

MEMORIAL DE PROJETO

Sistema de visão computacional para análise dimensional de fios trefilados

Guilherme Marim da Silva

William Regone

Engenharia Mecânica

Sumário

1. Apresentação	4
1.1 Visão computacional	4
1.2 Trefilação	5
1.3 Tecnologias utilizadas no desenvolvimento do sistema	7
1.3.1 Python	7
1.3.2 OpenCV	7
1.3.3 PIL/Pillow	8
1.3.4 NumPy	8
1.3.5 Flask	8
1.3.6 HTML	8
1.3.7 CSS	9
2. Memorial descritivo do sistema	10
2.1 Introdução	10
2.2 Descrição geral do projeto	10
3. Desenvolvimento do sistema	11
3.1 Aplicação das ferramentas de visão computacional	11
3.1.1 Aquisição da imagem	11
3.1.2 Pré-processamento	11
3.1.2.1 Filtro média	13
3.1.2.2 Filtro Gaussiano	13
3.1.2.3 Filtro mediana	14
3.1.2.4 Filtro bilateral	15
3.1.3 Segmentação	16
3.1.3.1 Binarização	17

3.1.3.2 Operações morfológicas	18
3.1.4 Extração de características.....	21
3.1.5 Reconhecimento de padrões	29
3.2 Páginas HTML e microframework Flask	29
3.3 Validação do sistema.....	29
4. Considerações Finais	33
5. Referências	35
6. Anexos	37
6.1 Funções implementadas no arquivo <i>helpers.py</i>	37
6.2 Integração com o <i>microframework</i> Flask no arquivo <i>views.py</i>	46
6.3 <i>Templates</i> HTML e CSS.....	49
6.4 Telas gráficas	58

1. Apresentação

O presente projeto buscou desenvolver um sistema baseado nas ferramentas de visão computacional para análise dimensional de fios trefilados para controle de qualidade e redução de custos.

Os tópicos a seguir apresentam com mais detalhes o que é a visão computacional e suas ferramentas, o processo de trefilação e as tecnologias utilizadas no desenvolvimento do sistema.

1.1. Visão computacional

De acordo com Ballard & Brown (1982) a visão computacional é definida pela ciência que estuda e desenvolve tecnologias que permitem que máquinas processem imagens. A partir desse processamento é possível então, extrair informações e reconhecer padrões dos objetos de interesse contidos na imagem através de um conjunto hardware e software (BARELLI, 2018).

As ferramentas de visão computacional buscam simular o funcionamento do cérebro humano por meio de modelos matemáticos e algoritmos para receber estímulos através da visão (imagens), processar imagens realizando associações e transmitir essas informações. Com o avanço da tecnologia, esse recente campo da ciência pode nos ajudar com muitas contribuições nas mais distintas áreas da medicina, biologia, física e engenharia, extraíndo informações e reconhecendo padrões que antes levariam muito tempo para serem processados nos meios convencionais.

Os sistemas de visão computacional possuem um fluxo comum à diversas aplicações, em que seguem as etapas de aquisição da imagem, pré-processamento, segmentação, extração de características, reconhecimento de padrões e resultados, conforme exemplificado na Figura 1:



Figura 1. Fluxo da visão computacional.

Fonte: Autor.

1.2. Trefilação

Na conformação mecânica, o processo de trefilação é um dos mais utilizados pela indústria na fabricação de arames, barras cilíndricas, fios e tubos de pequeno diâmetro (BUTTON, 2001; SOARES, 2012).

A trefilação é definida como um processo de fabricação por deformação plástica, em que a matéria-prima ou fio-máquina é tracionado ao passar por uma ferramenta cônica chamada de fieira. Esse processo resulta na redução da área da seção transversal do fio e aumento do comprimento, permanecendo assim o mesmo volume, alterando as características dimensionais da peça (NUNES, 2012; SOARES, 2012; SOUZA, 2011). A operação é comumente realizada a frio, com alteração das propriedades mecânicas do material do fio.

A redução do diâmetro da peça se dá através da deformação plástica permanente do material, causada pela reação de compressão realizada pela ferramenta no metal, enquanto esta sofre a ação da força de tração. Essa compressão é chamada de compressão indireta (ALTAN et al., 1999; HELMAN & CETLIN; 1983; SOUZA, 2011; NUNES, 2012; SOARES, 2012).

Por sua vez, do outro lado da fieira, há um mecanismo responsável por aplicar a força de tração para que ocorra o trabalho de redução de diâmetro. Esse mecanismo é chamado de cabeçote de tração e está exemplificado na Figura 2.

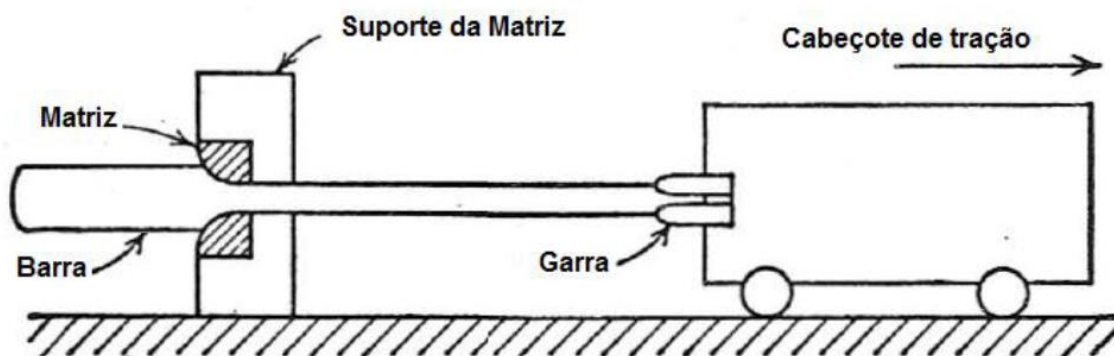


Figura 2. Ilustração esquemática do processo de trefilação.

Fonte: (DIETER, 1981).

As principais vantagens da trefilação com relação a outros processos de conformação são: aumento da resistência à tração e tensão de escoamento do material devido ao encruamento que a peça sofre durante todo o processo (SOARES, 2012; SOUZA, 2011).

Pode-se observar na Figura 3 um “zoom” esquemático do processo de trefilação. O diâmetro inicial (D_i) representa as características da matéria-prima, ou seja, antes da deformação iniciar, que afetam o comportamento do material na zona de deformação e as propriedades do produto obtido. Também são importantes a qualidade superficial e o tratamento de superfície anterior ao processo de conformação. Após a trefilação temos o diâmetro final (D_f), que representa as características do produto conformado, principalmente as propriedades mecânicas e superficiais, e sua qualidade dimensional e geométrica.

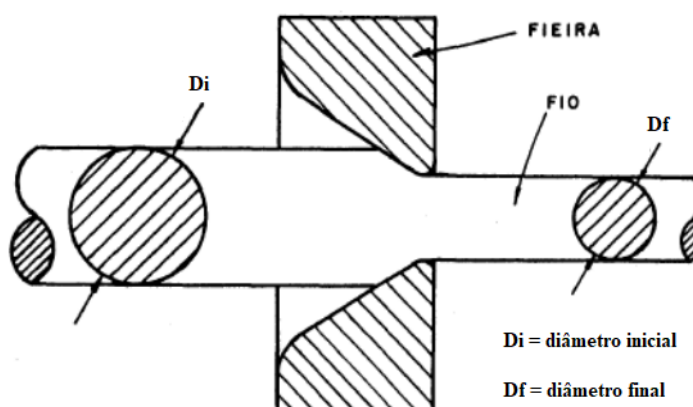


Figura 3. Esquema de fieira e fio sendo trefilado.

Fonte: (BUTTON, 2001).

1.3. Tecnologias utilizadas no desenvolvimento do sistema

Para o desenvolvimento do sistema foram utilizadas as seguintes tecnologias: a linguagem de programação Python em sua terceira versão e as bibliotecas OpenCV, PIL/Pillow e NumPy, a linguagem de marcação HTML junto com as páginas de estilos CSS para criação de páginas Web, e o *microframework* Flask para integrar o código Python com o código HTML.

1.3.1 Python

Criada em 1990 por Guido van Rossum, no Instituto Nacional de Pesquisa para Matemática e Ciência da Computação da Holanda (CWI), Python vem crescendo em várias áreas da ciência e sendo adotada por muitos pesquisadores.

Em pesquisas científicas, o raciocínio na maioria dos casos é complexo. Python trata alguns detalhes importantes na programação, como alocação de memória e gerenciamento de recursos de maneira automática e competente, permitindo que os pesquisadores se concentre exclusivamente com o estudo em questão, e não com detalhes técnicos da linguagem. Todo esse gerenciamento interno possibilita fluidez e agilidade nos trabalhos científicos.

1.3.2 OpenCV

Desenvolvida originalmente pela Intel em 2000, a biblioteca OpenCV foi idealizada para manipular e processar imagens digitais, permitindo a popularização da visão computacional a usuários, programadores e engenheiros das mais diversas áreas (BRADSKI & KAEHLER, 2008).

Essa biblioteca possui código aberto, sendo totalmente livre para o uso comercial e acadêmico, e está dividida em cinco grandes funções que são: processamento de imagens, análise estrutural, análise de movimento e rastreamento de objetos, reconhecimento de padrões e calibração de câmera, e reconstrução 3D (MARENGONI & STRINGHINI, 2009; MORDVINTSEV & RAHMAN K, c2013).

1.3.3 PIL/Pillow

A biblioteca PIL (*Python Imaging Library*) possui métodos e funções para manipulação de imagens e suporta alguns formatos como PNG, TIFF, BMP, EPS e GIF. Posteriormente, foi atualizada para a biblioteca Pillow com o objetivo de suportar a terceira versão da linguagem Python (LUNDH & ELLIS, 2002; LUNDH & CLARK, c2015).

1.3.4 NumPy

NumPy é uma biblioteca usada para realizar cálculos em arranjos/vetores multidimensionais (matrizes). Ela fornece um conjunto de ferramentas para uso em álgebra linear, transformadas de Fourier e números aleatórios (NUMPY DEVELOPERS, c2019).

Imagens são construídas de matrizes bidimensionais, sendo assim, seus recursos são fundamentais para se trabalhar no processamento de imagens, tornando as operações e manipulações envolvendo figuras mais rápidas e eficientes. Sua sintaxe se assemelha muito com o software MatLab®, porém com toda expressividade da linguagem Python.

1.3.5 Flask

Flask é um *microframework* escrito em Python baseados nas bibliotecas WSGI Werkzeug e Jinja2, que possui ferramentas simples para o desenvolvimento de aplicações Web. É chamado de *microframework* porque ele não precisa de várias dependências para funcionar, ou seja, para colocar uma página na Web ou criar uma aplicação. Além disso, ele tem uma forma bem minimalista de utilizar o Python, sem que seja necessário escrever muito código.

1.3.6 HTML

Quando acessamos uma página Web, o *browser* (navegador) é responsável por interpretar o código HTML e renderizá-lo de forma compreensível para o usuário final, exibindo textos, botões, formulários, etc, com as configurações definidas por meio das diversas *tags* que essa linguagem dispõe. HTML é uma das linguagens que utilizamos para desenvolver websites. Seu acrônimo vem do inglês *Hypertext Markup*

Language ou em português Linguagem de Marcação de Hipertexto. Atualmente o HTML encontra-se na versão 5 e é padronizada pelo W3C (*World Wide Web Consortium*).

1.3.7 CSS

CSS é a linguagem que descreve o estilo e formatação que um documento HTML deve ter, ou seja, decora uma página de Web. O CSS separa o conteúdo da representação visual do site. Utilizando o CSS é possível alterar a cor do texto e do fundo, fonte e espaçamento entre parágrafos. Também é possível criar tabelas, usar variações de layouts, ajustar imagens para suas respectivas telas e etc.

2. Memorial descritivo do sistema

2.1. Introdução

Na literatura, as aplicações de visão computacional em processos de conformação mecânica são pouco usuais. Desta forma, este projeto busca apresentar uma ferramenta inovadora que seja capaz de reconhecer e extrair informações dimensionais de fios trefilados, sendo uma alternativa no processo de controle de qualidade, que pode resultar em redução de custo de produção, aumento na precisão das medidas do objeto de estudo e velocidade no controle de qualidade.

Este projeto tem como objetivo aplicar as ferramentas de visão computacional para extração das dimensões de fios trefilados como o diâmetro final (Df). Suas etapas são organizadas nos seguintes objetivos específicos:

- Capturar a imagem de um fio trefilado;
- Realizar o pré-processamento da imagem;
- Aplicar binarização e operações morfológicas;
- Separar os objetos de interesse na figura;
- Extrair informações dimensionais dos objetos segmentados.

2.2. Descrição geral do projeto

Para facilitar a organização do projeto e seguir algumas recomendações de boas práticas da programação, o desenvolvimento do sistema foi dividido em três partes. A primeira parte trata de um arquivo nomeado como *helpers.py*, que possui todas as funções do fluxo comum a sistemas de visão computacional. A segunda parte são os *templates* HTML e CSS que cuidam da identidade visual do sistema. Por último, foi criado um arquivo *views.py* que realiza a integração dos códigos Python com códigos HTML através do *microframework* Flask.

3. Desenvolvimento do sistema

Neste tópico são descritas as três partes que compõem o sistema, sendo elas o fluxo de visão computacional apresentado na Figura 1, os *templates* HTML e CSS, e a integração dos códigos Python com códigos HTML.

3.1. Aplicação das ferramentas de visão computacional

3.1.1 Aquisição da imagem

Nesta etapa devem ser considerados dois importantes elementos para se obter uma imagem com qualidade aprovável, que facilitará nas etapas posteriores. O primeiro elemento trata-se dos equipamentos a serem utilizados no ambiente da captura, como câmeras e sistemas de iluminação. O segundo elemento é o algoritmo que faz a leitura e gerencia às próximas ações a serem realizadas (RUDEK et al., 2001; BARELLI, 2018). O código responsável pelo carregamento da imagem capturada está informado abaixo:

```
import cv2 as cv

def carrega_imagem(file):
    imagem = cv.imread(file)
    return imagem
```

A primeira linha é responsável por importar a biblioteca OpenCV, e a função *carrega_imagem* recebe uma *string* do diretório juntamente com o nome e extensão da figura capturada.

3.1.2 Pré-processamento

Após a aquisição da imagem, com o arquivo digital armazenado e carregado pelo algoritmo, é desempenhada a etapa de pré-processamento para melhorar a qualidade da imagem. É realizado correção de iluminação, contraste, e ruídos com o auxílio de operações aritméticas, geométricas e aplicação de filtros (RUDEK et al., 2001; BRADSKI & KAEHLER, 2008; BARELLI, 2018).

