipulação de histogramas é uma técnica utilizada em processamento digital de imagens para alterar o contraste, brilho ou distribuição tonal de uma imagem, com base na análise e modificação do seu histograma.

O histograma de uma imagem é uma representação gráfica da distribuição de intensidades dos pixels ao longo de uma escala de tons. Ele mostra a frequência de ocorrência de cada valor de intensidade na imagem.

A manipulação de histogramas pode ser aplicada em várias áreas, como melhoria de qualidade de imagem, correção de iluminação, segmentação de objetos e detecção de características. É uma técnica poderosa para ajustar e realçar informações em uma imagem com base na análise da

distribuição tonal dos pixels.

5.1 Histograma

O histograma é uma ferramenta fundamental para a análise e processamento de sinais. Ele fornece informações importantes sobre a distribuição dos dados e pode revelar características como o valor médio, variação, assimetria e presença de picos ou ruído.

Existem técnicas e algoritmos avançados que podem ser aplicados, dependendo das necessidades específicas do processamento de sinal. O histograma é uma ferramenta poderosa para analisar e manipular dados de sinal, ajudando a extrair informações importantes e melhorar a qualidade e a compreensão dos sinais.

5.1.1 Código & Resultado

implemente um programa equalize.cpp. Este deverá, para cada imagem capturada, realizar a equalização do histogram antes de exibir a imagem. Teste sua implementação apontando a câmera para ambientes com iluminações variadas e observando o efeito gerado. Assuma que as imagens processadas serão em tons de cinza.

equalize.cpp

```
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

int main(int argc, char** argv) {

    Mat imagemOriginal, imagemCinza, imagemEqualizada;
    char diretorio[1000];

    // Carregar a imagem
    cout << "Digite a localizacao da imagem: " << endl;
    cout << "EX.: C:\\User\\Desktop\\<nomedaimagem>.<png ou jpeg>" << endl;
    cin >> diretorio;

    imagemOriginal = imread(diretorio);

    if (imagemOriginal.empty()) {
        cout << "Não foi possível carregar a imagem" << endl;
        return -1;
    }

    // Converter para escala de cinza
    cvtColor(imagemOriginal, imagemCinza, COLOR_BGR2GRAY);

    // Equalizar o histograma
    equalizeHist(imagemCinza, imagemEqualizada);
```

```
    imshow("Imagem em Escala de Cinza", imagemCinza);
    imwrite("ImagememEscaladeCinza.png", imagemCinza);
    imshow("Imagem Equalizada", imagemEqualizada);
    imwrite("ImagemEqualizada.png", imagemEqualizada);
    waitKey(0);

    return 0;
}
```

Figure 12. Biel original

Figure 13. Biel em escala de cinza

Figure 14. Biel equalizado

6. Filtragem no domínio espacial I.

A filtragem no domínio espacial refere-se à aplicação de um filtro direto aos pixels de uma imagem, no próprio domínio espacial da imagem. Nesse tipo de filtragem, cada pixel é processado individualmente, sem levar em consideração a estrutura ou as características globais da imagem.

Existem vários tipos de filtros que podem ser aplicados no domínio espacial, e eles têm diferentes efeitos na imagem. Alguns exemplos comuns são: filtro de suavização, filtro de realce e filtro de nitidez. Esses são apenas alguns exemplos de filtros que podem ser aplicados no domínio espacial. A escolha do filtro depende das características específicas da imagem e do objetivo desejado. A filtragem no domínio espacial é uma técnica amplamente utilizada no processamento de imagens e possui uma variedade de aplicações, incluindo restauração de imagens, detecção de bordas e redução de ruído.

6.1 Laplaciano do Gaussiano.

O Laplaciano do Gaussiano (LoG) é um filtro espacial que combina as propriedades do filtro gaussiano e do operador laplaciano para detecção de bordas e características pontuais em uma imagem. A combinação do filtro gaussiano e do operador laplaciano no Laplaciano do Gaussiano permite que o filtro seja menos sensível ao ruído e mais eficaz na detecção de bordas do que o operador laplaciano aplicado diretamente à imagem original.

O Laplaciano do Gaussiano é uma técnica amplamente utilizada no processamento de imagens, especialmente em aplicações que envolvem a detecção de bordas e características pontuais.

