Universidade Federal do Rio Grande do Norte

DCA0445 - Processamento Digital de Imagens

link para github: https://github.com/mateusArnaudGoldbarg/PDI

Exercícios práticos da 2° unidade

Componentes:
Eduardo Andre Cordeiro Diogo
Mateus Arnaud Santos de Sousa Goldbarg

Natal - RN
09/2021

7.2 - Utilizando o programa exemplos/dft.cpp como referência, 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. Assima que a imagem fornecida é em tons de cinza.

O filtro homomórfico pode ser utilizado para melhorar imagens prejudicadas por má iluminação. Tomando como base o prgrama fornecido pelo professor (dft.cpp) foi escrito o programa abaixo.

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

#define RADIUS 50

using namespace cv;
using namespace std;

int gl_slider = 0;
int gl_slider_max = 100;

int gh_slider = 50;
int gh_slider_max = 100;

int c_slider = 5;
int c_slider_max = 100;

int d0_slider = 5;
int d0_slider_max = 200;

Mat input_img;
Mat padded;
Mat_<float> realInput, zeros;
Mat complexImage;
Mat filter, tmp;
vector<Mat> planos;

int dft_M, dft_N;

void deslocaDFT(Mat& image);
Mat create_homomorfic_filter(Size paddedSize, double gl, double
gh, double c, double d0);
void on_trackbar_move(int, void*);

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

    input_img = imread("C:\\Users\\mateu\\Downloads\\piano.jpeg",
CV_LOAD_IMAGE_GRAYSCALE);
    resize(input_img, input_img, Size(800, 600));
```

```
    imshow("Original", input_img);

    // identifica os tamanhos otimos para
    // calculo do FFT
    dft_M = getOptimalDFTSize(input_img.rows);
    dft_N = getOptimalDFTSize(input_img.cols);

    // realiza o padding da imagem
    copyMakeBorder(input_img, padded, 0,
        dft_M - input_img.rows, 0,
        dft_N - input_img.cols,
        BORDER_CONSTANT, Scalar::all(0));

    zeros = Mat_<float>::zeros(padded.size());

    char TrackbarName[50];


    namedWindow("a", WINDOW_NORMAL);


    sprintf_s(TrackbarName, "Gamma L x %d", gl_slider_max);
    createTrackbar(TrackbarName, "a", &gl_slider, gl_slider_max,
on_trackbar_move);

    sprintf_s(TrackbarName, "Gamma H x %d", gh_slider_max);
    createTrackbar(TrackbarName, "a", &gh_slider, gh_slider_max,
on_trackbar_move);

    sprintf_s(TrackbarName, "C x %d", c_slider_max);
    createTrackbar(TrackbarName, "a", &c_slider, c_slider_max,
on_trackbar_move);

    sprintf_s(TrackbarName, "Cutoff Frequency x %d",
d0_slider_max);
    createTrackbar(TrackbarName, "a", &d0_slider, d0_slider_max,
on_trackbar_move);

    on_trackbar_move(0, NULL);



    waitKey(0);
    return 0;
}



//troca os quadrantes da imagem da DFT
void deslocaDFT(Mat& image) {
    Mat tmp, A, B, C, D;

    //se a imagem tiver tamanho impar, recorta a regiao para
    //evitar cópias de tamanho desigual
    image = image(Rect(0, 0, image.cols & -2, image.rows & -2));
```

```
    int cx = image.cols / 2;
    int cy = image.rows / 2;

    //reorganizacao dos quadrantes da transformada
    //A B   ->  D C
    //C D       B A
    A = image(Rect(0, 0, cx, cy));
    B = image(Rect(cx, 0, cx, cy));
    C = image(Rect(0, cy, cx, cy));
    D = image(Rect(cx, cy, cx, cy));

    // A <-> D
    A.copyTo(tmp);  D.copyTo(A);  tmp.copyTo(D);