Os ruídos gerados no processo anterior podem ser provenientes de diferentes fontes e os principais fatores são: sensores utilizados na captura da cena, iluminação do ambiente, condições climáticas no momento da aquisição da imagem e a posição relativa entre o objeto de interesse e a câmera.

Para tratar esses ruídos, a biblioteca OpenCV possui diversos filtros que são divididos em filtros espaciais e filtros de frequência. Os filtros espaciais são matrizes que percorrem toda a imagem modificando diretamente os pixels, com a função de corrigir, realçar e suavizar regiões específicas. Essas matrizes são denominadas de máscaras ou núcleos e atuam alterando os valores da intensidade de cinza de cada pixel. O deslocamento da máscara sobre a imagem que modifica os pixels é chamado de convolução (BRADSKI & KAEHLER, 2008; MARENGONI & STRINGHINI, 2009; BARELLI, 2018).

Os filtros de frequência utilizam transformadas de Fourier para transformar a imagem do domínio espacial para o domínio da frequência. Estes também podem ser classificados quanto ao tipo de frequência em passa-baixas e passa-altas. Filtros passa-baixa bloqueiam altas frequências, suavizando a imagem atenuando regiões de bordas e contornos. Filtros passa-altas, portanto, inibem baixas frequências realçando bordas e contornos. A descrição do funcionamento e modelagem matemática dos filtros são descritas nos trabalhos de Bradski & Kaehler (2008), Marengoni & Stringhini (2009) e Barelli (2018).

Foram implementados no sistema quatro filtros para o tratamento de ruídos, sendo eles o filtro média, gaussiano, mediana e bilateral. A escolha do filtro deve levar em consideração as características do objeto de estudo e ruídos que interferiram na imagem. Estes são descritos a seguir:

3.1.2.1 Filtro média

O filtro médio é um filtro passa-baixas, portanto, atua bloqueando altas frequências suavizando regiões de bordas e contornos.

```
def suaviza_imagem_metodo_filtro_de_media(file):  
    filtro_media = cv.blur(file, (5,5))  
    return filtro_media
```

Este filtro recebe dois parâmetros através do método *blur*, sendo o primeiro o arquivo carregado e o segundo a dimensão da máscara que será aplicada. Quanto maior a máscara aplicada maior será a suavização da imagem. Pode-se comparar a imagem original com a suavizada pelo filtro médio na Figura 4.

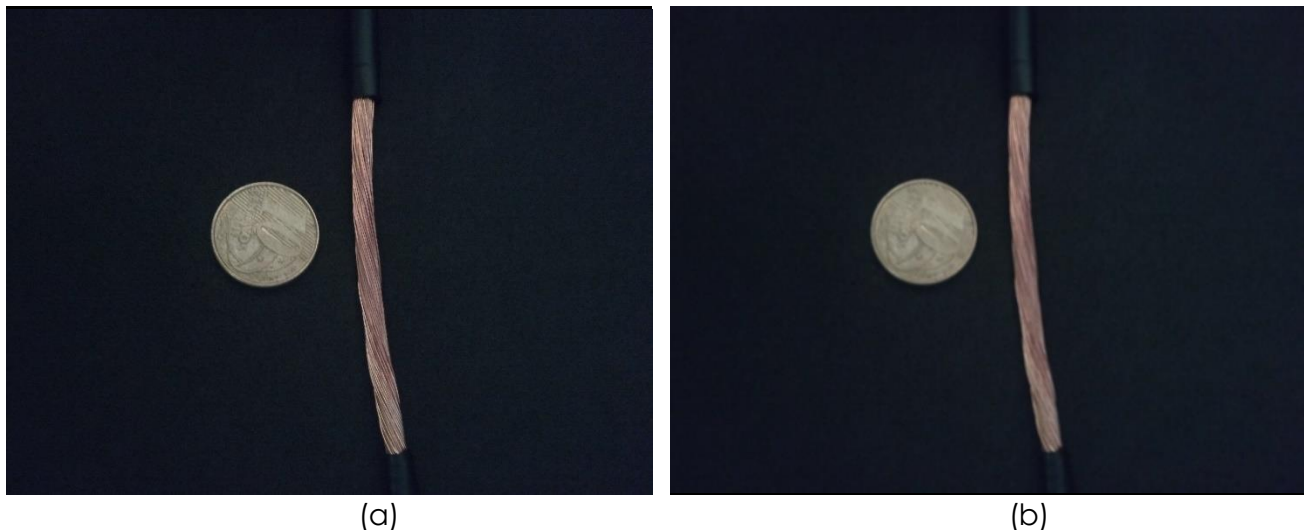


Figura 4. Aplicação dos filtros. (a) original. (b) filtro média.

Fonte: Autor.

3.1.2.2 Filtro Gaussiano

O filtro gaussiano também é um filtro passa-baixas e possui funcionamento semelhante com o filtro de média, porém recebe três parâmetros executados através do método *GaussianBlur*.

```
def suaviza_imagem_metodo_filtro_gaussiano(file):  
    filtro_gaussiano = cv.GaussianBlur(file, (5,5), 0)  
    return filtro_gaussiano
```

O primeiro parâmetro refere-se a imagem carregada, o segundo, a dimensão da matriz que representa a máscara de filtragem, e o último, o grau de suavização. Quanto maior a máscara e o grau de suavização aplicados, maior será a suavização da imagem. De acordo com (MORDVINTSEV & RAHMAN K, c2013), recomenda-se que a máscara seja uma matriz quadrada, com número ímpar de linhas e colunas e o grau de suavização seja definido por um número inteiro positivo. Para exemplificar, compara-se a figura original com a aplicação do filtro Gaussiano na Figura 5.

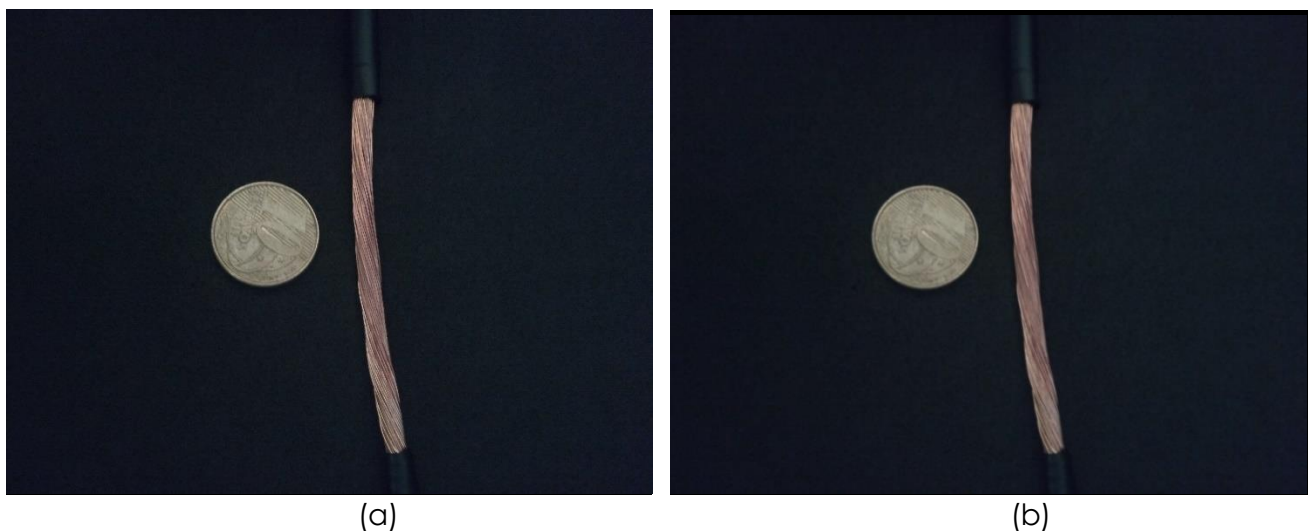


Figura 5. Aplicação dos filtros. (a) original. (b) filtro gaussiano.

Fonte: Autor.

3.1.2.3 Filtro mediana

O filtro mediana não é classificado quanto ao tipo de frequência. Ele atua alterando o valor de cada pixel-alvo pela mediana estatística dos valores dos pixels vizinhos através do método *medianBlur*. Esta técnica possui maior eficácia para suavizar a imagem preservando bordas e contornos.

```
def suaviza_imagem_metodo_filtro_de_mediana(file):  
    filtro_mediana = cv.medianBlur(file, 5)  
    return filtro_mediana
```

Os parâmetros que este filtro recebe são: a imagem carregada e um valor inteiro, ímpar e positivo indicando a intensidade da suavização. Exemplificando, a Figura 6 compara a imagem original com a suavização do filtro mediana.

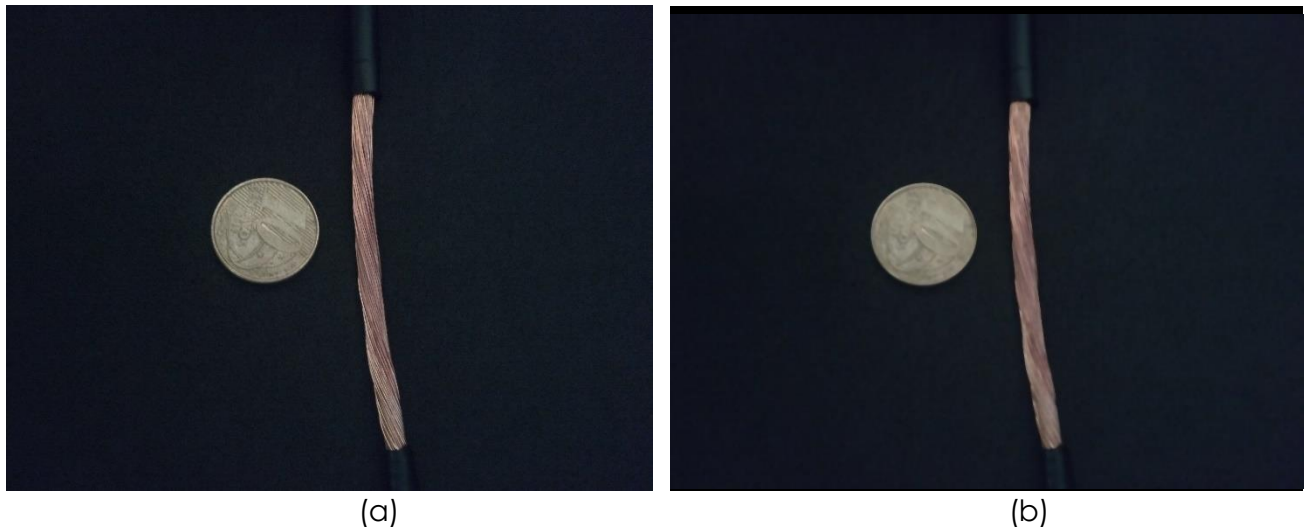


Figura 6. Aplicação dos filtros. (a) original. (b) filtro mediana.

Fonte: Autor.

3.1.2.4 Filtro bilateral

Este é o filtro mais indicado para preservação de bordas e contornos. Ele possui um funcionamento semelhante a filtros passa-baixas, entretanto, modificado para preservar detalhes de contorno.

```
def suaviza_imagem_metodo_bilateral(file):  
    filtro_bilateral = cv.bilateralFilter(file, 10, 100, 100)  
    return filtro_bilateral
```

Os parâmetros requeridos pelo método *bilateralFilter* são: a imagem carregada, a intensidade do filtro, o σ Color (*sigma color*), e o σ Space (*sigma space*). Quanto maior o valor do σ Color, maior será a mistura das cores vizinhas, e quanto maior o valor do σ Space, maior será a influência do filtro nos pixels vizinhos, desde que suas cores sejam próximas.

Todos os valores dos parâmetros de todos os filtros apresentados devem ser testados para melhorar a qualidade da imagem de acordo com o ambiente, iluminação, ruídos e outras interferências em que esta foi submetida no momento da captura. A aplicação do filtro bilateral está demonstrada na Figura 7.

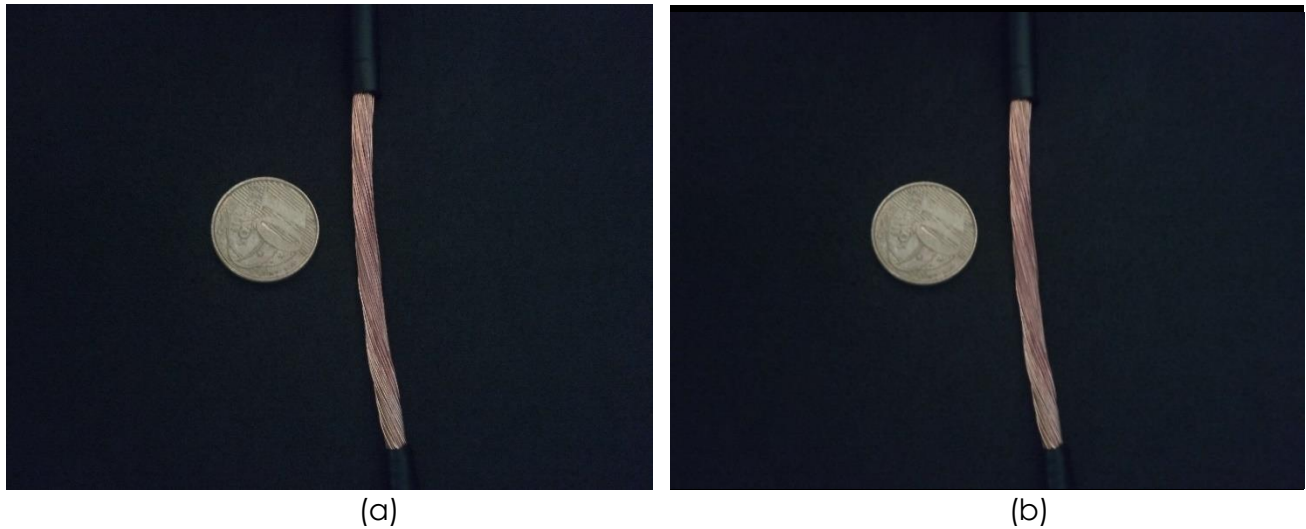


Figura 7. Aplicação dos filtros. (a) original. (b) filtro bilateral.

Fonte: Autor.

3.1.3 Segmentação

A segmentação é uma das etapas mais importantes da visão computacional e consiste em separar os objetos de maior interesse de uma imagem. Os objetos a serem estudados são chamados de primeiro plano, e os pixels que não fazer parte dessa região são denominados como segundo plano (BARELLI, 2018; BRADSKI & KAEHLER, 2008).