6.1.1 Código & Resultado

Implemente um programa laplgauss.cpp. O programa deverá acrescentar mais uma funcionalidade ao exemplo fornecido, permitindo que seja calculado o laplaciano do gaussiano das imagens capturadas. Compare o resultado desse filtro com a simples aplicação do filtro laplaciano.

laplgauss.cpp

```
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace std;
using namespace cv;

void printmask(Mat &m) {
    for (int i = 0; i < m.size().height; i++) {
        for (int j = 0; j < m.size().width; j++) {
            cout << m.at<float>(i, j) << ",";
        }
        cout << "\n";
    }
}

int main() {
    float media[] = { 0.1111, 0.1111, 0.1111, 0.1111, 0.1111,
                    0.1111, 0.1111, 0.1111, 0.1111 };
    float gauss[] = { 0.0625, 0.125,  0.0625, 0.125, 0.25,
                      0.125,  0.0625, 0.125,  0.0625 };
    float horizontal[] = { -1, 0, 1, -2, 0, 2, -1, 0, 1 };
    float vertical[] = { -1, -2, -1, 0, 0, 0, 1, 2, 1 };
    float laplacian[] = { 0, -1, 0, -1, 4, -1, 0, -1, 0 };
    float boost[] = { 0, -1, 0, -1, 5.2, -1, 0, -1, 0 };
    float laplacianOfGaussian[] = { 0,0,1,0,0,
                                    0,1,2,1,0,
                                    1,2,-16,2,1,
                                    0,1,2,1,0,
                                    0,0,1,0,0 };
    char diretorio[1000], key;
    Mat imagem, framegray, frame32f, frameFiltered,
        mask(3, 3, CV_32F), result;
    int absolut;

    do {
        cout << "Digite a localizacao da imagem: " << endl;
        cout << "EX.: C:\\User\\Desktop\\<nomedaimagem>.<png ou jpeg>" << endl;
        cin >> diretorio;

        imagem = imread(diretorio);

        if (!imagem.data) {
            cout << "Imagem nao encontrada!" << endl;
            cout << "Verifique se a escrita esta correta." << endl;
        }
    } while (!imagem.data);

    cvtColor(imagem, framegray, COLOR_BGR2GRAY);
    imshow("original", framegray);
```

```
    framegray.convertTo(frame32f, CV_32F);

    mask = Mat(3, 3, CV_32F, media);
    absolut = 1;

    for (;;) {
        filter2D(frame32f, frameFiltered, frame32f.depth(), mask, Point(1, 1), 0);

        if (absolut) {
            Mat frameAbs;
            absdiff(frameFiltered, Scalar(0), frameAbs);
            frameFiltered = frameAbs;
        }

        frameFiltered.convertTo(result, CV_8U);
        imshow("filtroespacial", result);

        key = (char)waitKey(0);
        if (key == 27) break;
        switch (key) {
        case 'a':
            absolut = !absolut;
            break;
        case 'm':
            mask = Mat(3, 3, CV_32F, media);
            printmask(mask);
            break;
        case 'g':
            mask = cv::Mat(3, 3, CV_32F, gauss);
            printmask(mask);
            break;
        case 'l':
            mask = cv::Mat(3, 3, CV_32F, laplacian);
            printmask(mask);
            imwrite("laplacian.png", result);
            break;
        case 'o':
            mask = cv::Mat(5, 5, CV_32F, laplacianOfGaussian);
            printmask(mask);
            imwrite("laplacianofgaussian.png", result);
            break;
        default:
            break;
        }
    }

    return 0;
}
```

Figure 15. Biel original

Figure 16. Biel com filtro espacial

Figure 17. Biel com filtro laplaciano

Figure 18. Biel com filtro laplciano do gaussiano

6.2 TiltShift

O efeito Tilt-Shift é uma técnica popular de pós-processamento de imagens que cria a ilusão de miniaturização em uma foto, fazendo com que a cena pareça uma maquete ou um modelo em escala reduzida. O nome "Tilt-Shift" refere-se aos movimentos de inclinação (tilt) e deslocamento (shift) que são usados em câmeras de grande formato para controlar a profundidade de campo e a perspectiva.

Na fotografia tradicional, a miniaturização é alcançada através do uso de uma lente de deslocamento (shift lens) que permite controlar a área de foco seletivamente e criar uma profundidade de campo extremamente rasa.

