Introdução

Este pequeno relatório descreve o trabalho feito para realizar segmentação por limiares.

Primeiro, limiares para separação das classes foram escolhidos manualmente, com auxílio da ferramenta de edição GIMP. Os valores de limiares selecionados são descritos na Tabela 1.

Tabela 1: limiares utilizados para separar os pixels em classes.

	Classe positiva
Dentina	$45 \leq pixel < 119$
Canal	$0 \leq pixel < 4$
Pinos	$144 \leq pixel < 255$

Após a separação por limiar, pixels dentro do intervalo supracitado são considerados como classe positiva (pintados de branco), enquanto pixels fora do intervalo são da classe negativa (pixel preto).

Para cada classe, compara-se o valor predito com o valor real, do ground truth. Gera-se uma nova Figura, onde a cor dos pixels determina os acertos da figura segmentada:

- Pixel colorido (da cor da classe do ground truth): verdadeiro positivo
- Pixel cinza-claro: falso positivo
- Pixel preto: falso negativo
- Pixel branco: verdadeiro negativo

O resultado da execução do algoritmo é descrito na Figura 1. A legenda de cada uma das subfiguras é descrita na Tabela 2. O teste foi realizado para a imagem do dataset *A2 Dentes 01 02 03__rec0684.bmp*.

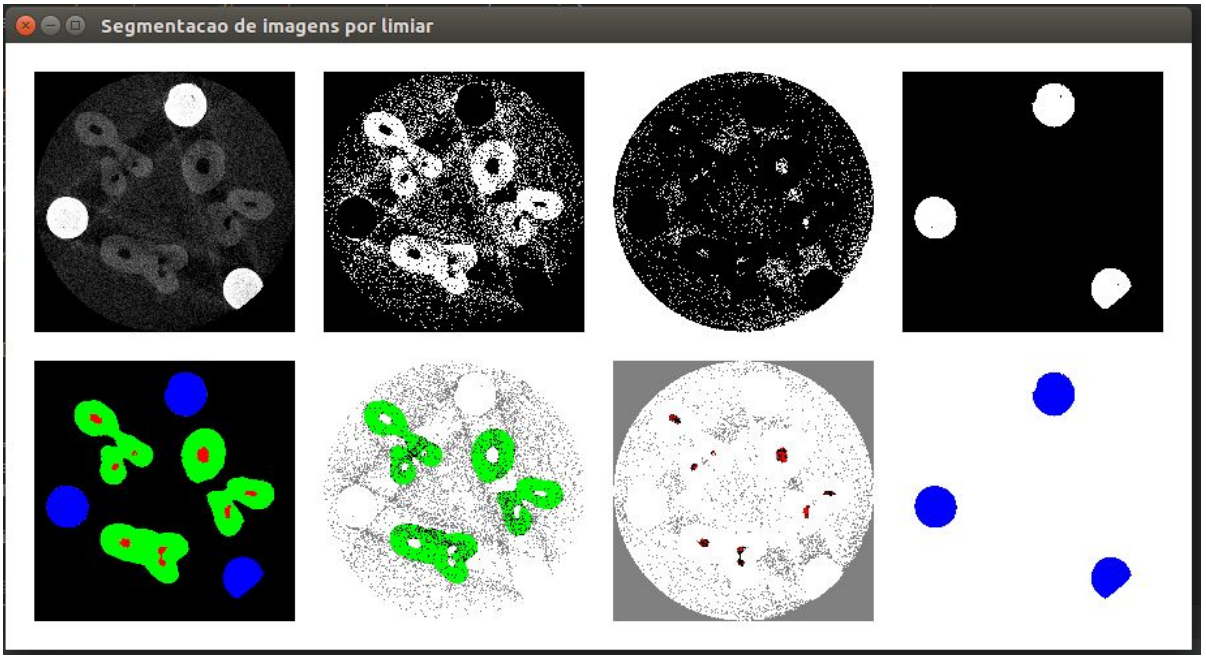


Figura 1: resultado da execução do algoritmo proposto.

Tabela 2: legendas para as subfiguras da Figura 1.