Neste projeto foi realizado um processo de segmentação por binarização. Este processo é suscetível a falhas nas regiões que compõem os objetos de interesse. Para corrigir isso, são aplicadas operações morfológicas que modificam o formato ou estrutura dos objetos representados na imagem através de um elemento estruturante, exemplificado como uma imagem binária menor que a imagem original, descrita por uma matriz em que atribuímos suas dimensões (i,j) . Esse elemento estruturante percorre toda a imagem binarizada, corrigindo pixel a pixel

as falhas de acordo com o tipo de operação morfológica (BRADSKI & KAEHLER, 2008; BARELLI, 2018).

3.1.3.1 Binarização

Foram implementados no sistema três funções de binarização que realizam o mesmo processo, porém com algumas características distintas. É importante salientar que para escolha da função a ser utilizada, deve ser levado em consideração a iluminação, ruídos e o nível de detalhes dos objetos que queremos segmentar na figura. Antes de binarizarmos a imagem é necessário transformá-la para outra imagem na escala tons de cinza através da função abaixo:

```
def transforma_para_escala_cinza(file):  
    cinza = cv.cvtColor(file, cv.COLOR_BGR2GRAY)  
    return cinza
```

As três funções de binarização estão apresentadas a seguir:

```
OBJETO_DE_INTERESSE_NA_COR_PRETA = cv.THRESH_BINARY  
OBJETO_DE_INTERESSE_NA_COR_BRANCA = cv.THRESH_BINARY_INV
```

```
def transforma_para_binario(file):  
    cinza = transforma_para_escala_cinza(file)  
    ret, bin = cv.threshold(cinza, 90, 255, OBJETO_DE_INTERESSE_NA_COR_BRANCA)  
    # ret ---> valor inicial do limiar  
    return bin
```

```
def transforma_para_binario_metodo_adaptativo(file):  
    metodo_media = cv.ADAPTIVE_THRESH_MEAN_C  
    metodo_gaussiano = cv.ADAPTIVE_THRESH_GAUSSIAN_C  
    cinza = transforma_para_escala_cinza(file)  
    bin = cv.adaptiveThreshold(cinza, 255, metodo_media,  
OBJETO_DE_INTERESSE_NA_COR_BRANCA, 11, 5)  
    return bin
```

```
def transforma_para_binario_metodo_nobuyuki_otsu(file):  
    tipo = OBJETO_DE_INTERESSE_NA_COR_BRANCA + cv.THRESH_OTSU  
    cinza = transforma_para_escala_cinza(file)  
    limiar, bin = cv.threshold(cinza, 0, 255, tipo)  
    return limiar, bin
```

Para imagens com iluminação não uniforme e muitos detalhes, recomenda-se usar a primeira função que implementa o método *threshold*. Este método recebe quatro parâmetros que são: a imagem transformada para a escala tons de cinza, um número inteiro positivo como valor inicial, outro número inteiro positivo como valor final, e a definição da cor dos objetos de interesse. Exemplificamos seu funcionamento da seguinte forma:

Quando convertemos a imagem original para escala tons de cinza, temos somente um canal de cor que varia a intensidade dos pixels entre 0 e 255, sendo 0 o nível mais baixo de luz, a cor preta, 255 o nível mais alto de luz, a cor branca, e os valores intermediários representando os tons da escala cinza. A partir do quarto parâmetro, definimos a cor dos objetos de interesse e plano de fundo entre branco (255) ou preto (0). O segundo e terceiro parâmetro formam um limiar. Esse limiar deve ser compreendido entre 0 e 255, por exemplo 100 a 200 ou 50 a 255. Os pixels com intensidade luminosa abaixo desse limiar receberão um único valor e os pixels dentro do limiar outro valor fixo. O valor que será recebido depende da definição no quarto parâmetro. Assim, ao final, temos apenas uma imagem com valores de pixel 0 ou 255.

As demais funções encontram limiares aproximados e não possuem eficácia para imagens ruidosas com iluminação não uniforme, sendo eficazes somente para imagens menos complexas.

3.1.3.2 Operações morfológicas

Para implementarmos as operações morfológicas é necessário escolhermos o padrão do elemento estruturante de acordo com as características geométricas dos objetos de interesse na imagem. A biblioteca OpenCV possui três padrões prontos, sendo eles:

```
RETANGULAR = cv.getStructuringElement(cv.MORPH_RECT, (5, 5))
# array([
# [1, 1, 1, 1, 1],
# [1, 1, 1, 1, 1],
# [1, 1, 1, 1, 1],
# [1, 1, 1, 1, 1],
# [1, 1, 1, 1, 1]], dtype=uint8)
```

```

ELIPITICO = cv.getStructuringElement(cv.MORPH_ELLIPSE, (5,5))
# array([
# [0, 0, 1, 0, 0],
# [1, 1, 1, 1, 1],
# [1, 1, 1, 1, 1],
# [1, 1, 1, 1, 1],
# [0, 0, 1, 0, 0]], dtype=uint8)

CRUZ = cv.getStructuringElement(cv.MORPH_CROSS, (5,5))
# array([
# [0, 0, 1, 0, 0],
# [0, 0, 1, 0, 0],
# [1, 1, 1, 1, 1],
# [0, 0, 1, 0, 0],
# [0, 0, 1, 0, 0]], dtype=uint8)

```

Caso necessário, para obtenção de melhores resultados, um novo padrão pode ser criado com o auxílio da biblioteca NumPy.

```

import numpy as np

PERSONALIZADO = np.matrix([
[0, 0, 1, 0, 0],
[0, 1, 1, 1, 0],
[1, 1, 1, 1, 1],
[0, 1, 1, 1, 0],
[0, 0, 1, 0, 0]
], np.uint8)

```

Estabelecido o elemento estruturante, escolhemos então o tipo de operação morfológica a ser aplicado de acordo com suas características. Elas são classificadas em dois grupos que são: operações de erosão e dilatação, e operações de abertura e fechamento.

Dentro do primeiro grupo, a primeira operação é caracterizada pela corrosão das arestas do objeto de estudo que resulta no encolhimento do objeto. A segunda operação realiza o oposto, preenchendo a área de interesse do objeto aumentando suas dimensões.

Para o segundo grupo, a operação de abertura é caracterizada por aplicar as operações do primeiro grupo, uma seguida da outra, resultando em maior eficiência para tratar ruídos gerados durante a binarização. Por fim, a operação de fechamento é usada para preencher a imagem, corrigindo falhas dentro dos objetos de interesse também causados pelo processo de binarização. As funções de operações morfológicas estão apresentadas abaixo:

```

def operacao_morfologica_de_erosao(file):
    elemento_estruturante = ELIPITICO
    imagem_tratada = cv.erode(file, elemento_estruturante, iterations=2)
    return imagem_tratada

def operacao_morfologica_de_dilatacao(file):
    elemento_estruturante = ELIPITICO
    imagem_tratada = cv.dilate(file, elemento_estruturante, iterations=2)
    return imagem_tratada

def operacao_morfologica_de_abertura(file):
    elemento_estruturante = ELIPITICO
    imagem_tratada = cv.morphologyEx(file, cv.MORPH_OPEN, elemento_estruturante)
    return imagem_tratada

def operacao_morfologica_de_fechamento(file):
    elemento_estruturante = ELIPITICO
    imagem_tratada = cv.morphologyEx(file, cv.MORPH_CLOSE, elemento_estruturante)
    return imagem_tratada

```

As operações morfológicas de abertura e fechamento possuem melhor eficiência para tratar falhas na região dos objetos de interesse, portanto, são essenciais no processamento de imagens dos fios trefilados propostos neste projeto.

Pode-se observar na Figura 8 uma imagem binarizada e corrigida através de operações morfológicas.



Figura 8. (a) Imagem binarizada com regiões de falhas. (b) Imagem corrigida por meio das operações morfológicas de abertura e fechamento.

Fonte: Autor.

3.1.4 Extração de características

A extração de características é um processo essencial para categorizarmos os objetos de interesse segmentados. Podemos classificá-los quanto as características de aspecto, dimensionais, inerciais e topológicas. Para este projeto foram extraídas as características dimensionais, que podem ser utilizadas na obtenção da área, perímetro e diâmetro (BRADSKI & KAEHLER, 2008; BARELLI, 2018).

Após a segmentação dos objetos de estudo através da binarização e operações morfológicas, obtêm-se por meio da função a seguir, um *array* (vetor) com os pontos (coordenadas cartesianas compreendidas em x e y) que compõem o perímetro desses objetos.

```
def encontra_objetos_de_interesse(file):  
    imagem_suavizada = suaviza_imagem_metodo_bilateral(file)  
    bin = transforma_para_binario(imagem_suavizada)  
    imagem_tratada = operacao_morfologica_de_abertura(bin)  
    modo = cv.RETR_TREE  
    metodo = cv.CHAIN_APPROX_SIMPLE  
    contornos, hierarquia = cv.findContours(imagem_tratada, modo, metodo)  
    return contornos, hierarquia
```

Exemplificando a função, armazena-se os pontos referentes aos objetos contidos na Figura 9.



Figura 9. Imagem de estudo que possui dois objetos para análise.

Fonte: Autor.

```

[array([[[ 0, 0]],

        [[ 0, 3473]],

        [[4631, 3473]],

        [[4631, 0]]], dtype=int32),

array([[[[1839, 1233]],

        [[1840, 1232]],

        [[1862, 1232]],

        ...,

        [[1819, 1234]],

        [[1830, 1234]],

        [[1831, 1233]]], dtype=int32),

array([[[[2540, 637]],

        [[2541, 636]],

        [[2547, 636]],

        ...,

```

```
[[2531, 638]],
```

```
[[2537, 638]],
```

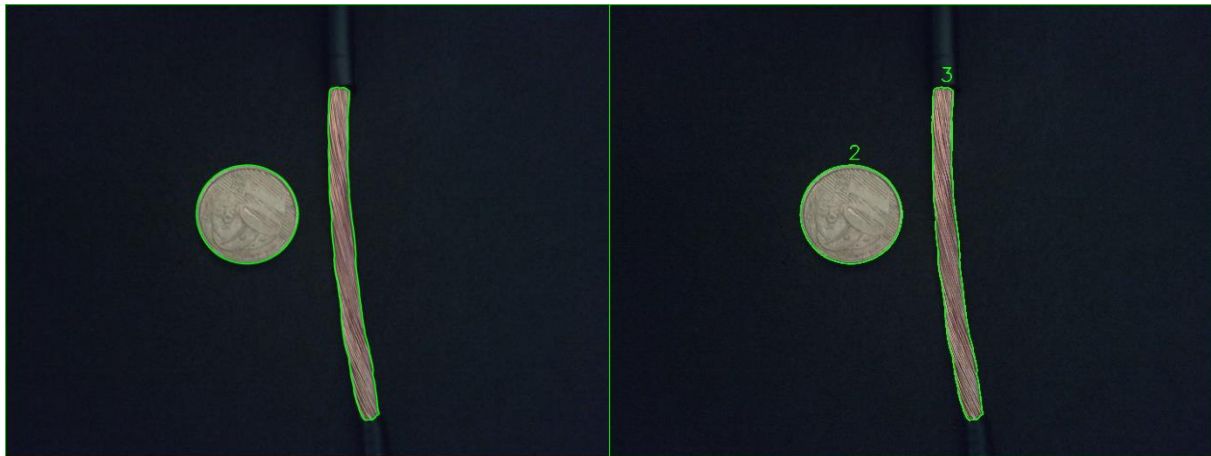
```
[[2538, 637]]], dtype=int32)]
```

Analisando os dados acima pode-se observar que um vetor foi criado para armazenar outros três vetores, estes são: pontos de perímetro do próprio fundo da imagem, da moeda de RS 0,10 e do fio de cobre desencapado. Os dados quanto ao fundo, mesmo que também segmentados não possuem importância no projeto, portanto, são desconsiderados em etapas posteriores.

Partindo desses pontos, é possível por meio das funções a seguir contornar e enumerar os objetos de interesse na imagem para melhorar a identificação visual, conforme demonstrado na Figura 10.

```
def contorna_os_objetos_de_interesse(file):
    contornos, hierarquia = encontra_objetos_de_interesse(file)
    objetos = contornos
    objetos_contornados = cv.drawContours(file, objetos, -1, (0, 255, 0), 10)
    return objetos_contornados

def enumera_os_objetos_de_interesse(file):
    contornos, hierarquia = encontra_objetos_de_interesse(file)
    objetos = contornos
    for ind in range(0, len(objetos)):
        x = objetos[ind][0][0][0]
        y = objetos[ind][0][0][1] - 40
        objetos_enumerados = cv.putText(file, str(ind + 1), (x, y), FONTE, 5, (0,
255, 0), 9, cv.LINE_AA)
    return objetos_enumerados
```



(a)

(b)

Figura 10. (a) imagem com objetos contornados. (b) imagem com objetos contornados e numerados.

Fonte: Autor.