6.2.1 Código & Resultado

tiltshift.cpp

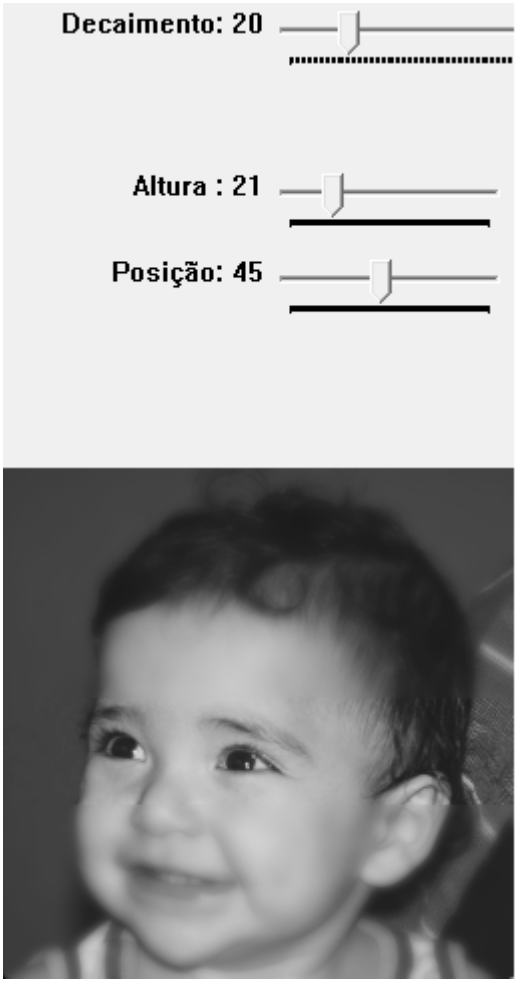
```
#include <iostream>
#include <opencv2/opencv.hpp>

using namespace cv;
using namespace std;

double alfa;
int center_slider = 0;
int center_slider_max = 100;

int alfa_slider = 0;
int alfa_slider_max = 100;

int top_slider = 0;
int top_slider_max = 100;

float media[] = { 1,1,1,1,1,1,1,1,1 };

Mat imagemOriginal, imagemEditada, blended;
```

```
Mat imageTop;

char TrackbarName[50];

void on_trackbar_blend(int, void*) {
    alfa = (double)alfa_slider / alfa_slider_max;
    addWeighted(imagemOriginal, alfa, imageTop, 1 - alfa, 0.0, blended);
    imshow("addweighted", blended);
}

void on_trackbar_change(int, void*) {
    imagemEditada.copyTo(imageTop);
    Size size = imagemEditada.size();
    int width = size.width;
    int height = size.height;
    int limit = top_slider * width / 100;
    int base = center_slider * width / 100;
    if (limit > 0) {
        if (base >= 0 && base <= height - limit) {
            Mat tmp = imagemOriginal(Rect(0, base, width, limit));
            tmp.copyTo(imageTop(Rect(0, base, width, limit)));
        }
        else {
            Mat tmp = imagemOriginal(Rect(0, height - limit, width, limit));
            tmp.copyTo(imageTop(Rect(0, height - limit, width, limit)));
        }
    }
    on_trackbar_blend(alfa_slider, 0);
}

int main() {
    char diretorio[1000];

    do {
        cout << "Digite o endereço da imagem" << endl;
        cout << "C:\\Users\\User\\Desktop\...\\<nomedaimagem>.<extencao>" << endl;
        cin >> diretorio;

        imagemOriginal = imread(diretorio);

        if (!imagemOriginal.data) {
            cout << "Imagem nao encontrada!" << endl;
            cout << "Verifique se a escrita esta correta." << endl;
        }
    } while (!imagemOriginal.data);

    imagemEditada = imagemOriginal.clone();
    Mat aux, mask, mask1;

    mask = Mat(3, 3, CV_32F, media);
    scaleAdd(mask, 1 / 9.0, Mat::zeros(3, 3, CV_32F), mask1);
```

```
    swap(mask, mask1);
    imagemEditada.convertTo(aux, CV_32F);

    for (int i = 0; i < 10; i++) {
        filter2D(aux, aux, aux.depth(), mask, Point(1, 1), 0);
        aux = abs(aux);
        aux.convertTo(imagemEditada, CV_8UC3);
        imagemOriginal.copyTo(imageTop);
    }

    namedWindow("addweighted", 1);

    sprintf_s(TrackbarName, "Decaimento");
    createTrackbar(TrackbarName, "addweighted", &alfa_slider, alfa_slider_max,
on_trackbar_blend);
    on_trackbar_blend(alfa_slider, 0);

    sprintf_s(TrackbarName, "Altura ");
    createTrackbar(TrackbarName, "addweighted", &top_slider, top_slider_max,
on_trackbar_change);
    on_trackbar_change(top_slider, 0);

    sprintf_s(TrackbarName, "Posição");
    createTrackbar(TrackbarName, "addweighted", &center_slider, center_slider_max,
on_trackbar_change);

    imwrite("imagemEditada.png", imagemEditada);

    waitKey(0);
    return 0;
}
```