    // C <-> B
    C.copyTo(tmp);  B.copyTo(C);  tmp.copyTo(B);
}


//cria filtro homomorfico
cv::Mat create_homomorfic_filter(cv::Size paddedSize, double gl,
double gh, double c, double d0) {
    Mat filter = Mat(paddedSize, CV_32FC2, Scalar(0));
    Mat tmp = Mat(dft_M, dft_N, CV_32F);

    for (int i = 0; i < tmp.rows; i++) {
        for (int j = 0; j < tmp.cols; j++) {
            float coef = (i - dft_M / 2) * (i - dft_M / 2) + (j -
dft_N / 2) * (j - dft_N / 2);
            tmp.at<float>(i, j) = (gh - gl) * (1.0 -
(float)exp(-(c * coef / (d0 * d0)))) + gl;
        }
    }

    // cria a matriz com as componentes do filtro e junta
    // ambas em uma matriz multicanal complexa
    Mat comps[] = { tmp, tmp };
    merge(comps, 2, filter);
    return filter;
}

void on_trackbar_move(int, void*) {
    // limpa o array de matrizes que vao compor a
    // imagem complexa
    planos.clear();

    // cria a compoente real e imaginaria (zeros)
    realInput = Mat_<float>(padded);
    //realInput += Scalar::all(1);
    //log(realInput,realInput);

    // insere as duas componentes no array de matrizes
    planos.push_back(realInput);
    planos.push_back(zeros);
```

```
    // combina o array de matrizes em uma unica
    // componente complexa
    // prepara a matriz complexa para ser preenchida
    complexImage = Mat(padded.size(), CV_32FC2, Scalar(0));
    merge(planos, complexImage);

    // calcula o dft
    dft(complexImage, complexImage);

    // realiza a troca de quadrantes
    deslocaDFT(complexImage);
    resize(complexImage, complexImage, padded.size());
    normalize(complexImage, complexImage, 0, 1, CV_MINMAX);

    // aplica o filtro frequencial
    float gl = (float)gl_slider / 100.0;
    float gh = (float)gh_slider / 100.0;
    float d0 = 25.0 * d0_slider / 100.0;
    float c = (float)c_slider / 100.0;

    cout << "gl = " << gl << endl;
    cout << "gh = " << gh << endl;
    cout << "d0 = " << d0 << endl;
    cout << "c = " << c << endl;

    Mat filter = create_homomorfic_filter(padded.size(), gl, gh,
c, d0);
    mulSpectrums(complexImage, filter, complexImage, 0);

    // troca novamente os quadrantes
    deslocaDFT(complexImage);

    // calcula a DFT inversa
    idft(complexImage, complexImage);

    // limpa o array de planos
    planos.clear();

    // separa as partes real e imaginaria da
    // imagem filtrada
    split(complexImage, planos);