A análise dimensional é realizada através dos pontos armazenados e devem ser feitas de acordo com a geometria do objeto de estudo. Implementaram-se no projeto funções para extração do diâmetro de objetos com geometrias uniformes e não uniformes. A função a seguir é responsável por separar os pontos em vetores de x e y .

```
def coordenadas_dos_objetos_de_interesse(file):
    contornos, hierarquia = encontra_objetos_de_interesse(img)
    x = []
    y = []
    for ind in range(0, len(contornos)):
        objeto = contornos[ind]
        x.append(objeto[:, 0, 0])
        y.append(objeto[:, 0, 1])

    pontos = np.array([x, y])
                        #0, #1
    return pontos
```

A partir das coordenadas x e y , obtêm-se com a função *pontos_maximos_e_minimos_dos_objetos_de_interesse* os pontos máximos e mínimos desses vetores. Com esses valores subtraímos os pontos máximos dos pontos mínimos e então temos por meio da função *diferenca_entre_os_pontos* essa distância.


```

def pontos_maximos_e_minimos_dos_objetos_de_interesse(pontos):
    ponto_minimo_em_x = []
    ponto_maximo_em_x = []
    ponto_minimo_em_y = []
    ponto_maximo_em_y = []
    tamanho_do_vetor = len(pontos[0])

    for ind in range(0, tamanho_do_vetor):
        ponto_minimo_em_x.append(min(pontos[0,ind]))
        ponto_maximo_em_x.append(max(pontos[0,ind]))
        ponto_minimo_em_y.append(min(pontos[1,ind]))
        ponto_maximo_em_y.append(max(pontos[1,ind]))

    pontos_maximos_e_minimos = np.array([ponto_minimo_em_x, #0
                                          ponto_maximo_em_x, #1
                                          ponto_minimo_em_y, #2
                                          ponto_maximo_em_y]) #3

    return pontos_maximos_e_minimos

def diferenca_entre_os_pontos(pontos_maximos_e_minimos):
    diferenca_entre_os_pontos_eixo_x = []
    diferenca_entre_os_pontos_eixo_y = []
    tamanho_do_vetor = len(pontos_maximos_e_minimos[0])

    for ind in range(0, tamanho_do_vetor):
        diferenca_entre_os_pontos_eixo_x.append(pontos_maximos_e_minimos[1, ind]
        - pontos_maximos_e_minimos[0, ind])
        diferenca_entre_os_pontos_eixo_y.append(pontos_maximos_e_minimos[3, ind]
        - pontos_maximos_e_minimos[2, ind])
    diferenca = np.array([diferenca_entre_os_pontos_eixo_x, #0
                          diferenca_entre_os_pontos_eixo_y]) #1

    return diferenca

```

A distância obtida pode ser compreendida como o diâmetro do objeto, porém, esses valores são dados somente em pixels. Para associar e relacionar tais valores de pixels com valores reais (mm) é necessário um outro objeto com dimensões já conhecidas como parâmetro. Isso é feito com o auxílio da função *diametro_de_objetos_uniformes*.

```

def diametro_de_objetos_uniformes(diferenca_entre_os_pontos):
    distancia_fisica_em_x = []
    distancia_fisica_em_y = []
    parametro_fisico = 20      # valor físico de comparação
    parametro_em_pixels_x = diferenca_entre_os_pontos[0,1]
    parametro_em_pixels_y = diferenca_entre_os_pontos[1,1]
    tamanho_do_vetor = len(diferenca_entre_os_pontos[0])

    for ind in range(0, tamanho_do_vetor):
        distancia_fisica_em_x.append((parametro_fisico *
diferenca_entre_os_pontos[0, ind]) / parametro_em_pixels_x)
        distancia_fisica_em_y.append((parametro_fisico *
diferenca_entre_os_pontos[1, ind]) / parametro_em_pixels_y)

    diametros_uniformes = np.array([distancia_fisica_em_x,  #0
                                   distancia_fisica_em_y]) #1

    return diametros_uniformes

```

Analisando a Figura 9, utilizando como parâmetro físico a moeda de R\$ 0,10 de 20 mm de diâmetro, relacionamos os valores em pixels com as medidas reais. Deste modo, obtêm-se os valores em mm para todos os objetos. Entretanto, neste caso, pelo fio desencapado apresentar uma geometria não uniforme, os resultados não foram precisos quando comparados com a medição convencional através de um paquímetro. Assim, implementaram-se as funções a seguir para realizar a extração dos diâmetros com objetos não uniformes.

```

def primeiro_e_ultimo_ponto(pontos):
    primeiro_ponto_em_x = []
    ultimo_ponto_em_x = []
    primeiro_ponto_em_y = []
    ultimo_ponto_em_y = []
    tamanho_do_vetor = len(pontos[0])

    for ind in range(0, tamanho_do_vetor):
        quadrante = (len(pontos[0, ind])-1) / 4
        primeiro_ponto_em_x.append(pontos[0, ind][math.ceil(3.5 * quadrante)])
        ultimo_ponto_em_x.append(pontos[0, ind][math.floor(0.45 * quadrante)])
        primeiro_ponto_em_y.append(pontos[1, ind][math.ceil(3.2 * quadrante)])
        ultimo_ponto_em_y.append(pontos[1, ind][math.floor(0.85 * quadrante)])

    primeiros_e_ultimos_pontos = np.array([primeiro_ponto_em_x,  #0
                                           ultimo_ponto_em_x,    #1
                                           primeiro_ponto_em_y,  #2
                                           ultimo_ponto_em_y])   #3

    return primeiros_e_ultimos_pontos

```

```

def diferenca_entre_primeiro_e_ultimo_ponto(primeiros_e_ultimos_pontos):

    tamanho_do_vetor = len(primeiros_e_ultimos_pontos[0])
    distancia_em_x = []
    distancia_em_y = []

    for ind in range(0, tamanho_do_vetor):
        distancia_em_x.append( int( math.sqrt(math.pow( (
primeiros_e_ultimos_pontos[1, ind] - primeiros_e_ultimos_pontos[0, ind]
) ) , 2)

        distancia_em_y.append( int( math.sqrt(math.pow( (
primeiros_e_ultimos_pontos[3, ind] - primeiros_e_ultimos_pontos[2, ind]
) ) , 2)

    distancias = np.array([distancia_em_x, #0
                           distancia_em_y]) #1

    return distancias

def diametro_de_objetos_nao_uniformes(distancias, diferencas):

    parametro_fisico = 20
    parametro_em_pixels = diferencas[0, 1]
    diametros_nao_uniformes = []

    diametros_nao_uniformes.append(0)
    diametros_nao_uniformes.append((parametro_fisico * parametro_em_pixels) /
parametro_em_pixels)
    diametros_nao_uniformes.append((parametro_fisico * distancias[0, 2]) /
parametro_em_pixels)

    return diametros_nao_uniformes

```

A função *primeiro_e_ultimo_ponto* é responsável por pegar a primeira e última coordenada do perímetro do objeto. Feito isso, pegamos a quantidade total de pontos que contorna este objeto e dividimos por quatro, assim, o mesmo é dividido em quadrantes. Por fim, multiplicamos o quadrante por constantes que variam de um a quatro para percorrer todo objeto. Desta forma, podemos escolher o local exato que realizaremos a medição do diâmetro, conforme a Figura 11.

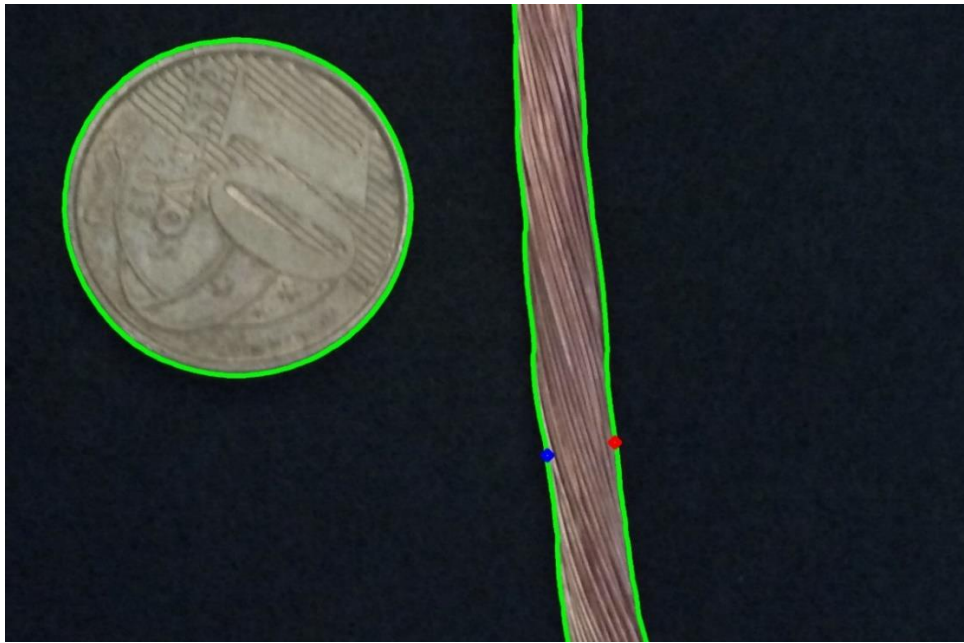


Figura 11. Imagem com os pontos escolhidos para medição do diâmetro.

Fonte: Autor.

Observando a Figura 11, nota-se o ponto azul como o primeiro ponto, obtido através da multiplicação do quadrante em x por 3.5 e em y por 3.2, e o ponto vermelho como o último ponto, dado pela multiplicação em x por 0.45 e em y por 0.85.

A partir desses pontos utilizamos a função *diferenca_entre_primeiro_e_ultimo_ponto* para obter a distância entre o ponto azul e ponto vermelho. Por fim, a função *diagnostico_de_objetos_nao_uniformes* fica responsável por relacionar as medidas reais do parâmetro com as medidas em pixels resultantes das funções anteriores, e assim, obtêm-se os diâmetros para objetos não uniformes.

3.1.5 Reconhecimento de padrões

Embora a etapa de reconhecimento de padrões faça parte do processo de visão computacional, a ferramenta desenvolvida neste trabalho não faz uso de algoritmos computacionais classificadores.

3.2. Páginas HTML e microframework Flask

Para visualização gráfica do projeto foram escritas as páginas HTML *template.html*, *index.html* e *relatório_imagem.html*. A página *template.html* é responsável por importar todos as páginas de estilo CSS e *scripts Javascript*. A página *index.html* é nossa página de entrada que contém o formulário para submissão da imagem à ser analisada. Por fim, a página *relatorio_imagem.html* exibi o relatório com os metadados da imagem, a imagem original, binarizada e contornada e os resultados finais com os diâmetros dos objetos segmentados.

Para decorar e estilizar as páginas HTML foram utilizados o CSS padrão do *Bootstrap* e criado uma outra página de estilo nomeada de *template.css*.

A integração das funções implementadas no arquivo *helpers.py* com os *templates* HTML, foi realizada através do *microframework* Flask no arquivo *views.py*.

Os códigos HTML, CSS e Flask estão disponíveis para visualização na seção 6, assim como as telas gráficas.

3.3. Validação do sistema

Para validar a precisão do sistema, foram utilizados objetos com dimensões conhecidas do nosso cotidiano. A validação foi feita com moedas brasileiras cujos os diâmetros nominais estão explícitos na Tabela 1.

Tabela 1. Valores e diâmetros nominais das moedas utilizadas como corpos de prova.

Valor (R\$)	D _n (mm)
0,05	22
0,10	20
0,25	25
0,50	23
1,00	27

Legenda: D_n = diâmetro nominal.

Fonte: Autor.

Seguindo os processos apresentados da seção 3.1.1 à 3.1.4, obteve-se a Figura 12 que exemplifica o fluxo de funcionamento da visão computacional e os resultados apresentados na Tabela 2.

Tabela 2. Diâmetros das moedas obtidos pelo sistema de visão computacional.

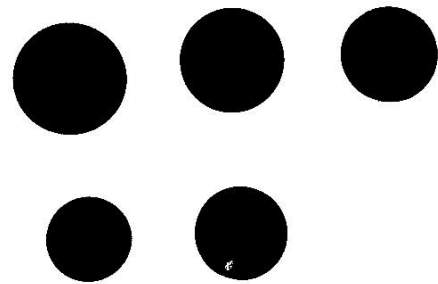
Valor (R\$)	D _{vc} (mm)	Erro (%)
0,05	21,607	1,79
0,10	20,000*	-
0,25	24,296	2,82
0,50	22,596	1,76
1,00	26,553	1,66

Legenda: * Parâmetro dimensional de referência; D_{vc} = diâmetro obtido através do sistema de visão computacional.

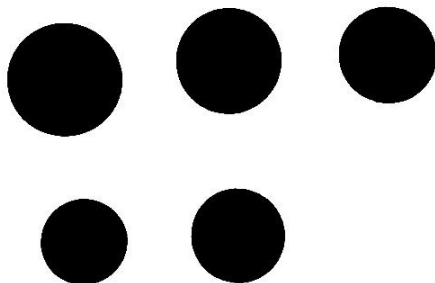
Fonte: Autor.



(a)



(b)



(c)



(d)

Figura 12. Imagem das moedas (corpos de prova). (a) original. (b) binarizada. (c) morfológica. (d) processada.

Fonte: Autor.

É possível analisar na Figura 12 (a) os resultados obtidos no processamento da imagem. Nota-se na Figura 12 (b), que a binarização direta da imagem original resultou em pequenas regiões de falha nos objetos de estudo, assim como observado na moeda de R\$ 0,05. A Figura 12 (c) corrigi essas falhas por meio de operações morfológicas. Por fim, na Figura 12 (d), temos os objetos segmentados, contornados e numerados.

As medidas obtidas foram próximas aos valores nominais, apresentando um erro máximo de 2,82%. Deste modo, os resultados demonstraram um grau de precisão da ordem de centésimos de mm, porém tal precisão pode ser afetada pelas interferências apresentadas na seção 3.1.2.

Como exemplo, podemos comparar a Figura 13, com iluminação inadequada, com a Figura 12. Nota-se claramente que alguns dos objetos de estudo (moedas de R\$ 0,10 e R\$ 0,05), receberam pouca iluminação e consequentemente foram detectadas de maneira errônea. Neste caso, mesmo com a utilização das ferramentas de suavização, binarização e morfologia, não foi possível segmentar corretamente todos os objetos, contrapondo os resultados apresentados na Figura 12.

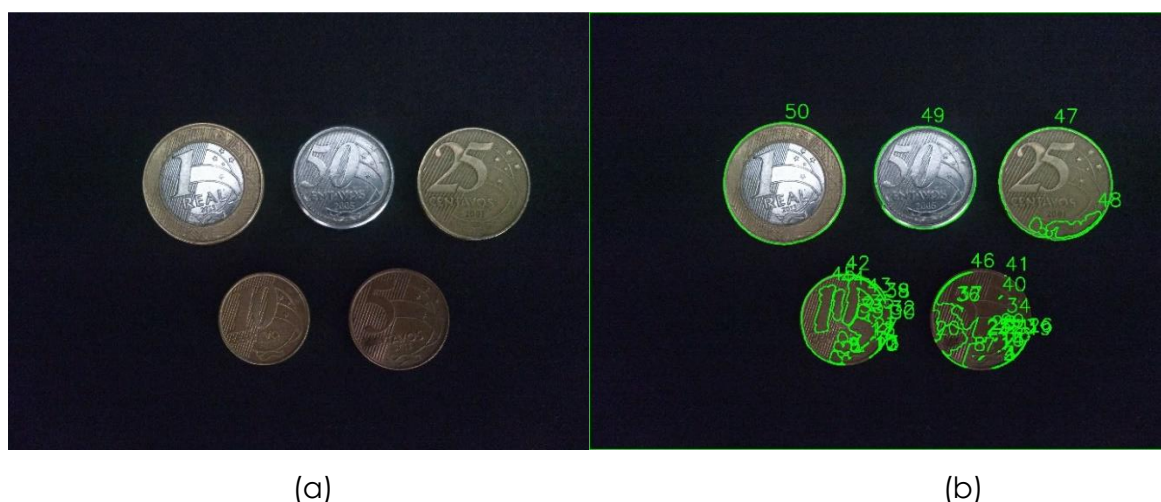


Figura 13. Imagem das moedas com iluminação inadequada. (a) original. (b) processada.