Figure 19. Biel original

Figure 20. Imagem sem aplicação do filtro



7. A Tranformada Discreta de Fourier

7.1

7.1.1 Código & Resultado

8. Filtragem no Domínio da Frequência

O processamento de imagens no domínio da frequência refere-se às técnicas que envolvem a manipulação das informações de frequência de uma imagem. Isso é feito usando a Transformada de Fourier, especificamente a Transformada Discreta de Fourier (DFT) ou sua variante mais comum, a Transformada Rápida de Fourier (FFT).

O processamento de imagens no domínio da frequência é usado em uma variedade de aplicações, incluindo filtragem de imagens, compressão de imagens, análise de texturas, detecção de bordas e restauração de imagens. Ele permite manipular seletivamente as características de frequência da imagem, explorando as informações de frequência para obter resultados desejados.

8.1 Filtro Homomórfico

O filtro homomórfico é uma técnica de processamento de imagens que combina as propriedades de filtragem espacial e frequencial para melhorar a qualidade de imagens que foram afetadas por iluminação não uniforme ou variações de contraste. Ele é amplamente utilizado em aplicações de processamento de imagens em áreas como visão computacional, análise de imagens e processamento de imagens médicas.

A ideia básica do filtro homomórfico é aplicar uma transformação logarítmica à imagem original para mapear os valores de intensidade da imagem para uma escala mais ampla. Isso ajuda a lidar com as variações de iluminação não uniforme. A transformação logarítmica é seguida pela aplicação de uma Transformada de Fourier para obter a representação da imagem no domínio da frequência. Nesse ponto, é possível realizar uma filtragem no domínio da frequência para separar as componentes de baixa e alta frequência.

O filtro homomórfico utiliza um filtro passa-alta (filtro de nitidez) para realçar os detalhes e uma função de ganho (filtro de correção) para controlar as variações de iluminação. A função de ganho é projetada para atenuar as baixas frequências, que correspondem aos componentes de iluminação, e amplificar as altas frequências, que correspondem aos detalhes e texturas. Depois da filtragem no domínio da frequência, é aplicada a transformada inversa de Fourier para retornar a imagem ao domínio espacial. A imagem resultante é uma versão melhorada da imagem original, com a correção das variações de iluminação e realce dos detalhes.

8.1.1 Código & Resultado

Implemente o filtro homomórfico para melhorar imagens com iluminação irregular. Crie uma cena mal iluminada e ajuste os parâmetros do filtro homomórfico para corrigir a iluminação da melhor forma possível. Assuma que a imagem fornecida é em tons de cinza.

homomorfico.cpp

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <opencv2/imgproc/imgproc.hpp>
```

```
using namespace cv;
using namespace std;

void reciveVal(int** x, int** x2) {

    *x = new int;
    *x2 = new int;
}

void setVal(int* gl, int* gh, int* c, int* d0) {

    *gl = 10;
    *gh = 20;
    *c  = 60;
    *d0 = 70;
}

//  Funcoes para direcionar ponteiros
void on_gammaL (int, void*){}
void on_gammaH (int, void*){}
void on_c      (int, void*){}
void on_d0     (int, void*){}


void deslDFT(Mat& image) {

    Mat temp, A, B, C, D;

    image = image(Rect(0, 0, image.cols & -2, image.rows & -2));
    int cols = image.cols / 2;
    int rows = image.rows / 2;

