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 image0 = 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;

}
```

[biel] | *https://github.com/KempesBarbosa/KempesBarbosa.github.io/blob/main/imagem/biel.png*

[janelaNegativo] |
https://github.com/KempesBarbosa/KempesBarbosa.github.io/blob/main/imagem/janelaNegativo.png

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;
}
```

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;
}
```

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;
}
```

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;
}
```

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;
}
```

6. Filtragem no domínio espacial I

6.1 Filtro Espacial

6.1.1 Código & Resultado

7. Filtragem no domínio espacial II

7.1 TiltShift

7.1.1 Código & Resultado

8. A Tranformada Discreta de Fourier

8.1

8.1.1 Código & Resultado

9. Filtragem no Domínio da Frequência

9.1 Filtro Homomórfico

9.1.1 Código & Resultado

10. Detecção de bordas com o algoritmo de Canny

10.1 Canny & Pontilhismo

10.1.1 Código & Resultado

11. Quantização vetorial com k-means

11.1 K-means

11.1.1 Código & Resultado