Fonte: Autor.

Para evitar a segmentação inadequada, conforme a Figura 13, deve-se controlar ao máximo o ambiente da captura, mantendo a iluminação o mais uniforme possível a fim de diminuir as interferências que agem sobre a cena. Após a captura, realizar o pré-processamento com as ferramentas já apresentadas para tratar os ruídos e falhas que permaneceram. Seguindo essas etapas, aumentamos muito a qualidade da imagem para análise e consequentemente os resultados serão satisfatórios.

4. Considerações Finais

Implementando as ferramentas de visão computacional disponíveis pela biblioteca OpenCV, utilizando Python como a linguagem base de programação, desenvolveu-se um sistema capaz de obter valores de diâmetro de fios trefilados usando um corpo conhecido como parâmetro dimensional. Para validar o sistema, utilizou-se objetos comuns do nosso cotidiano e atingiu-se um resultado eficiente na análise dimensional.

Durante o desenvolvimento do projeto, observou-se a existência de interferências que podem agir diretamente na captura e processamento de imagens, afetando os resultados finais. A fim de melhorar os dados a serem analisados, otimizou-se as funções responsáveis pela aplicação de filtros, binarização e operações morfológicas, compensando os principais fatores que agem sobre a imagem, dentre eles a iluminação inadequada da cena em que o corpo de análise se situa, qualidade dos sensores de captura e influência do ambiente.

O sistema apresentou-se totalmente adaptável, podendo processar imagens em diferentes ambientes com variações de iluminação e contraste, tratando ruídos e analisando os objetos de interesse com geometrias uniformes e não uniformes. Essa característica nos permite aplicar o sistema em diversos ambientes industriais, adaptando-se as condições em que a câmera de captura e sistemas de iluminação serão inseridos.

Expandindo o sistema, em projetos futuros, é factível utilizarmos a base de desenvolvimento aqui apresentada para processar imagens em tempo real. Como exemplo, pode-se afinar as ferramentas de visão computacional para identificar possíveis variações de diâmetros durante o processo de trefilação após o material passar pela fieira. Embora controlar as variáveis responsáveis por causar interferências em imagens exija grande capacidade de processamento, investir em tecnologias para realizar essa tarefa pode resultar em redução de custos de produção e aumento na rapidez do controle de qualidade de produtos.

As ferramentas de visão computacional não se limitam apenas a análises dimensionais, mas podem ser utilizadas em diversas áreas da engenharia mecânica. Assim, a exploração de tais ferramentas pode resultar em muitos benefícios tanto para os setores industriais como em possíveis avanços em trabalhos acadêmicos que envolvam processamento de imagens e vídeos.

5. Referências

ALTAN, T.; Oh, S.; GEGEL, H. L. *Conformação de metais: fundamentos e aplicações*. São Carlos: EESC-USP, 1999.

BALLARD, D. H.; BROWN, C. M. *Computer Vision*. Nova Jersey: Prentice Hall, 1982.

BARELLI, F. C. *Introdução à Visão Computacional*. São Paulo: Casa do Código, 2018.

BRADSKI G.; KAEHLER A. *Learning OpenCV*. 1 ed. Sebastopol: O'Reilly Media, Inc., 2008.

BUTTON, S. T. *Trefilação: programa de educação continuada*. São Paulo: ABM, 2001.

DIETER, G. E. *Metalurgia Mecânica*. 2 ed. Rio de Janeiro: Guanabara Dois, 1981.

HELMAN, H.; CETLIN, P. R. *Fundamentos da Conformação Mecânica dos Metais*. 1 ed. Rio de Janeiro: Guanabara Dois, 1983.

LUNDH, F.; CLARK, A & Contributors. *Pillow*. c2015. Disponível em: <<https://pillow.readthedocs.io/en/3.0.0/index.html>>. Acesso em: 29 ago. 2019.

LUNDH, F.; ELLIS, M. *Python Imaging Library Overview*. PIL 1.1.3. 2002.

MARENGONI, M.; STRINGHINI, D. Tutorial: Introdução à Visão Computacional usando OpenCV. *Revista de Informática Teórica e Aplicada*, Porto Alegre, v. 16, n. 1, 2009.

MORDVINTSEV, A.; RAHMAN K, A. *Welcome to OpenCV-Python Tutorials's documentation!* c2013. Disponível em: <<https://opencv-python-tutroals.readthedocs.io/en/latest/index.html>>. Acesso em: 29 ago. 2019.

NUMPY DEVELOPERS. *NumPy Documentation*. c2019. Disponível em: <<https://numpy.org/doc/>>. Acesso em: 29 ago. 2019.

NUNES, R. M. *Estudo de distorção de barras cilíndricas de aço ABNT 1045 em uma rota de fabricação envolvendo trefilação combinada e têmpera por indução*. Tese (Doutorado em

Engenharia) - Programa de Pós-Graduação em Engenharia de Minas, Metalúrgica e de Materiais - PPGE3M, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2012. Acesso em: 30 mai. 2019.

RUDEK, M.; COELHO, L. S.; CANGIOLIERI JR., O. *Visão Computacional Aplicada a Sistemas Produtivos: Fundamentos e Estudo de Caso*. XXI Encontro Nacional de Engenharia de Produção - 2001, Salvador: 2001.

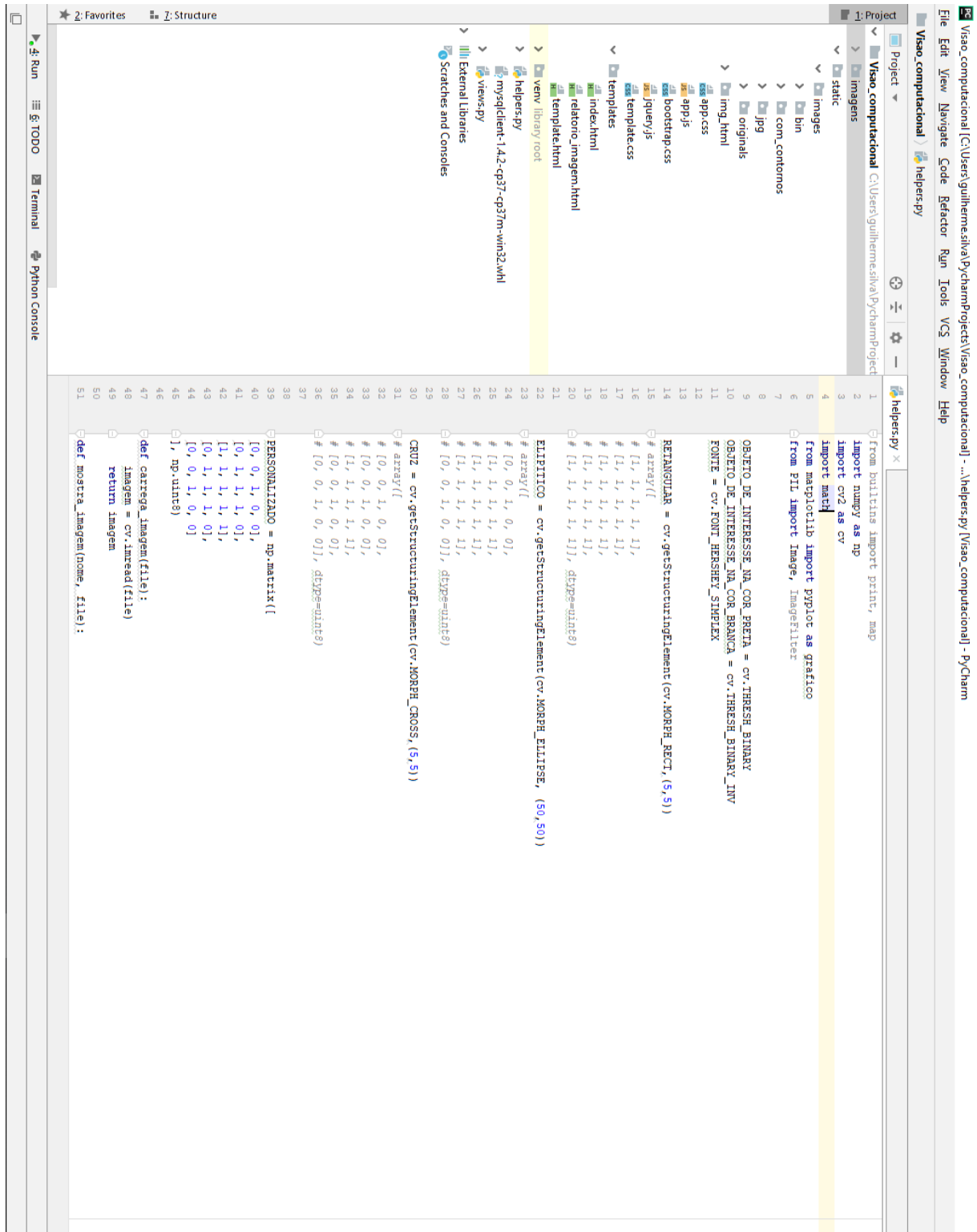
SOARES, C. A. T. *Análise das Tensões Residuais no Processo de Trefilação Considerando os Efeitos de Anisotropia*. 2012. Dissertação (Mestrado em Engenharia) - Programa de Pós-Graduação em Engenharia de Minas, Metalúrgica e de Materiais - PPGE3M, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2012. Acesso em: 30 mai. 2019.

SOUZA, T. F. *Simulações Computacionais para Análise e Minimização das Tensões Residuais no Processo de Trefilação*. 2011. Dissertação (Mestrado em Engenharia) - Programa de Pós-Graduação em Engenharia de Minas, Metalúrgica e de Materiais - PPGE3M, Universidade Federal do Rio Grande do Sul, Porto Alegre, 2011. Acesso em: 30 mai. 2019.

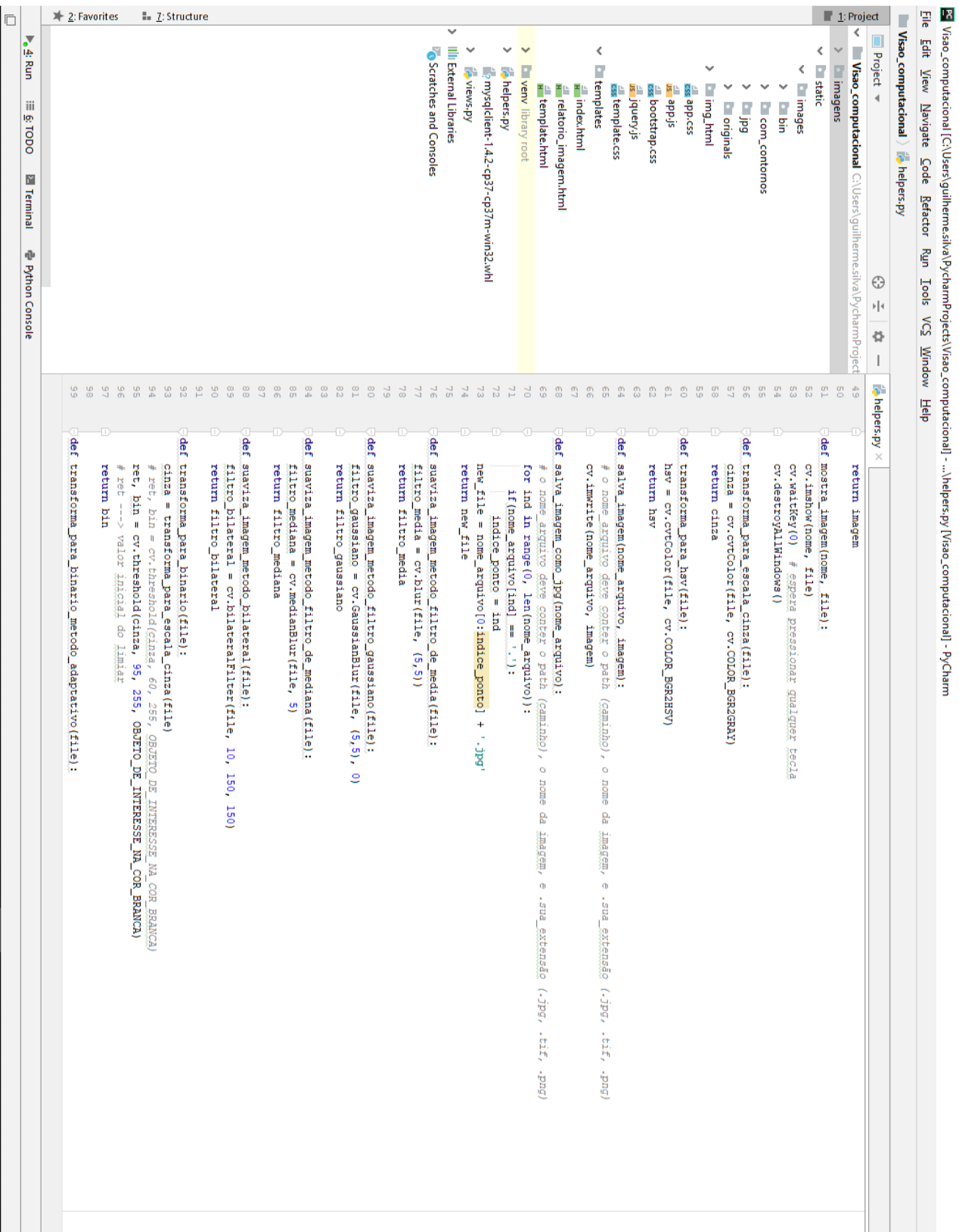
6. Anexos

6.1. Funções implementadas no arquivo *helpers.py*

Ferramentas de visão computacional implementadas a partir da biblioteca OpenCV que compõem o projeto.



```
1 from builtins import print, map
2 import numpy as np
3 import cv2 as cv
4 import math
5 from matplotlib import pyplot as gráfico
6 from PIL import Image, ImageFilter
7
8
9 OBJETO_DE_INTERESSE_NA_COR_PRETA = cv.THRESH_BINARY
10 OBJETO_DE_INTERESSE_NA_COR_BRANCA = cv.THRESH_BINARY_INV
11 FONTE = cv.FONT_HERSHEY_SIMPLEX
12
13
14 RETANGULAR = cv.getStructuringElement(cv.MORPH_RECT, (5, 5))
15
16 # array[[
17 # [1, 1, 1, 1, 1],
18 # [1, 1, 1, 1, 1],
19 # [1, 1, 1, 1, 1],
20 # [1, 1, 1, 1, 1],
21 # [1, 1, 1, 1, 1]]
22
23 ELIPTICO = cv.getStructuringElement(cv.MORPH_ELLIPSE, (50, 50))
24 # array[[
25 # [0, 0, 1, 0, 0],
26 # [1, 1, 1, 1, 1],
27 # [1, 1, 1, 1, 1],
28 # [1, 1, 1, 1, 1],
29 # [0, 0, 1, 0, 0]]
30
31 CRUZ = cv.getStructuringElement(cv.MORPH_CROSS, (5, 5))
32 # array[[
33 # [0, 0, 1, 0, 0],
34 # [0, 0, 1, 0, 0],
35 # [1, 1, 1, 1, 1],
36 # [0, 0, 1, 0, 0],
37 # [0, 0, 1, 0, 0]]
38
39 PERSONALIZADO = np.matrix([
40 [0, 0, 1, 0, 0],
41 [0, 1, 1, 1, 0],
42 [1, 1, 1, 1, 1],
43 [0, 1, 1, 1, 0],
44 [0, 0, 1, 0, 0]
45 ], np.uint8)
46
47 def carrega_imagem(file):
48     imagem = cv.imread(file)
49     return imagem
50
51 def mostra_imagem(nome, file):
```

