Relatório dos exercícios propostos

Aluno: Mateus Lacerda Soares

Unidade 2

Todos os códigos e imagens encontram-se em https://github.com/MateusLacerda/PDI

9. Filtragem no domínio da frequência

Exercício 1

O objetivo do programa *filtro_homomorfico.cpp* é melhorar imagens com iluminação não adequada. Para isso, utilizamos o programa *dft.cpp* como base e implementamos um filtro homomórfico cuja função de transferência é dada por:

$$H(u,v)=(\gamma_h-\gamma_l)\left(1-e^{\left(-c*\left(\frac{D^2(u,v)}{D_0^2}\right)\right)}\right)+\gamma_l$$

Equação 1: Função de transferência do filtro homomórfico

Basicamente o que fizemos foi criar uma função para criar a equação do filtro e aplicar no espectro de frequência da imagem.

Também foi criado um *scrollbar* para ser definido os parâmetros do filtro. Segue o código fonte do programa:

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

using namespace cv;
using namespace std;

double gama_h;
int gama_h_slider = 0;
int gama_h_slider_max = 100;

double gama_l;
int gama_l_slider = 0;
int gama_l_slider_max = 100;

double c;
int c_slider = 0;
int c_slider_max = 100;

double d0;
int d0_slider = 0;
int d0_slider_max = 100;

char TrackbarName[50];
```

```
void on_trackbar_gama_h(int, void*){
 gama_h = (double) gama_h_slider ;
}

void on_trackbar_gama_l(int, void*){
 gama_l = (double) gama_l_slider  ;
}

void on_trackbar_c(int, void*){
 c = (double) c_slider  ;
}

void on_trackbar_d0(int, void*){
 d0 = (double) d0_slider ;
}

Mat ObterFiltroHomomorfico(int M, int N){
  Mat H = Mat(M, N, CV_32F), filter;
  for(int i=0; i<M; i++){
    for(int j=0; j<N; j++){
                H.at<float>  (i,j)  =  (gama_h-gama_l)*(1.0-exp(-1.0*(float)c*((((float)i-M/2.0)*((float)i-M/2.0)  +  ((float)j-
N/2.0)*((float)j-N/2.0))/(d0*d0))))+ gama_l;
    }
  }
  Mat comps[]= {H, H};
  merge(comps, 2, filter);
  return filter;
}

// 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;

  // reorganiza os 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);
}

int main(int , char** argv){
  VideoCapture cap;
  Mat imaginaryInput, complexImage, multsp;
  Mat padded, filter, mag;
  Mat image, imagegray, tmp;
  Mat_<float> realInput, zeros;
  vector<Mat> planos;
  char key;
```

```
namedWindow("original",1);

sprintf( TrackbarName, "Gama H: ");
  createTrackbar( TrackbarName, "original",
          &gama_h_slider,
          gama_h_slider_max,
          on_trackbar_gama_h );
  on_trackbar_gama_h(gama_h_slider, 0 );

sprintf( TrackbarName, "Gama L: ");
  createTrackbar( TrackbarName, "original",
          &gama_l_slider,
          gama_l_slider_max,
          on_trackbar_gama_l );
  on_trackbar_gama_l(gama_l_slider, 0 );

sprintf( TrackbarName, "C: ");
  createTrackbar( TrackbarName, "original",
          &c_slider,
          c_slider_max,
          on_trackbar_c );
  on_trackbar_c(c_slider, 0 );

sprintf( TrackbarName, "D0: ");
  createTrackbar( TrackbarName, "original",
          &d0_slider,
          d0_slider_max,
          on_trackbar_d0 );
  on_trackbar_d0(d0_slider, 0 );

// valores ideais dos tamanhos da imagem
// para calculo da DFT
int dft_M, dft_N;

image = imread(argv[1]);
// identifica os tamanhos otimos para
// calculo do FFT
dft_M = getOptimalDFTSize(image.rows);
dft_N = getOptimalDFTSize(image.cols);

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

// parte imaginaria da matriz complexa (preenchida com zeros)
zeros = Mat_<float>::zeros(padded.size());

// prepara a matriz complexa para ser preenchida
complexImage = Mat(padded.size(), CV_32FC2, Scalar(0));

// a função de transferência (filtro frequencial) deve ter o
// mesmo tamanho e tipo da matriz complexa
filter = complexImage.clone();

