Relatório de Programação – U2 – Processamento Digital de Imagens

Exercícios de Open CV – Filtro Homomórfico, Técnicas de segmentação, Quantização de Imagens.

Departamento de Computação e Automação – DCA/UFRN
Engenharia da Computação – UFRN – 2018.1
Aluno/Discente – Lukas Maximo Grilo Abreu Jardim
Professor/Docente – Agostinho de Medeiros Brito Júnior

Filtro homomórfico
Um filtro que consiste de 4 parâmetros, e duas dft para melhorar a iluminação de uma imagem:
Imagem utilizada: wwolf.jpg

Código utilizado: homomorfico.cpp

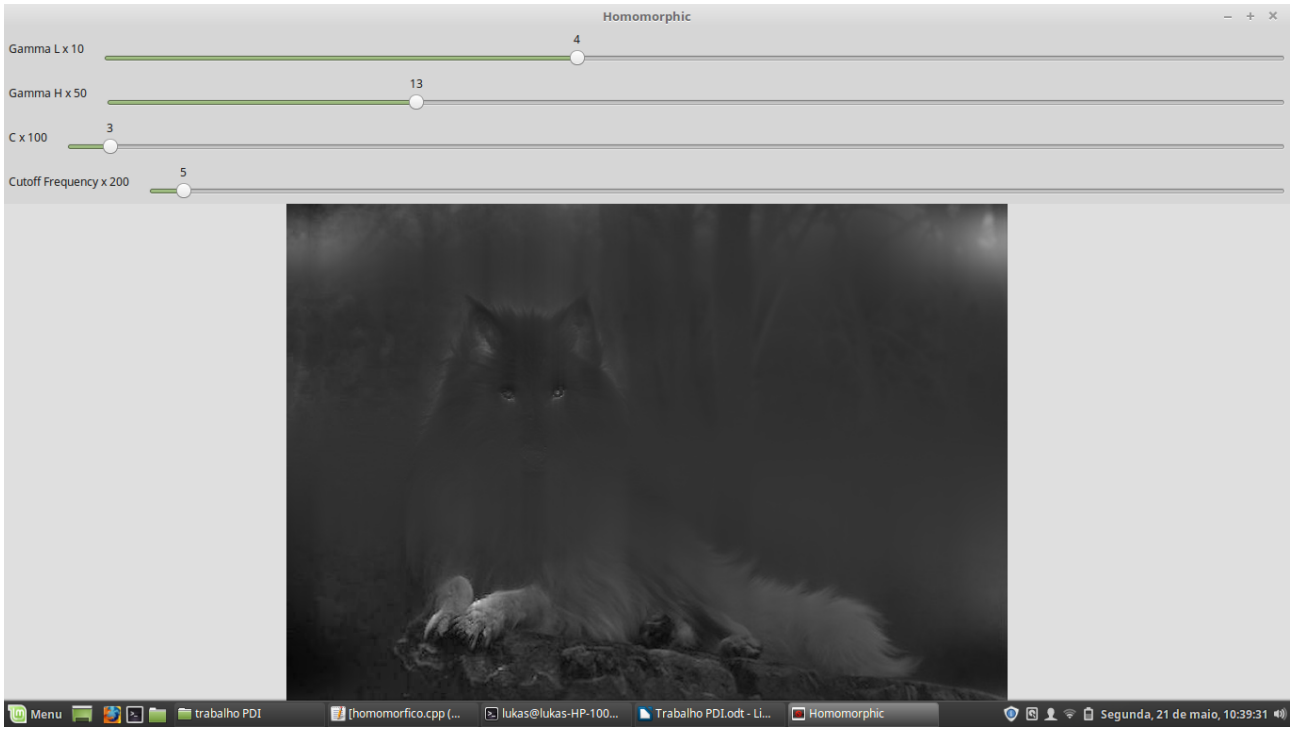
```
#include <iostream>
#include "opencv2/opencv.hpp"

using namespace std;
using namespace cv;

int top_slider = 5;
int top_slider_max = 200;
int gamL_slider = 4, gamH_slider = 20, shrC_slider = 10, cut_slider = 5;
const int gamL_max = 10, gamH_max = 50, shrC_max = 100, cut_max = 200;
int gammaL, gammaH, sharpC, cutoff;
char TrackbarName[50];

Mat image, p1, p2, padded, imFiltered;

int dft1, dft2;
Mat homomorphicFilter(double yl, double yh, double c, double d0){
    Mat filter = Mat(image.size(), CV_32FC2, Scalar(0));
    Mat tmp = Mat(dft1, dft2, CV_32F);

    for(int i=0; i<dft1; i++){
```

```
            for(int j=0; j<dft2; j++){
                //tmp.at<float> (i,j) = (gh - gl)*(1 - exp(-c*(( (i-dft1/2)*(i-dft1/2) + (j-dft2/2)*(j-
dft2/2) ) / (d0*d0) ))) + gl;
                float d2 = pow(i - dft1/2.0, 2) + pow(j - dft2/2.0, 2);
                float exp1 = - c*(d2/pow(d0, 2));
                float valor = (yh - yl)*(1 - expf(exp1) ) + yl;
                tmp.at<float> (i,j) = valor;
            }
        }
        Mat comps[]= {tmp,tmp};
        //imshow("Filter", tmp);
        merge(comps, 2, filter);
        return filter;
}

/*void calcHomomorphicFilter() {
    Mat filter = Mat(padded.size(), CV_32FC2, Scalar(0));
    Mat tmp = Mat(dft1, dft2, CV_32F);

    for (int i = 0; i < dft_M; i++) {
        for (int j = 0; j < dft_N; j++) {
            float d2 = pow(i - dft1/2.0, 2) + pow(j - dft2/2.0, 2);
            float exp1 = - (d2/pow(d0, 2));
            float valor = (yh - yl)*(1 - expf(exp1) ) + yl;
            tmp.at<float> (i,j) = valor;
        }
    }

    Mat comps[] = {tmp, tmp};
    merge(comps, 2, filter);

    Mat dftClone = imageDft.clone();

    mulSpectrums(dftClone,filter,dftClone,0);

    deslocaDFT(dftClone);

    idft(dftClone, dftClone);

    vector<Mat> planos;

    split (dftClone, planos);

    normalize(planos[0], planos[0], 0, 1, CV_MINMAX);

    ...
}*/


void deslocaDFT(Mat& image ){
        Mat tmp, A, B, C, D;