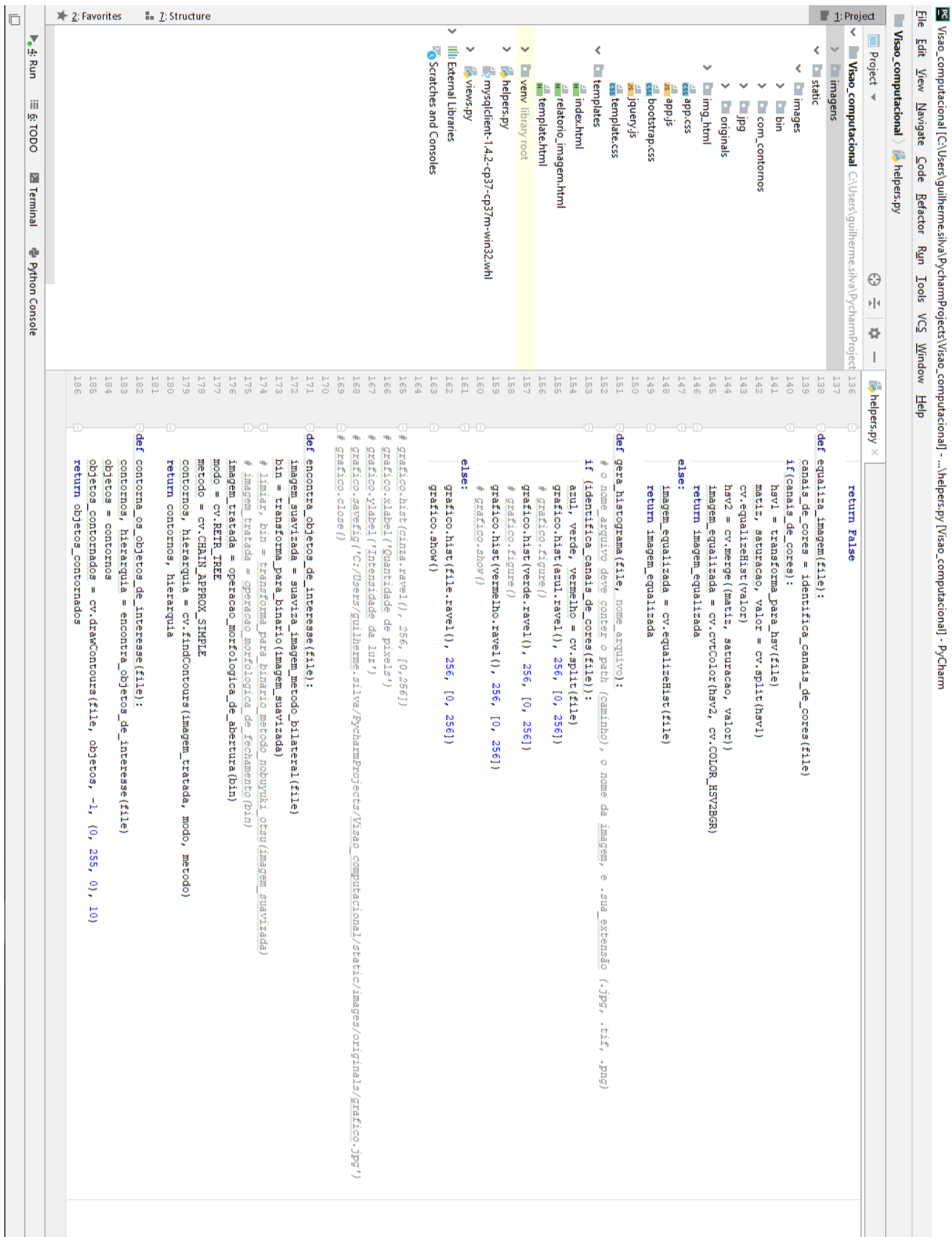


Visao_computacional helpers.py

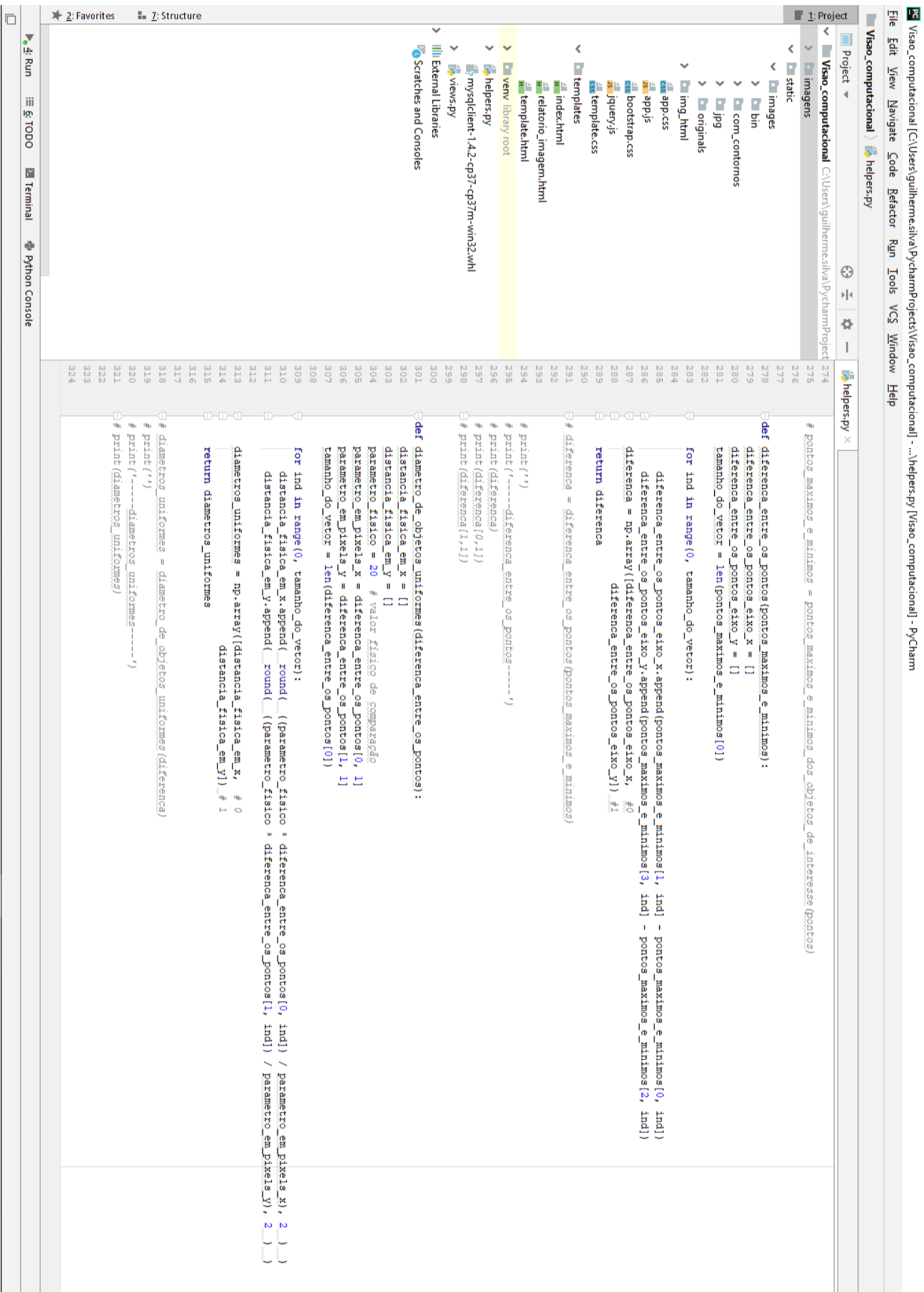
```

1: Project
  > Project
    > Visao_computacional C:\Users\guilherme.silva\PycharmProject
      > static
      > images
      > bin
      > com_contornos
      > jpg
      > originals
      > img.html
      > app.css
      > app.js
      > bootstrap.css
      > jquery.js
      > template.css
      > templates
      > index.html
      > relatorio_imagem.html
      > template.html
      > venv library root
      > helpers.py
      > mysqlclient-1.4.2-cp37-cp37m-win32.whl
      > views.py
      > External Libraries
      > Scratches and Consoles

helpers.py x
97 return bin
98
99 def transforma_para_binario_metodo_adaptativo(file):
100     metodo_media = cv.ADAPTIVE_THRESH_MEAN_C
101     metodo_gaussiano = cv.ADAPTIVE_THRESH_GAUSSIAN_C
102     cinza = transforma_para_escala_cinza(file)
103     bin = cv.adaptiveThreshold(cinza, 255, metodo_media, OBJETO_DE_INTERESSE_NA_COR_BRANCA, 11, 5)
104     return bin
105
106 def transforma_para_binario_metodo_nobuyuki_otsu(file):
107     tipo = OBJETO_DE_INTERESSE_NA_COR_BRANCA + cv.THRESH_OTSU
108     cinza = transforma_para_escala_cinza(file)
109     limiar, bin = cv.threshold(cinza, 0, 255, tipo)
110     return limiar, bin
111
112 def operacao_morfologica_de_erosao(file):
113     elemento_estruturante = ELIPITICO
114     imagem_tratada = cv.erode(file, elemento_estruturante, iterations=2)
115     return imagem_tratada
116
117 def operacao_morfologica_de_dilatacao(file):
118     elemento_estruturante = ELIPITICO
119     imagem_tratada = cv.dilate(file, elemento_estruturante, iterations=2)
120     return imagem_tratada
121
122 def operacao_morfologica_de_abertura(file):
123     elemento_estruturante = ELIPITICO
124     imagem_tratada = cv.morphologyEx(file, cv.MORPH_OPEN, elemento_estruturante)
125     return imagem_tratada
126
127 def operacao_morfologica_de_fechamento(file):
128     elemento_estruturante = ELIPITICO
129     imagem_tratada = cv.morphologyEx(file, cv.MORPH_CLOSE, elemento_estruturante)
130     return imagem_tratada
131
132 def identifica_canais_de_cores(file):
133     if len(file.shape) > 2:
134         return True
135     else:
136         return False
137
138 def equaliza_imagem(file):
139     canais_de_cores = identifica_canais_de_cores(file)
140     if (canais_de_cores):
141         havi = transforma_para_hsv(file)
142         matiz, saturacao, valor = cv.split(havi)
143         cv.equalizeHist(valor)
144         havi2 = cv.merge((matiz, saturacao, valor))
145         imagem_equalizada = cv.cvtColor(havi2, cv.COLOR_HSV2BGR)
146         return imagem_equalizada
147     else:
148         return imagem_equalizada
149
  