// cria uma matriz temporária para criar as componentes real
// e imaginaria do filtro ideal
tmp = Mat(dft_M, dft_N, CV_32F);



// cria a matriz com as componentes do filtro e junta
```

```
// ambas em uma matriz multicanal complexa
Mat comps[]= {tmp, tmp};
merge(comps, 2, filter);

for(;;){
  //cap >> image;
  image = imread(argv[1]);
  cvtColor(image, imagegray, CV_BGR2GRAY);
  imshow("original", imagegray);

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

  // limpa o array de matrizes que vao compor a
  // imagem complexa
  planos.clear();
  // cria a compoente real
  realInput = Mat_<float>(padded);
  // 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
  merge(planos, complexImage);

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

  // realiza a troca de quadrantes
  deslocaDFT(complexImage);

  filter = ObterFiltroHomomorfico(dft_M, dft_N);

  // aplica o filtro frequencial
  mulSpectrums(complexImage,filter,complexImage,0);

  // limpa o array de planos
  planos.clear();
  // separa as partes real e imaginaria para modifica-las
  split(complexImage, planos);

  // recompoe os planos em uma unica matriz complexa
  merge(planos, complexImage);

  // 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);
```

```
  imshow("filtrada", planos[0]);

  key = (char) waitKey(10);
  if( key == 27 ) break; // esc pressed!
  if (key == 99){
    imwrite("filtrada.png", planos[0]);
    imwrite("original.png", imagegray);
  }
 }
 return 0;
}
```

As figuras abaixo mostram o resultado do programa:

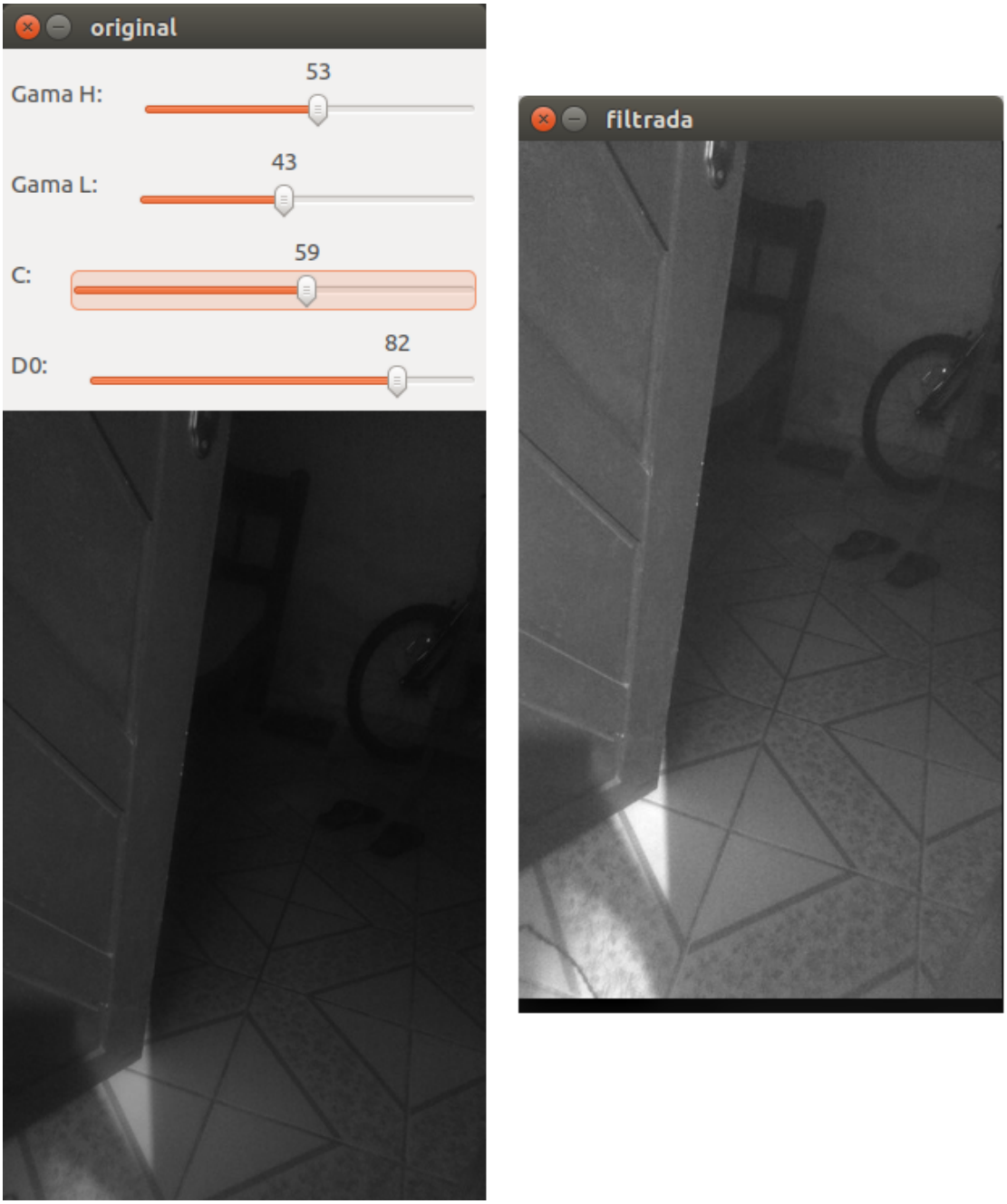

Figura 1: Imagens original e filtrada

10. Canny e a arte com pontilhismo

Exercício 1

Nesse exercício iremos unir os algoritmos de canny e pontilhismo a fim de obter imagens melhores. A ideia é identificar bordas na imagem e criar mais pontos nessas bordas. Para detectar essas bordas será usado o algoritmo de canny, e para o pontilhismo o algoritmo de pontilhismo. Foi usado os programas *canny.cpp* e *pontilhismo.cpp* como base.

Foi feito algumas adaptações para trabalharmos com imagem colorida.

No primeiro momento foi aplicado o pontilhismo na imagem. Depois disso, foi aplicado o detector de bordas da imagem original algumas vezes, sendo que a cada vez o *Threshold* do filtro de canny foi cada vez maior. A cada vez que o algoritmo de canny era aplicado na imagem, a imagem com bordas era varrida e os valores em que o pixel é diferente de zero – ou seja, as bordas – era adicionado pontos na imagem pontilhista com o valor do pixel da imagem original naquela posição. Além disso, a cada iteração o raio dos pontos adicionados na imagem pontilhista ia diminuindo.

Foi usado 5 iterações, com o limiar inferior do filtro de canny indo de 15 a 75 e o limiar superior 3 vezes o inferior. Além disso, o raio das bordas começou em 8 e terminando em 3.

O código abaixo mostra o que fizemos:

```
#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 main(int argc, char** argv){
  vector<int> yrange;
  vector<int> xrange;

  Mat imagem, imagemBorda, imagemPontilhismo;

  int width, height, x, y;

  Vec3b pixel;

  imagem= imread(argv[1],CV_LOAD_IMAGE_COLOR);

  srand(time(0));

  if(!imagem.data){
          cout << "nao abriu" << argv[1] << endl;
    cout << argv[0] << " imagem.jpg";
    exit(0);
  }

  width=imagem.size().width;
```

```
height=imagem.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;
}

imagemPontilhismo = Mat(height, width, CV_8UC3, Scalar(255,255,255));

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;
    pixel = imagem.at<Vec3b>(x,y);
    circle(imagemPontilhismo,
          cv::Point(y,x),
          RAIO,
          cv::Scalar(pixel[0],pixel[1],pixel[2]),
          -1,
          CV_AA);
  }
}

// acrescentar pontos nas bordas usando o filtro de canny

int raio2 = 8;
for (int interacoes = 1; interacoes < 6;interacoes++){
  Canny(imagem, imagemBorda, interacoes*15, interacoes*45);
  raio2 = raio2 - 1;
  // procurar onde há borda, e, onde encontrar, colocar um
  // ponto na imagem com pontilhismo
  for (int i = 0; i < height; ++i){
    for (int j = 0; i < width; ++i){
      if (imagemBorda.at<uchar>(i,j)>0){
        pixel = imagem.at<Vec3b>(x,y);
        circle(imagemPontilhismo,
          cv::Point(y,x),
          raio2,
          cv::Scalar(pixel[0],pixel[1],pixel[2]),
          -1,
          CV_AA);
      }
    }
  }
}
imshow("Imagem com pontilhismo e canny", imagemPontilhismo);
imwrite("cannyPontilhismo.png", imagemPontilhismo);

char key = (char) waitKey();
return 0;
```

```
}
```

Abaixo mostra o resultado do programa aplicado à imagem lena:

Figura 2: Programa cannypoints.cpp aplicado a lena