    // // normaliza a parte real para exibicao
    normalize(planos[0], planos[0], 0, 1, CV_MINMAX);
    resize(planos[0], planos[0], Size(800, 600));
    imshow("Filtro Homomórfico", planos[0]);
    namedWindow("Filtro Homomórfico", WINDOW_NORMAL);
}
```

O resultado da filtragem pode ser visto nas Figuras 1 e 2:

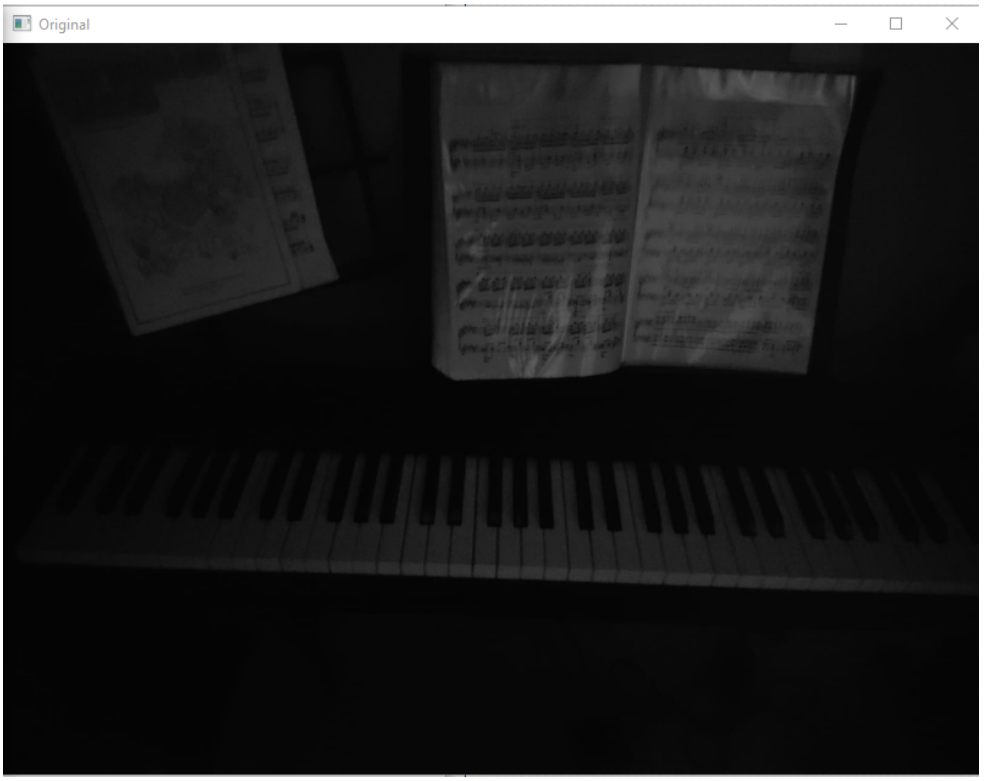

Figura 1 - Imagem Original

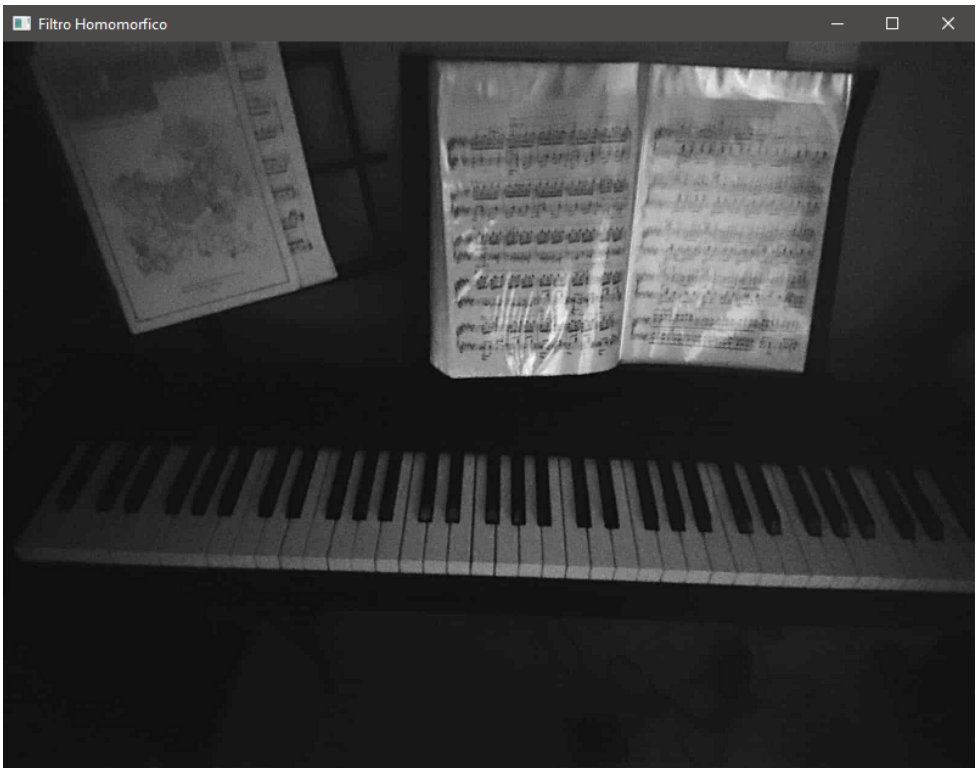

Figura 2 - Imagem Filtrada

É possível notar que mais detalhes do piano podem ser visualizados, as teclas mais graves e mais agudas podem ser vistas e até a partitura está mais nítida.

8.3 - Utilizando os programas exemplo/canny.cpp e exemplos/pontilhismo.cpp como referência, implemente um programa cannypoints.cpp; A idéia é usar as bordas produzidas pelo algoritmo Canny para melhorar a qualidade da imagem pontilhista gerada. A forma como a informação de borda será usada é livre. Entretanto, são apresentadas algumas sugestões de técnicas que poderiam ser utilizadas:

- **Desenhar pontos grandes na imagem pontilhista básica;**
- **Usar a posição dos pixels de borda encontrados pelo algoritmo de Canny para desenhar pontos nos respectivos locais na imagem gerada.**
- **Experimente ir aumentando os limiares do algoritmo de Canny e, para cada novo par de limiares, desenhar círculos cada vez menores nas posições encontradas. A Figura 19 foi desenvolvida usando essa técnica.**
- **Escolha uma imagem de seu gosto e aplique a técnica que você desenvolveu.**
- **Descreva no seu relatório detalhes do procedimento usado para criar sua técnica pontilhista.**

Foi realizada a união dos dois algoritmos fornecidos pelo professor. O resultado pode ser visualizado nas figuras 3, 4 e 5.