        image = image(Rect(0, 0, image.cols & -2, image.rows & -2));
```

```
	int cx = image.cols/2;
	int cy = image.rows/2;
	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.copyTo(tmp);  D.copyTo(A);  tmp.copyTo(D);
	C.copyTo(tmp);  B.copyTo(C);  tmp.copyTo(B);
}

void applyFilter(void){
	vector<Mat> planos; planos.clear();
	Mat zeros = Mat_<float>::zeros(padded.size());
	Mat realInput = Mat_<float>(padded);
	Mat complex;
	cout<<"running"<<endl;
	realInput += Scalar::all(1);
	log(realInput,realInput);
	//normalize(realInput, realInput, 0, 1, CV_MINMAX);
	//imshow("logimage",realInput);
	planos.push_back(realInput);
	planos.push_back(zeros);
	merge(planos, complex);
	cout<<"running"<<endl;
	dft(complex, complex);
	deslocaDFT(complex);
	resize(complex,complex,padded.size());
	normalize(complex,complex,0,1,CV_MINMAX);
	cout<<"running"<<endl;
	p2 = homomorphicFilter(gammaL,gammaH,sharpC,cutoff);
	cout<<"running"<<endl;
	mulSpectrums(complex,p2,complex,0);
	deslocaDFT(complex);
	idft(complex, complex);
	//normalize(complex, complex, 0, 1, CV_MINMAX);

	planos.clear();
	split(complex, planos);
	exp(planos[0],planos[0]);
	planos[0] -= Scalar::all(1);
	normalize(planos[0], planos[0], 0, 1, CV_MINMAX);
	imFiltered = planos[0].clone();
	imwrite("Homomorphic.jpeg", imFiltered);
}

void on_trackbar(int, void*){
	gammaL = (double) gamL_slider/10;
	gammaH = (double) gamH_slider/10;
	sharpC = (double) shrC_slider;
	cutoff = (double) cut_slider;
	applyFilter();
	imshow("Homomorphic",imFiltered);
```

```
}



int main(int argc, char* argv[]){
	image = imread(argv[1],CV_LOAD_IMAGE_GRAYSCALE);
	namedWindow("opened", WINDOW_NORMAL);
	namedWindow("Filter",WINDOW_NORMAL);

	imshow("opened",image);
	waitKey();
	dft1 = image.rows;
	dft2 = image.cols;
	copyMakeBorder(image, padded, 0, dft1 - image.rows, 0, dft2 - image.cols,
BORDER_CONSTANT, Scalar::all(0));
	imFiltered = padded.clone();
	namedWindow("Homomorphic", WINDOW_NORMAL);
	char TrackbarName[50];

	sprintf( TrackbarName, "Gamma L x %d", gamL_max );
	createTrackbar( TrackbarName, "Homomorphic", &gamL_slider, gamL_max, on_trackbar);

	sprintf( TrackbarName, "Gamma H x %d", gamH_max );
	createTrackbar( TrackbarName, "Homomorphic", &gamH_slider, gamH_max, on_trackbar);

	sprintf( TrackbarName, "C x %d", shrC_max );
	createTrackbar( TrackbarName, "Homomorphic", &shrC_slider, shrC_max, on_trackbar);

	sprintf( TrackbarName, "Cutoff Frequency x %d", cut_max );
	createTrackbar( TrackbarName, "Homomorphic", &cut_slider, cut_max, on_trackbar);
	on_trackbar(0,0);
	waitKey(0);
	//imshow("Filter", p2);
}
```