```







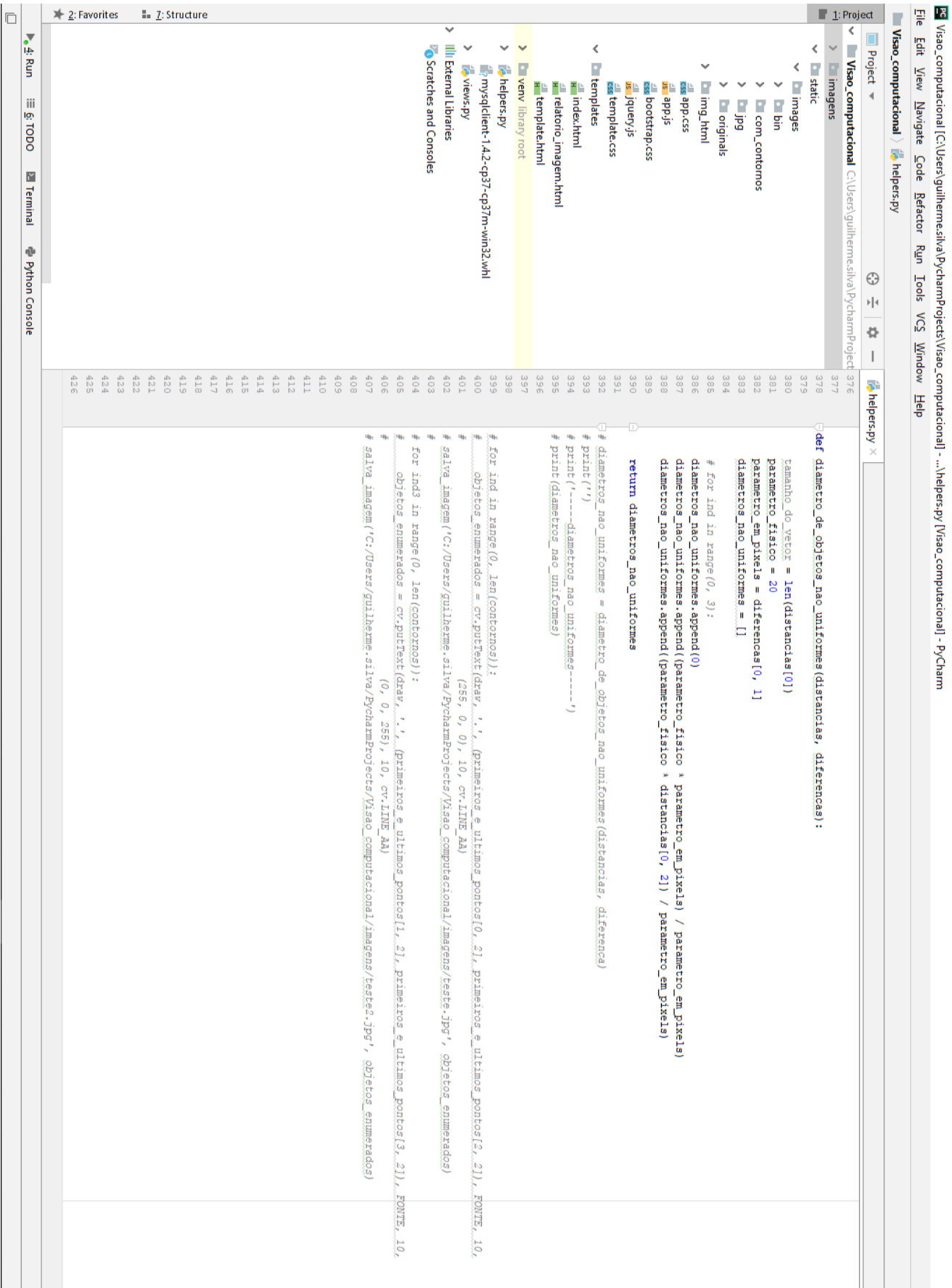
Visao_computacional helpers.py

Project
 1. Project
 Visao_computacional C:\Users\guilhermesiva\PycharmProjects\Visao_computacional
 > static
 > images
 > bin
 > com_contornos
 > jpg
 > originals
 > img_html
 > app.css
 > app.js
 > bootstrap.css
 > jquery.js
 > templates
 > index.html
 > relatorio_imagem.html
 > template.html
 > venv library root
 > helpers.py
 > mysqlclient-14.2-cp37-cp37m-win32.whl
 > views.py
 > External Libraries
 > Scratches and Consoles

```

325 def primeiro_e_ultimo_ponto(pontos):
326     primeiro_ponto_em_x = []
327     ultimo_ponto_em_x = []
328     primeiro_ponto_em_y = []
329     ultimo_ponto_em_y = []
330     tamanho_do_vetor = len(pontos[0])
331
332     for ind in range(0, tamanho_do_vetor):
333         quadrante = (len(pontos[0], ind)-1) / 4
334         primeiro_ponto_em_x.append(pontos[0, ind][math.floor(0.45 * quadrante)])
335         ultimo_ponto_em_x.append(pontos[0, ind][math.floor(0.45 * quadrante)])
336         primeiro_ponto_em_y.append(pontos[1, ind][math.ceil(3.2 * quadrante)])
337         ultimo_ponto_em_y.append(pontos[1, ind][math.floor(0.85 * quadrante)])
338
339     # primeiro_ponto_em_x.append(pontos[0, ind][math.floor(1.12 * quadrante)])
340     # ultimo_ponto_em_x.append(pontos[0, ind][math.floor(1.12 * quadrante)])
341     # primeiro_ponto_em_y.append(pontos[1, ind][math.ceil(3.2 * quadrante)])
342     # ultimo_ponto_em_y.append(pontos[1, ind][math.floor(0.76 * quadrante)])
343
344     primeiros_e_ultimos_pontos = np.array([primeiro_ponto_em_x, #0
345                                             ultimo_ponto_em_x, #1
346                                             primeiro_ponto_em_y, #2
347                                             ultimo_ponto_em_y]) #3
348
349     return primeiros_e_ultimos_pontos
350
351 # print('')
352 # print('-----primeiros e ultimos pontos-----')
353 # print(primeiros_e_ultimos_pontos)
354
355 def diferenca_entre_primeiro_e_ultimo_ponto(primeiros_e_ultimos_pontos):
356     tamanho_do_vetor = len(primeiros_e_ultimos_pontos[0])
357     distancia_em_x = []
358     distancia_em_y = []
359
360     for ind in range(0, tamanho_do_vetor):
361         # distancia_em_x.append( primeiros_e_ultimos_pontos[1, ind] - primeiros_e_ultimos_pontos[0, ind])
362         # distancia_em_y.append( primeiros_e_ultimos_pontos[3, ind] - primeiros_e_ultimos_pontos[2, ind])
363         distancia_em_x.append( int( math.sqrt(math.pow( ( primeiros_e_ultimos_pontos[1, ind] - primeiros_e_ultimos_pontos[0, ind] ) , 2 ) ) ) )
364         distancia_em_y.append( int( math.sqrt(math.pow( ( primeiros_e_ultimos_pontos[3, ind] - primeiros_e_ultimos_pontos[2, ind] ) , 2 ) ) ) )
365
366     distancias = np.array([distancia_em_x, #0
367                             distancia_em_y]) #1
368
369     return distancias
370
371
372 # distancias = diferenca_entre_primeiro_e_ultimo_ponto(primeiros_e_ultimos_pontos)
373 # print('')
374 # print('-----distancias-----')
375 # print(distancias)

```



6.2. Integração com o microframework Flask no arquivo views.py

Códigos de integração entre os *templates* HTML e as funções de visão computacional executadas pelo *microframework* Flask.



```
1 import os
2 from flask import Flask, render_template, request, session, flash, redirect, url_for, send_from_directory
3 from werkzeug.utils import secure_filename
4
5 from helpers import *
6
7 app = Flask(__name__)
8 app.secret_key = 'visao_computacional'
9
10 UPLOAD_FOLDER = 'C:/Users/guilherme.silva/PycharmProjects/Visao_computacional/static/images/originals/'
11 ALLOWED_EXTENSIONS = {'png', 'jpg', 'jpeg', 'gif', 'tif', 'tiff'}
12 app.config['UPLOAD_FOLDER'] = UPLOAD_FOLDER
13
14 @app.route('/')
15 def index():
16     return render_template('index.html')
17
18
19 def allowed_file(filename):
20     return '.' in filename and \
21         filename.rsplit('.', 1)[1].lower() in ALLOWED_EXTENSIONS
22
23
24 @app.route('/relatorio_imagem', methods=['POST', ])
25 def relatorio_imagem():
26     file = request.files['file']
27     upload_path = app.config['UPLOAD_FOLDER']
28
29     if (request.method == 'POST'):
30         if 'file' not in request.files:
31             flash('Nenhuma parte do arquivo', 'alert alert-danger')
32             return redirect(url_for('index'))
33         if (file.filename == ''):
34             flash('Nenhum arquivo selecionado', 'alert alert-danger')
35             return redirect(url_for('index'))
36         if (file and allowed_file(file.filename)):
37             filename = secure_filename(file.filename)
38             file.save(os.path.join(upload_path, filename))
39             # return redirect(url_for('uploaded_file',
40             #                             filename=filename))
41         else:
42             flash('Extensão do arquivo não suportada', 'alert alert-danger')
43             return redirect(url_for('index'))
44
45
46 caminho_e_nome_do_arquivo = 'C:/Users/guilherme.silva/PycharmProjects/Visao_computacional/static/images/originals/' + file.filename
47 imagem = carrega_imagem(caminho_e_nome_do_arquivo)
48
49 caminho_imagem_jpg = upload_path.replace('originals', 'jpg') + salva_imagem_com_jpg(file.filename)
50 salva_imagem(caminho_imagem_jpg, imagem)
51
```




FileEditViewNavigateCodeRefactorRunToolsVCSWindowHelp

Visao_computacionalviews.py

ProjectVisao_computacionalC:\Users\guilherme.silva\PycharmProjects\Visao_computacional - PyCharmviews.py

staticimagesbincom_contornosjpgoriginalsmg.htmlappcssappjsbootstrapcssjqueryicstemplatecss

templatesindex.htmlrelatorio_imagem.htmltemplate.html

venv library roothelpers.py

myclient-1.4.2-cp37-win32.whlviews.py

External LibrariesScratches and Consoles

4 Run6 TODOTerminalPython Console

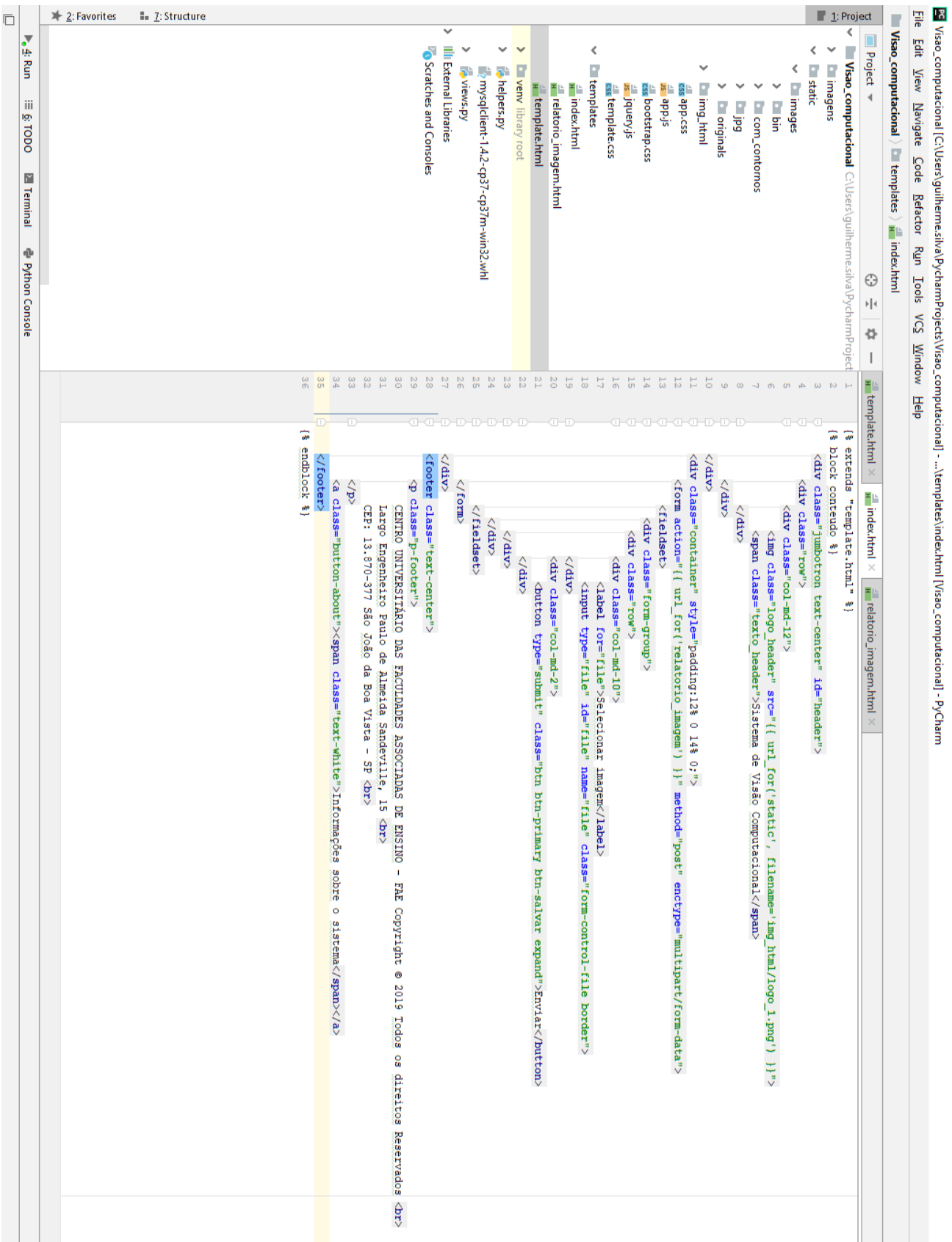
views.py

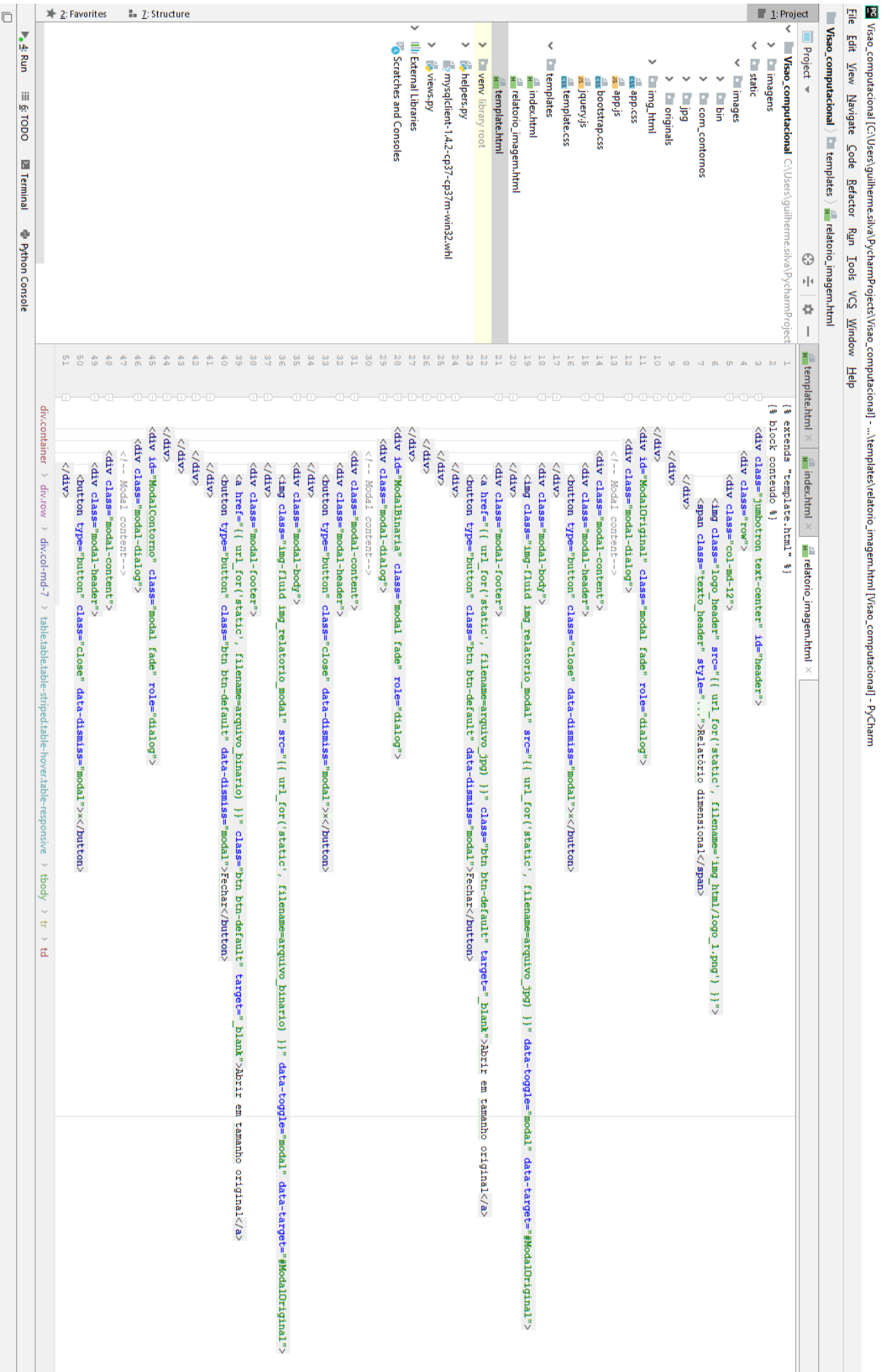
```
54
55
56     imagem_suavizada = suaviza_imagem_metodo_bilateral(imagem)
57     bin = transforma_para_binario(imagem_suavizada)
58     # bin = transforma_para_binario_metodo_adaptativo(imagem_suavizada)
59     # limiar, bin = transforma_para_binario_metodo_nobuyuki_otsu(imagem_suavizada)
60     # imagem_tratada = operacao_morfologica_de_abertura(bin)
61     imagem_tratada = operacao_morfologica_de_fechamento(bin)
62     caminho_imagem_binaria = upload_path.replace('originals', 'bin') + salva_imagem_como_jpg(file.filename)
63     salva_imagem(caminho_imagem_binaria, imagem_tratada)
64     arquivo_binario = 'images/bin/' + salva_imagem_como_jpg(file.filename)
65
66     # as informações e cálculos de área deve vir antes do método enumera os objetos de interesse(),
67     # pois ele altera as propriedades da imagem original
68     info = informacoes_imagem(imagem, caminho_e_nome_do_arquivo)
69
70     # objetos uniformes
71     pontos = coordenadas dos objetos de interesse(imagem)
72     pontos_maximos_e_minimos = pontos_maximos_e_minimos_dos_objetos_de_interesse(pontos)
73     diferenca = diferenca_entre_os_pontos(pontos_maximos_e_minimos)
74     diametro_uniformes = diametro_de_objetos_uniformes(diferenca)
75
76     # objetos não uniformes
77     primarios_e_ultimos_pontos = primario_e_ultimo_ponto(pontos)
78     distancias = diferenca_entre_primario_e_ultimo_ponto(primarios_e_ultimos_pontos)
79     diametro_nao_uniformes = diametro_de_objetos_nao_uniformes(distancias, diferenca)
80
81     objetos_contornados = contorna_os_objetos_de_interesse(imagem)
82     objetos_contornados_e_numerados = enumera_os_objetos_de_interesse(objetos_contornados)
83     caminho_imagem_com_contornos = upload_path.replace('originals', 'com_contornos') + salva_imagem_como_jpg(file.filename)
84     salva_imagem(caminho_imagem_com_contornos, objetos_contornados_e_numerados)
85     arquivo_com_contornos = 'images/com_contornos/' + salva_imagem_como_jpg(file.filename)
86
87
88
89
90
91
92     return render_template('relatorio_imagem.html',
93                           diferenca=diferenca,
94                           diametro_uniformes=diagramas_uniformes,
95                           informacoes=info,
96                           nome_arquivo=file.filename,
97                           arquivo_jpg=arquivo_jpg,
98                           arquivo_binario=arquivo_binario,
99                           arquivo_com_contornos=arquivo_com_contornos)
100
101 # def uploaded_file(filename):
102 #     return send_from_directory(app.config['UPLOAD_FOLDER'], filename)
103
104 app.run(debug=True)
```