```
#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;

#define STEP 5
#define JITTER 3
#define RAIO 3

int top_slider = 10;
```

```
int top_slider_max = 200;
int step_slider = 5;
int step_slider_max = 10;
int jitter_slider = 3;
int jitter_slider_max = 10;
int raio_slider = 3;
int raio_slider_max = 10;
int width, height;

char TrackbarName[50];

Mat image, border, points;


void pointillism(int, void*) {
    vector<int> yrange;
    vector<int> xrange;

    int width, height, gray;
    int x, y;

    width = image.size().width;
    height = image.size().height;

    xrange.resize(height / STEP);
    yrange.resize(width / STEP);

    iota(xrange.begin(), xrange.end(), 0);
    iota(yrange.begin(), yrange.end(), 0);

    for (uint i = 0; i < xrange.size(); i++) {
        xrange[i] = xrange[i] * STEP + STEP / 2;
    }

    for (uint i = 0; i < yrange.size(); i++) {
        yrange[i] = yrange[i] * STEP + STEP / 2;
    }

    points = Mat(height, width, CV_8U, Scalar(255));

    random_shuffle(xrange.begin(), xrange.end());

    for (auto i : xrange) {
        random_shuffle(yrange.begin(), yrange.end());
        for (auto j : yrange) {
            x = i + rand() % (2 * JITTER) - JITTER + 1;
            y = j + rand() % (2 * JITTER) - JITTER + 1;
            gray = image.at<uchar>(x, y);
            circle(points,
                cv::Point(y, x),
                RAIO,
                CV_RGB(gray, gray, gray),
                -1,
                LINE_AA);
        }
```

```
    }

    vector<vector<Point> > contours;
    vector<Vec4i> hierarchy;
    for (int i = 0; i < int(contours.size()); i++) {
        for (int j = 0; j < int(contours[i].size()); j++) {
            gray = image.at<uchar>(contours[i][j].y,
contours[i][j].x);
            circle(points, cv::Point(contours[i][j].x,
contours[i][j].y),
                1,
                CV_RGB(gray, gray, gray),
                -1,
                LINE_AA);
        }
    }

    imshow("cannypoints", points);
    imwrite("cannypoints.png", points);
}

void on_trackbar_canny(int, void*) {
    Canny(image, border, top_slider, 3 * top_slider);
    pointillism(top_slider, 0);
}

int main(int argc, char** argv) {
    image = imread("we.jpeg", IMREAD_GRAYSCALE);
    resize(image,image,Size(600,800));
    width = image.size().width;
    height = image.size().height;

    namedWindow("canny", 1);

    sprintf(TrackbarName, "Threshold inferior");
    createTrackbar(TrackbarName, "canny",
        &top_slider,
        top_slider_max,
        on_trackbar_canny);

    sprintf(TrackbarName, "step");
    createTrackbar(TrackbarName, "canny",
        &step_slider,
        step_slider_max,
        pointillism);

    sprintf(TrackbarName, "jitter");
    createTrackbar(TrackbarName, "canny",
        &jitter_slider,
        jitter_slider_max,
        pointillism);

    sprintf(TrackbarName, "raio");
    createTrackbar(TrackbarName, "canny",
        &raio_slider,
```