    // trocar regiões
    A = image(Rect(0, 0, cols, rows));
    B = image(Rect(cols, 0, cols, rows));
    C = image(Rect(0, rows, cols, rows));
    D = image(Rect(cols, rows, cols, rows));

    A.copyTo(temp);
    D.copyTo(A);
    temp.copyTo(D);

    B.copyTo(temp);
    C.copyTo(B);
    temp.copyTo(C);
}

void filterHomomo(Mat temp, int* gl, int* gh, int* c,
                        int* d0, int dft_M, int dft_N) {

    float gl_temp, gh_temp, c_temp, d0_temp, aux, aux2;
```

```
    gl_temp = *gl / 10;
    gh_temp = *gh / 10;
    c_temp  = *c / 10;
    d0_temp = *d0 / 10;

    for (int i = 0; i < temp.rows; i++) {
        for (int j = 0; j < temp.cols; j++) {

            aux  = (i - dft_M / 2) * (i - dft_M / 2)
                 + (j - dft_N / 2) * (j - dft_N / 2);
            aux2 = (1.0 - (float)exp(-(c_temp * aux / (d0_temp * d0_temp))));
            temp.at<float>(i, j) = (gh_temp - gl_temp)
                * aux2 + gl_temp;
        }
    }
}

int main() {

    Mat imaginaryInput, imageComplex, multsp,
        padded, filter, mag;
    Mat image, grayimage, temp;
    Mat_<float> realInput, zeros;
    vector<Mat> planos;
    int* gl, * gh, * c, * d0;
    int dft_M, dft_N;
    char diretorio[1000];

    cout << "Digite o endereço da imagem" << endl;
    cin >> diretorio;
    image = imread(diretorio, IMREAD_GRAYSCALE);

    reciveVal(&gl, &gh);
    reciveVal(&c , &d0);
    setVal(gl, gh, c, d0);

    imshow("Original", image);

    dft_M = getOptimalDFTSize(image.rows);
    dft_N = getOptimalDFTSize(image.cols);

    copyMakeBorder(image, padded, 0, dft_M - image.rows,
                   0, dft_N - image.cols, BORDER_CONSTANT,
                   Scalar::all(0));

    zeros = Mat_<float>::zeros(padded.size());
    imageComplex = Mat(padded.size(), CV_32FC2, Scalar(0));
    filter = imageComplex.clone();
    temp = Mat(dft_M, dft_N, CV_32F);
    namedWindow("Filtro", 1);
```

```
    createTrackbar("c", "Filtro", c, 100, on_c);
    createTrackbar("d0 ", "Filtro", d0, 100, on_d0);
    createTrackbar("gammaH", "Filtro", gh, 100, on_gammaH);
    createTrackbar("gammaL", "Filtro", gl, 100, on_gammaL);

    while (1) {

        on_c(*c, 0);
        on_d0(*d0, 0);
        on_gammaH(*gh, 0);
        on_gammaL(*gl, 0);

        copyMakeBorder(image, padded, 0, dft_M - image.rows,
            0, dft_N - image.cols, BORDER_CONSTANT,
            Scalar::all(0));

        planos.clear();
        realInput = Mat_<float>(padded);
        planos.push_back(realInput);
        planos.push_back(zeros);

        merge(planos, imageComplex);
        dft(imageComplex, imageComplex);
        deslDFT(imageComplex);
        filterHomomo(temp, gl, gh, c, d0, dft_M, dft_N);

        Mat comps[] = { temp, temp };
        merge(comps, 2, filter);
        mulSpectrums(imageComplex, filter, imageComplex, 0);

        deslDFT(imageComplex);
        idft(imageComplex, imageComplex);
        planos.clear();
        split(imageComplex, planos);
        normalize(planos[0], planos[0], 0, 1, NORM_MINMAX);
        imshow("Filtro Homomo", planos[0]);

        if (waitKey(10) == 27) break;

    }
    return 0;