Imagem original	Dentina predito	Canal predito	Pinos predito
Imagem anotada	Dentina vs ground truth	Canal vs ground truth	Pinos vs ground truth

A seguir, são geradas matrizes de confusão para cada uma das classes: dentina, canal e pinos, nas Tabelas 3, 4, e 5, respectivamente.

Tabela 3: matriz de confusão para a classe dentina.

		Classe verdadeira	
		1	0
Classe predita	1	464967	458305
	0	83392	2993336

Tabela 3: matriz de confusão para a classe canal.

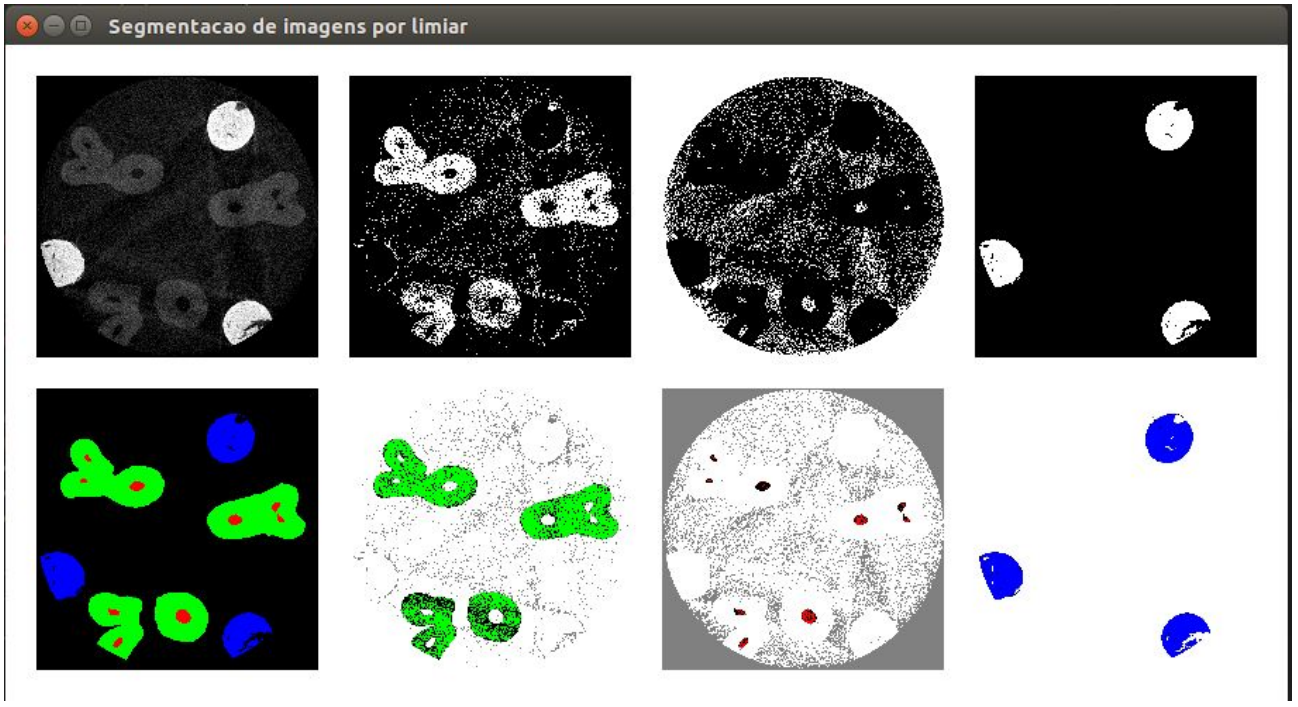
		Classe verdadeira	
		1	0
Classe predita	1	11799	1123562
	0	17940	2846699

Tabela 3: matriz de confusão para a classe pinos.