```
        raio_slider_max,
        pointillism);

    on_trackbar_canny(top_slider, 0);

    waitKey();
    imwrite("original.png", image);
    imwrite("cannyborders.png", border);
    return 0;
}
```

Figura 3 - Imagem Original

Figura 4 - threshold inferior = 10

Figura 5 - threshold inferior = 102

Figura 6 - Cannypoints

9.2 - Utilizando o programa kmeans.cpp como exemplo prepare 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. Explique porque elas podem diferir tanto.

Foi adicionado um loop mais externo para a que os centróides do algoritmo k-means fosse inicializado 10 vezes em locais randômicos. O algorítmo editado pode ser visto abaixo:

```
#include <opencv2/opencv.hpp>
#include <cstdlib>
#include <string>
using namespace cv;

int main(int argc, char** argv) {
	int nClusters = 8;
	Mat rotulos;
	int nRodadas = 1;
	Mat centros;

	Mat img = imread("clown.jpg", IMREAD_COLOR);
	Mat samples(img.rows * img.cols, 3, CV_32F);

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

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

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

	for (int y = 0; y < img.rows; y++) {
		for (int x = 0; x < img.cols; x++) {
			int indice = rotulos.at<int>(y + x * img.rows,
0);
			rotulada.at<Vec3b>(y, x)[0] =
(uchar)centros.at<float>(indice, 0);
			rotulada.at<Vec3b>(y, x)[1] =
(uchar)centros.at<float>(indice, 1);
			rotulada.at<Vec3b>(y, x)[2] =
```

```
(uchar)centros.at<float>(indice, 2);
            }
      }
      std::string idx = std::to_string(i);
      std::string asd = "clutered";
      asd.append(idx);
      asd.append(".jpg");
      std::cout << asd <<std::endl;

      imshow("clustered image", rotulada);
      imwrite(asd, rotulada);

      }

}
```

A imagem de entrada é apresentada na figura 6, e após isso a tabela mostra as 10 imagens resultantes. Ao se observar as imagens, pode-se perceber pequenas mudanças quando comparadas. Isso se deve ao parâmetro KMEANS_RANDOM_CENTERS, que inicializa os centros do algorítmo em posições aleatórias.

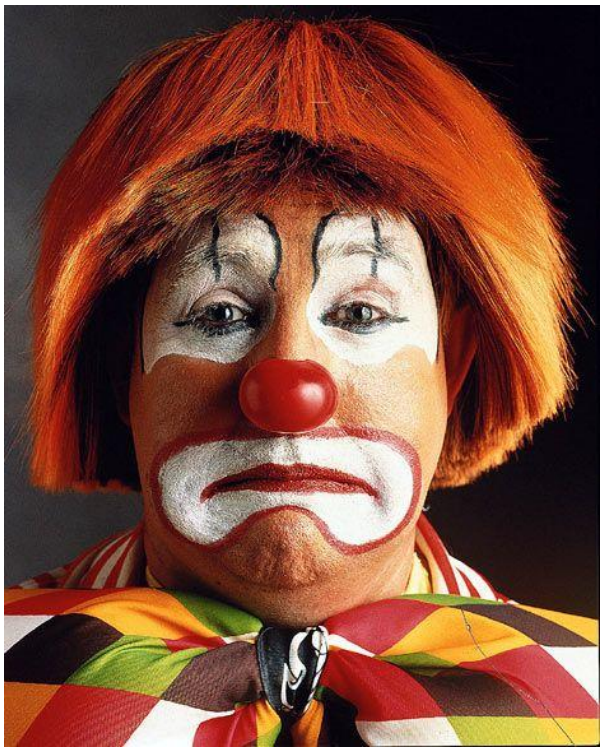

Figura 7 - Entrada para o algoritmo k-means