}
```

Figure 21. Biel original

Figure 22. Biel com o filtro aplicado

9. Detecção de bordas com o algoritmo de Canny

O algoritmo de Canny é uma técnica clássica e amplamente utilizada para detecção de bordas em imagens. Ele foi desenvolvido por John F. Canny em 1986 e é conhecido por sua precisão e robustez na detecção de bordas, minimizando a resposta a ruídos e fornecendo bordas bem definidas.

O algoritmo de Canny envolve os seguintes passos principais: redução do ruído, cálculo do gradiente, supressão não máxima, limiarização por histerese e rastreamento de bordas. O resultado final do algoritmo de Canny é uma imagem binária com as bordas claramente destacadas. O algoritmo oferece uma detecção precisa e robusta de bordas, com capacidade de lidar com ruídos e fornecer bordas bem definidas.

9.1 Canny & Pontilhismo

O algoritmo de Canny e o pontilhismo são conceitos distintos, mas é possível combinar essas técnicas para criar um efeito artístico interessante.

O algoritmo de Canny, como explicado anteriormente, é usado para a detecção de bordas em imagens. Ele identifica as transições abruptas de intensidade na imagem e destaca essas áreas como bordas. O resultado é uma imagem binária com bordas bem definidas. Por outro lado, o pontilhismo é uma técnica artística em que a imagem é criada a partir de pequenos pontos ou pontos de cor. Esses pontos, quando vistos em conjunto, formam a imagem final. O pontilhismo é inspirado na forma como o olho humano percebe a mistura de cores quando vários pontos são visualizados de certa distância.

A combinação do algoritmo de Canny e o pontilhismo pode ser alcançada aplicando-se a técnica do pontilhismo nas bordas detectadas pelo algoritmo de Canny. Em vez de usar pontos uniformes para representar a imagem inteira, os pontos são colocados apenas nas bordas encontradas. Essa abordagem pode resultar em uma representação artística da imagem, onde as bordas são enfatizadas pelos pontos, enquanto outras áreas da imagem podem permanecer mais suaves ou com um estilo diferente.

9.1.1 Código & Resultado

Implemente um programa cannypoints.cpp. A idéia é usar as bordas produzidas pelo algoritmo de Canny para melhorar a qualidade da imagem pontilhista gerada. A forma como a informação de borda será usada é livre.

cannypoints.cpp

```
#include <iostream>
#include <opencv2/opencv.hpp>
#include <fstream>
#include <iomanip>
#include <vector>
#include <algorithm>
#include <numeric>
#include <ctime>
#include <cstdlib>

using namespace std;
using namespace cv;

vector<int> showCenter(vector<int> x, int step) {

    for (uint i = 0; i < x.size(); i++) {
        x[i] = x[i] * step + step / 2;
    }
    return x;
}

void funcCircle(Mat image, Mat imageArt, Vec3b color, int r, int x, int y) {

    color = image.at<Vec3b>(x, y);
    circle(imageArt, Point(y, x), r, CV_RGB(color[2], color[1], color[0]), -1,
LINE_AA);
}
```

```
int main() {
    vector<int> cols, rows;
    Mat imagem, fps, points, gray, ofusc, borders;
    int x, y, fator = 60, passo = 5, jitter = 3, raio = 3;
    Vec3b cor;
    char diretorio[1000];

    cout << "Digite o endereço da imagem: " << endl;
    cin >> diretorio;

    imagem = imread(diretorio);
    Mat imagemArte(imagem.rows, imagem.cols, imagem.type());
    srand(time(0));

    if (!imagem.data) {
        cout << "nao abriu" << endl;
        exit(0);
    }

    rows.resize(imagem.rows / passo);
    cols.resize(imagem.cols / passo);
    iota(rows.begin(), rows.end(), 0);
    iota(cols.begin(), cols.end(), 0);
    rows = showCenter(rows, passo);
    cols = showCenter(cols, passo);

    random_shuffle(rows.begin(), rows.end());
    for (auto i : rows) {

        random_shuffle(cols.begin(), cols.end());
        for (auto j : cols) {

            x = i + rand() % (2 * jitter) - jitter + 1;
            y = j + rand() % (2 * jitter) - jitter + 1;
            funcCircle(imagem, imagemArte, cor, raio, x, y);
        }
    }
    imwrite("points.jpg", imagemArte);

    cvtColor(imagemArte, gray, COLOR_BGR2GRAY);
    GaussianBlur(gray, ofusc, Size(5, 5), 25, 25);
    Canny(ofusc, borders, fator, 3 * fator);
    imwrite("canny.jpg", borders);

    for (int i = 0; i < imagemArte.rows; i++) {
        for (int j = 0; j < imagemArte.cols; j++) {
            if (borders.at<uchar>(i, j) == 255) {
                funcCircle(imagemArte, imagemArte, cor, 2, i, j);
            }
        }
```