Resultado Final:



Método de Canny e Pontilhismo

Usando um algorítimo de segmentação é possível melhorar a definição de uma imagem através da detecção de pontos linhas e bordas: por exemplo, utilizando os filtros de detecção de bordas de Canny, em conjunto com um segundo algorítimo para preencher a imagem com pontos coloridos com a mesma tonalidade dos pixels onde cada um se encontra, ou outra\s técnicas de suavização de imagem, como apresentadas no código cannypontilismo.cpp mostrado a seguir:

Imagem utilizada: 9_wolf.jpeg

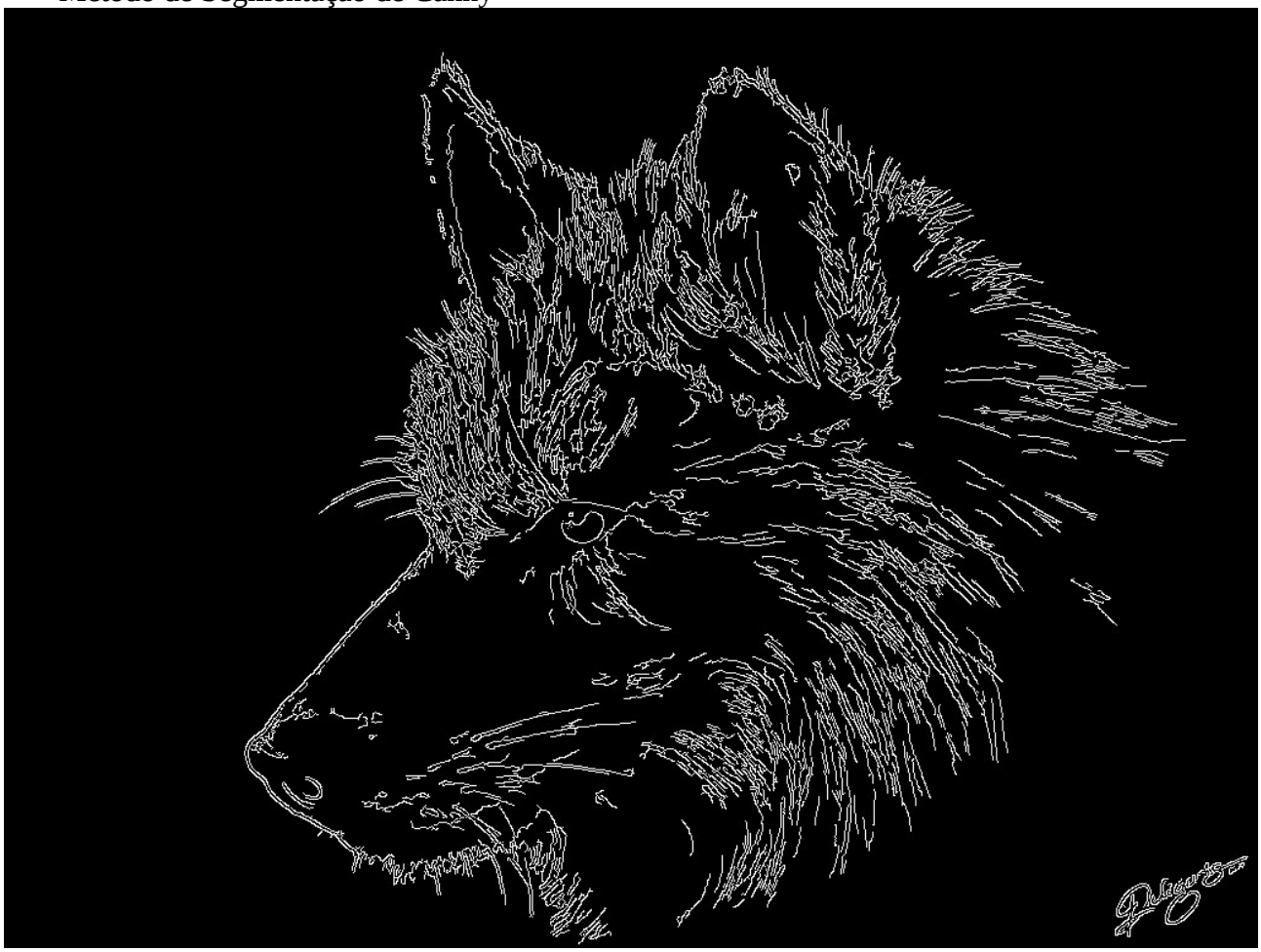
Código utilizado: cannypontilismo.cpp

```
#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;

#define STEP 5
```

```
#define JITTER 3
#define RAIO 3
int top_slider = 5;
int top_slider_max = 200;

char TrackbarName[50];

Mat image, border, frame, points, join;

void on_trackbar_canny(int, void*){
	Canny(image, border, top_slider, 3*top_slider);
	imshow("cannyborders", border);

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

	int width, height;

	image= imread(argv[1],CV_LOAD_IMAGE_GRAYSCALE);

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

	sprintf( TrackbarName, "Threshold inferior", top_slider_max );

	namedWindow("cannyborders",1);

	createTrackbar( TrackbarName, "cannyborders",
		&top_slider,
		top_slider_max,
		on_trackbar_canny );

	on_trackbar_canny(top_slider, 0 );
	waitKey();
	vector<int> yrange;
	vector<int> xrange;

	int gray;
	int x, y;

	srand(time(0));

	if(!image.data){
		cout << "nao abriu" << argv[1] << endl;
		cout << argv[0] << " imagem.jpg";
		exit(0);
	}
	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/4;
    }

    for(uint i=0; i<yrange.size(); i++){
        yrange[i]= yrange[i]*STEP+STEP/4;
    }

    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()%(1*JITTER)-JITTER+1;
            y = j+rand()%(1*JITTER)-JITTER+1;
            gray = image.at<uchar>(x,y);
            circle(points,
                cv::Point(y,x),
                RAIO,
                CV_RGB(gray,gray,gray),
                -1,
                CV_AA);
        }
    }
    join = points + border;

    imwrite("cannypontos.jpg", points);
    imwrite("cannypontos2.jpg", border);
    imwrite("cannypontosjoin.jpg", join);
    return 0;
}
```