		Classe verdadeira	
		1	0
Classe predita	1	229947	2262

	0	807	3766984

Outro exemplo da execução do algoritmo é mostrado na Figura 2, executada para a figura do dataset *Dentes 01 02 03_rec0745.bmp*



Código fonte

O código fonte encontra-se em anexo a este documento. As partes mais importantes do código são descritas a seguir.

```
#include <iostream>
#include <cmath>

using namespace std;

#ifdef __APPLE__
#include <GLUT/glut.h>
#endif

#include "ImageClass.h"

ImageClass original, annotated;
ImageClass dentina, canal, pinos;
ImageClass dentinaHit, canalHit, pinosHit;

int n_images = 8;
ImageClass *allImages[8] = {&original, &annotated, &dentina, &canal, &pinos, &dentinaHit,
&canalHit, &pinosHit};

#define LARGURA_JAN 910
#define ALTURA_JAN 466

#define FILTER_SIDE 3

#define X_DIST 22
#define Y_DIST 22
#define Y_PAD 22
#define X_PAD 22
#define ZOOM_LEVEL 0.1

```

```
30. #define N_SHADES 255
31.
32. typedef struct {
33.     unsigned char r;
34.     unsigned char g;
35.     unsigned char b;
36. } color;
```

Algoritmo 1: cabeçalho do código-fonte, com definição de estrutura cor.

```
37. ImageClass postThreshold(ImageClass img, unsigned char lowerThreshold, unsigned char
    upperThreshold) {
38.     ImageClass newImg = ImageClass(img.SizeX(), img.SizeY(), img.Channels());
39.     img.CopyTo(&newImg);
40.     newImg.SetZoomH(img.GetZoomH());
41.     newImg.SetZoomV(img.GetZoomV());
42.
43.     unsigned char r, g, b;
44.     for(int i = 0; i < newImg.SizeX(); i++) {
45.         for(int j = 0; j < newImg.SizeY(); j++) {
46.             newImg.ReadPixel(i, j, r, g, b);
47.             unsigned char gray_pixel = (unsigned char)((0.3 * r) + (0.59 * g) + (0.11 *
    b));
48.             unsigned char new_value;
49.             if((lowerThreshold <= gray_pixel) && (gray_pixel < upperThreshold)) {
50.                 new_value = N_SHADES;
51.             } else {
52.                 new_value = 0;
53.             }
54.             newImg.DrawPixel(i, j, new_value, new_value, new_value);
55.         }
56.     }
57.     return newImg;
58. }
59.
60. bool match(unsigned char r, unsigned char g, unsigned char b, color targetColor) {
61.     return (r == targetColor.r) && (g == targetColor.g) && (b == targetColor.b);
62. }
63.
```

Algoritmo 2: algoritmo utilizado para separar, baseado em limiares, uma imagem baseada em classes. Valores de pixel que ficarem entre lowerThreshold e upperThreshold são a classe positiva, enquanto pixels fora deste intervalo são da classe negativa.

```
64. ImageClass diffMatrix(ImageClass proposed, ImageClass groundTruth, int *confMatrix, color
    targetColor) {
65.     ImageClass output(original.SizeX(), original.SizeY(), original.Channels());
66.     proposed.CopyTo(&output);
67.     output.SetZoomH(proposed.GetZoomH());
68.     output.SetZoomV(proposed.GetZoomV());
69.
70.     unsigned char r0, g0, b0;
71.     unsigned char r1, g1, b1;
72.
73.     // TP, FP, FN, TN
74.     for(int i = 0; i < 4; i++) {
75.         confMatrix[i] = 0;
76.     }
77.
78.     for(int i = 0; i < groundTruth.SizeX(); i++) {
79.         for(int j = 0; j < groundTruth.SizeY(); j++) {
80.             proposed.ReadPixel(i, j, r0, g0, b0);
81.             groundTruth.ReadPixel(i, j, r1, g1, b1);
82.
83.             bool hit = (r0 == 255) && (g0 == 255) && (b0 == 255);
84.
85.             // se estamos em um pixel da classe positiva
86.             if(match(r1, g1, b1, targetColor)) {
87.                 if(hit) {  // se acertou
88.                     output.DrawPixel(i, j, targetColor.r, targetColor.g, targetColor.b);
89.                     confMatrix[0] += 1; // true positive
90.                 } else {  // se errou
91.                     output.DrawPixel(i, j, 0, 0, 0);
92.                     confMatrix[2] += 1;  // false negative
93.                 }
94.             } else {  // estamos em um pixel da classe negativa
95.                 if(!hit) {  // se classificou como sendo negativo
96.                     output.DrawPixel(i, j, 255, 255, 255);
97.                     confMatrix[3] += 1;  // true negative
98.                 } else {  // se errou
99.                     output.DrawPixel(i, j, 128, 128, 128);
100.                        confMatrix[1] += 1;  // false positive
101.                    }
102.                }
103.            }
104.        }
105.        return output;
106.    }
107.
```