```
    }

    imwrite("arte.jpg", imagemArte);
    imshow("arte", imagemArte);
    waitKey();
    return 0;
}
```

Figure 23. Biel original

Figure 24. Biel com points

Figure 25. Biel com points 2

Figure 26. Biel com filtro canny

Figure 27. Resultado final do código

10. Quantização vetorial com k-means

A quantização vetorial com o algoritmo k-means é uma técnica usada para reduzir a quantidade de cores em uma imagem. Ela agrupa os pixels da imagem em um número específico de clusters com

base em sua similaridade de cor e, em seguida, substitui os valores de cor dos pixels pelos valores médios dos clusters aos quais pertencem.

O algoritmo k-means é um algoritmo de aprendizado não supervisionado que visa particionar um conjunto de dados em k grupos distintos. No contexto da quantização vetorial de imagens, os pixels são tratados como pontos de dados no espaço de cores e o objetivo é agrupá-los em k clusters representativos.

A quantização vetorial com k-means é eficaz para reduzir o número de cores em uma imagem, o que pode ser útil para economizar espaço de armazenamento ou simplificar a análise de imagens. No entanto, é importante lembrar que a redução de cores pode resultar em perda de detalhes e qualidade visual, dependendo da aplicação e da quantidade de cores escolhida. Portanto, é necessário encontrar um equilíbrio entre a redução de cores e a preservação da qualidade visual desejada.

10.1 K-means

É importante mencionar que o resultado do k-means depende da escolha inicial dos centroides. Como o algoritmo pode convergir para mínimos locais, diferentes inicializações podem resultar em diferentes soluções. Portanto, para obter uma solução mais robusta, geralmente é necessário executar o algoritmo várias vezes com diferentes inicializações e escolher a melhor solução com base em algum critério, como a soma das distâncias dos pontos aos centroides.

10.1.1 Código & Resultado

Implemente um programa exemplo onde a execução do código se dê usando o parâmetro nRodadas=1 e inciar os centros de forma aleatória usando o parâmetro KMEANS_RANDOM_CENTERS ao invés de KMEANS_PP_CENTERS. Realize 10 rodadas diferentes do algoritmo e compare as imagens produzidas.

kmeansrandom.cpp

```
#include <opencv2/opencv.hpp>
#include <cstdlib>
#include <string>
#include <iostream>

using namespace std;
using namespace cv;

int main() {

    int nCluster = 6, nRodadas = 1, cont = 1;
    Mat rotulos, centros, imagem;
    char diretorio[1000];

    cout << "Digite o endereço da imagem: " << endl;
    cin >> diretorio;
```

```
    imagem = imread(diretorio, IMREAD_COLOR);
    Mat samples(imagem.rows * imagem.cols, 3, CV_32F);

    while (cont <= 10) {
        for (int y = 0; y < imagem.rows; y++) {
            for (int x = 0; x < imagem.cols; x++) {
                for (int z = 0; z < 3; z++) {
                    samples.at<float>(y + x * imagem.rows, z) = imagem.at<Vec3b>(y,
x)[z];
                }
            }
        }

        kmeans(samples, nCluster, rotulos,
            TermCriteria(TermCriteria::MAX_ITER | TermCriteria::EPS, 10000, 0.0001),
            nRodadas, KMEANS_RANDOM_CENTERS,centros);

        Mat rotulada(imagem.size(), imagem.type());
        for (int y = 0; y < imagem.rows; y++) {
            for (int x = 0; x < imagem.cols; x++) {
                int indice = rotulos.at<int>(y + x * imagem.rows, 0);
                rotulada.at<cv::Vec3b>(y, x)[0] = (uchar)centros.at<float>(indice, 0);
                rotulada.at<cv::Vec3b>(y, x)[1] = (uchar)centros.at<float>(indice, 1);
                rotulada.at<cv::Vec3b>(y, x)[2] = (uchar)centros.at<float>(indice, 2);
            }
        }

        stringstream ss;
        ss << "saida" << cont << ".jpg";
        string s = ss.str();
        cont++;
        imshow("clustered image", rotulada);
        imwrite(s, rotulada);
        waitKey(1);

    }

    return 0;
}
```

Figure 28. Biel original

[gif] | *gif.gif*

Figure 29. Resultado pós código