- Método de Segmentação de Canny



- Método de Pontilhamento

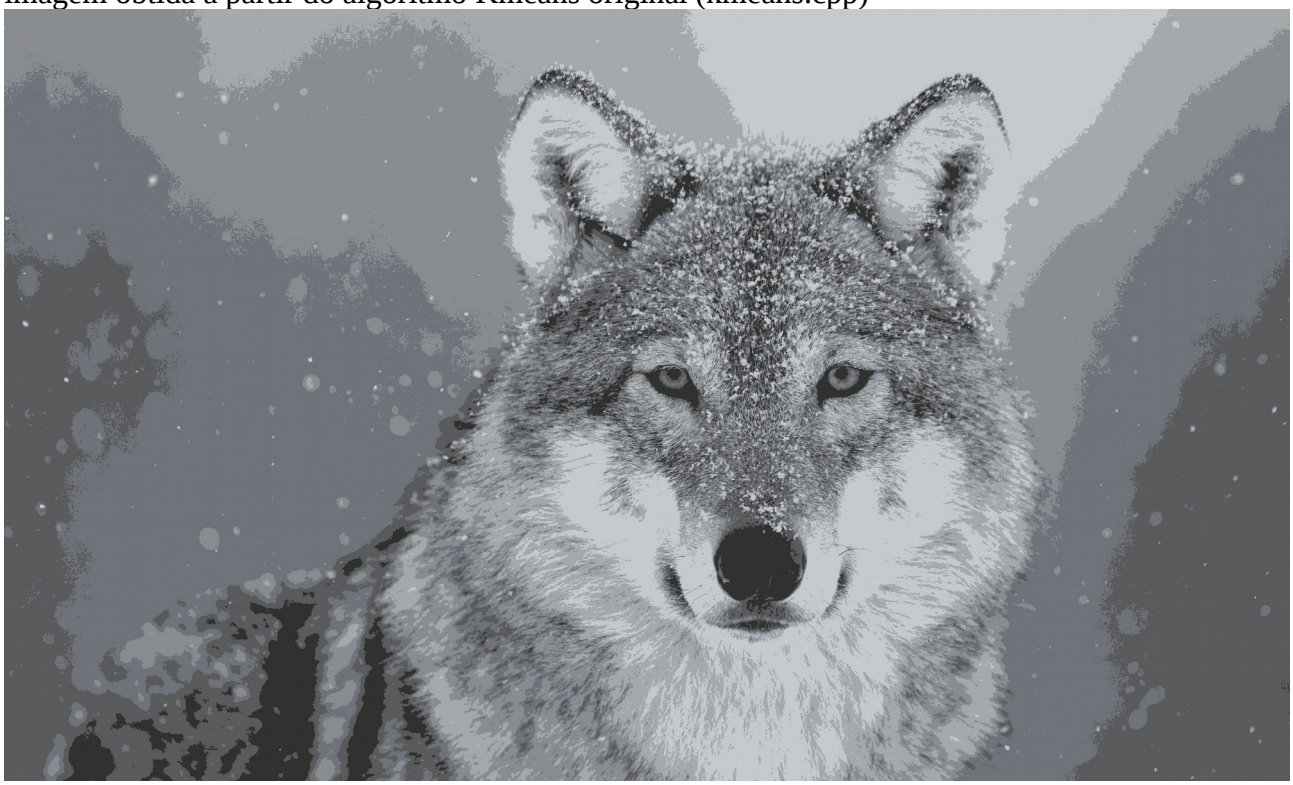
- Junção das imagens (Resultado Final)

Quantização de Imagens (K-Means)

E uma técnica que consiste em obter planos em diferentes faixas de cor ou intensidade a partir de processos de quantização e combiná-los em uma imagem quantizada, como demonstrado a seguir.

Imagem utilizada: wolf.jpg

imagem obtida a partir do algorítmo Kmeans original (kmeans.cpp)



Código kmeandmodif.cpp:

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

using namespace std;
using namespace cv;

int main( int argc, char** argv ){
	int nClusters = 6;
	Mat rotulos;
	int nRodadas = 1;
	Mat centros;

	if(argc!=3){
		exit(0);
	}
	cout<<"estou lendo\n";
	Mat img = imread( argv[1], CV_LOAD_IMAGE_COLOR);
	Mat samples(img.rows * img.cols, 3, CV_32F);

	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];
			}
		}
	}
	cout<<"estou escrevendo\n";
	kmeans(samples,
		nClusters,
		rotulos,
		TermCriteria(CV_TERMCRIT_ITER|CV_TERMCRIT_EPS, 10000, 0.0001),
		nRodadas,
		KMEANS_RANDOM_CENTERS,
		centros );
	namedWindow("clustered_image",1);
	cout<<"estou escrevendo\n";
	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);
		}
	}
	cout<<"estou escrevendo\n";
	imshow( "clustered_image", rotulada );
	imwrite(argv[2], rotulada);
	waitKey( 0 );
```

}

Imagems geradas pela mondificação dos parâmentros solicitados afim de iniciar os centros de quantização de forma aleatória.