Algoritmo 3: algoritmo utilizado para gerar uma nova figura, baseada nas predições encontradas pela separação em limiar, e o ground truth.

```
108.    void init(int argc, char **argv) {
109.        // Carrega a uma imagem
110.
111.        cout << "NOTA: esse programa apresenta problemas carregando uma imagem .bmp com o
    alpha channel." << endl;
112.
113.        if (argc < 3) {
114.            cout << "Usage:" << endl <<
115.                 "\t./threshold_segumentation <original_image_path> <annotated_image_path>"
    << endl;
116.            throw 6;
117.        }
118.
119.        int r1 = original.Load(argv[1]);
120.        int r2 = annotated.Load(argv[2]);
121.
122.        if ((r1 && r2) == 0) {
123.            exit(1); // Erro na carga da imagem
124.        } else {
125.            cout << ("Imagem carregada!\n");
126.        }
127.
128.        color dentinaColor = {.r = 0, .g = 255, .b = 0};
129.        color canalColor = {.r = 255, .g = 0, .b = 0};
130.        color pinosColor = {.r = 0, .g = 0, .b = 255};
131.
132.        original.SetZoomH(ZOOM_LEVEL);
133.        original.SetZoomV(ZOOM_LEVEL);
134.        original.SetPos(X_PAD, Y_PAD + (original.SizeY() * ZOOM_LEVEL + Y_DIST));
135.
136.        annotated.SetZoomH(ZOOM_LEVEL);
137.        annotated.SetZoomV(ZOOM_LEVEL);
138.        annotated.SetPos(X_PAD, Y_PAD);
139.
140.        dentina = postThreshold(original, 45, 119);
141.        dentina.SetPos(X_PAD + (((original.SizeX() * ZOOM_LEVEL) + X_DIST) * 1), Y_PAD +
    (original.SizeY() * ZOOM_LEVEL + Y_DIST));
142.
143.        int confMatrix[4];
144.        dentinaHit = diffMatrix(dentina, annotated, &confMatrix[0], dentinaColor);
145.        dentinaHit.SetPos(X_PAD + (((original.SizeX() * ZOOM_LEVEL) + X_DIST) * 1), Y_PAD);
146.        cout << "confusion Matrix for class dentina:" << endl << endl;
147.        printConfusionMatrix(confMatrix);
148.
149.        canal = postThreshold(original, 0, 4);
150.        canal.SetPos(X_PAD + (((original.SizeX() * ZOOM_LEVEL) + X_DIST) * 2), Y_PAD +
    (original.SizeY() * ZOOM_LEVEL + Y_DIST));
151.        canalHit = diffMatrix(canal, annotated, &confMatrix[0], canalColor);
152.        canalHit.SetPos(X_PAD + (((original.SizeX() * ZOOM_LEVEL) + X_DIST) * 2), Y_PAD);
153.        cout << "confusion Matrix for class canal:" << endl << endl;
154.        printConfusionMatrix(confMatrix);
155.
156.        pinos = postThreshold(original, 144, 255);
157.        pinos.SetPos(X_PAD + (((original.SizeX() * ZOOM_LEVEL) + X_DIST) * 3), Y_PAD +
    (original.SizeY() * ZOOM_LEVEL + Y_DIST));
158.        pinosHit = diffMatrix(pinos, annotated, &confMatrix[0], pinosColor);
159.        pinosHit.SetPos(X_PAD + (((original.SizeX() * ZOOM_LEVEL) + X_DIST) * 3), Y_PAD);
160.        cout << "confusion Matrix for class pinos:" << endl << endl;
161.        printConfusionMatrix(confMatrix);
162.    }
```

Algoritmo 4: Algoritmo utilizado para iniciar as figuras de diferenças e posicionar as figuras na janela do programa.