6.3. Templates HTML e CSS

Templates HTML e CSS que compõem o projeto.

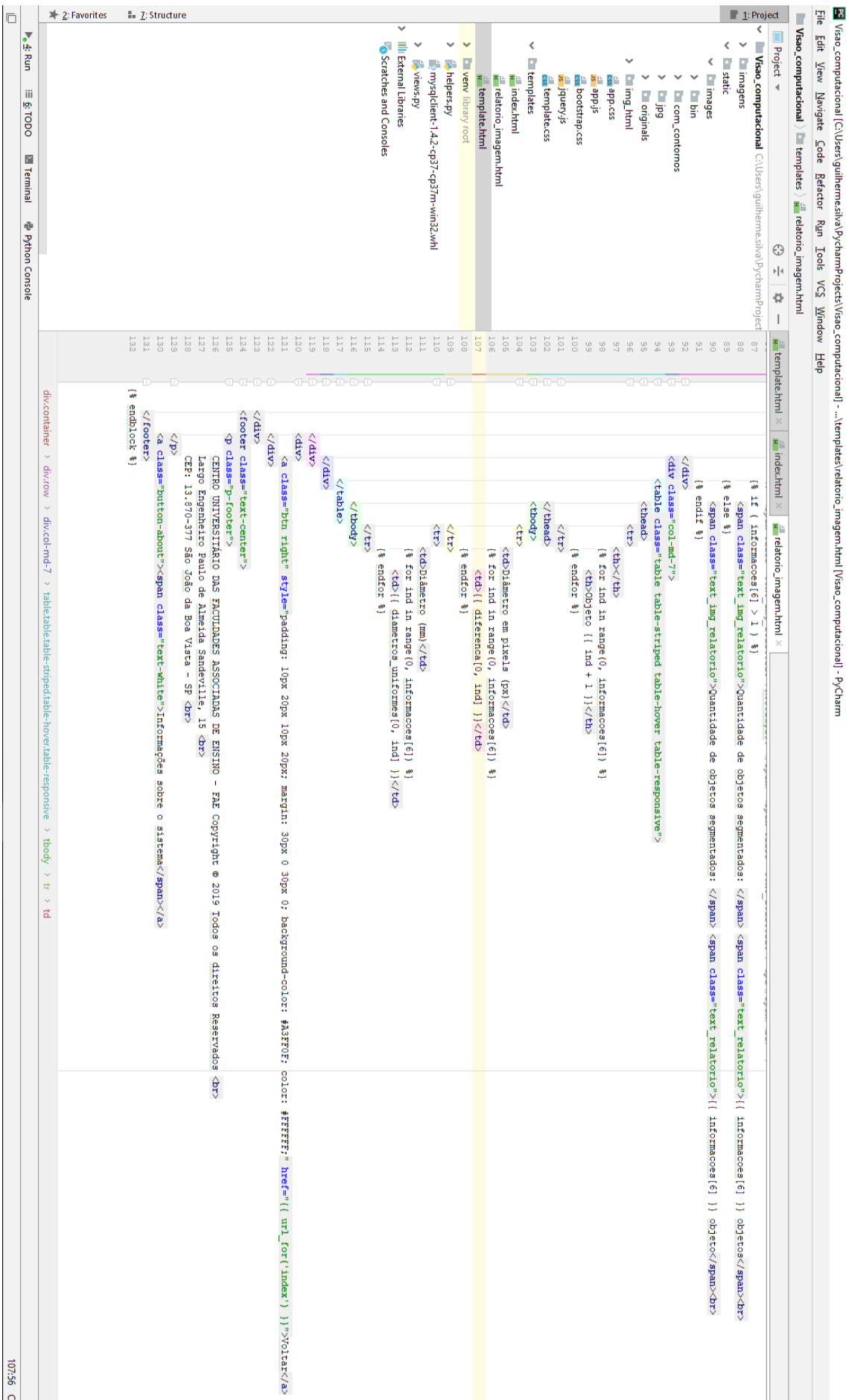




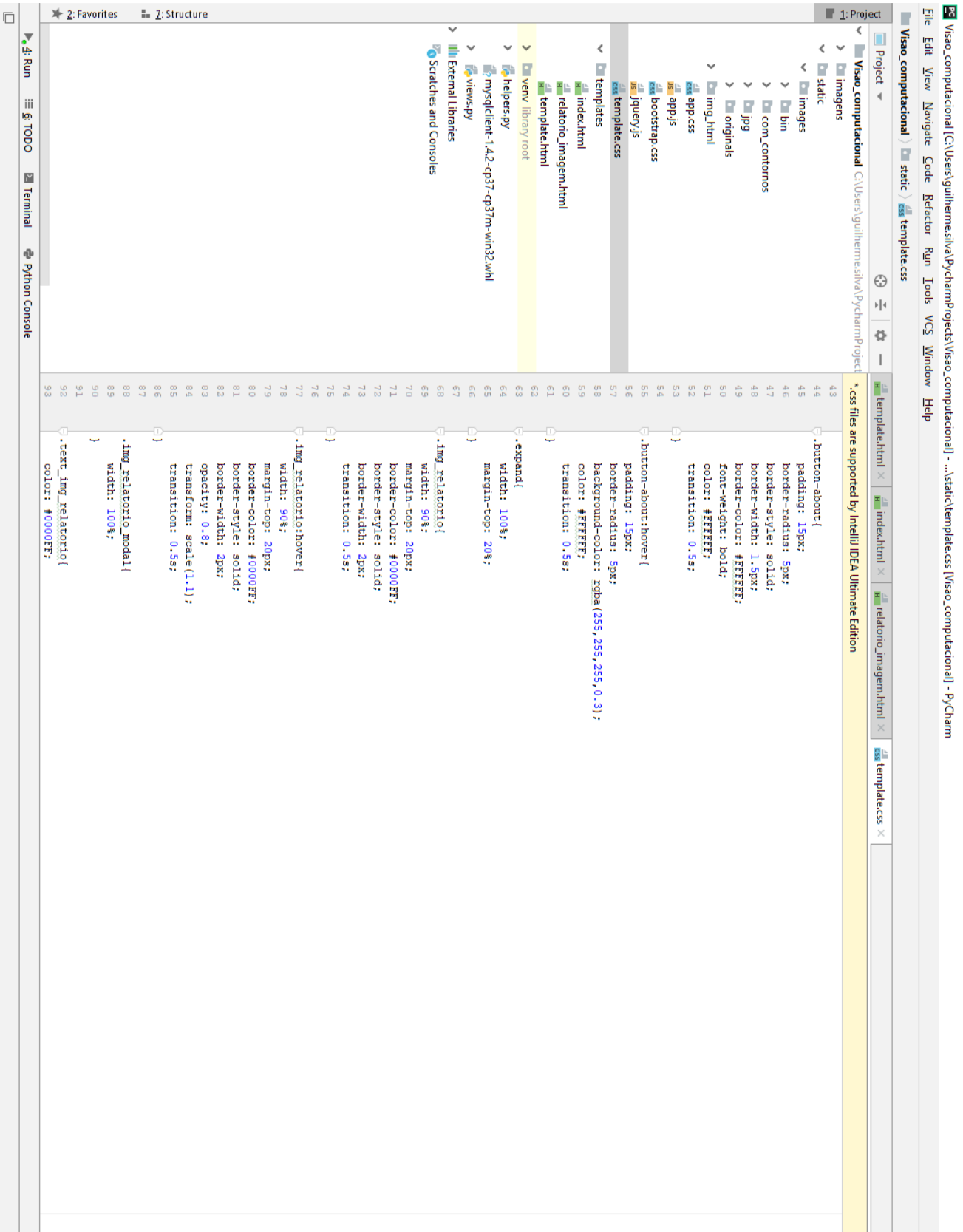


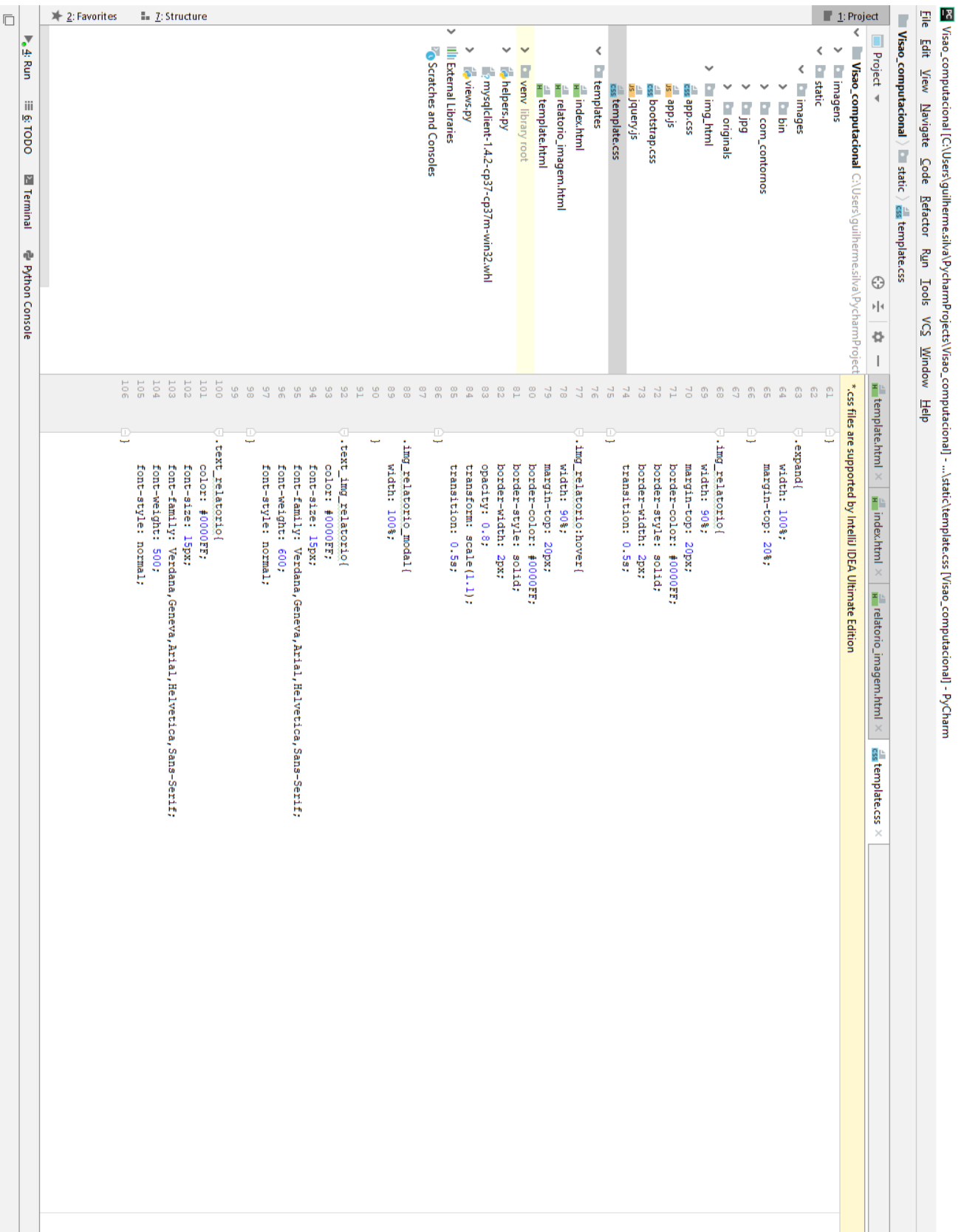






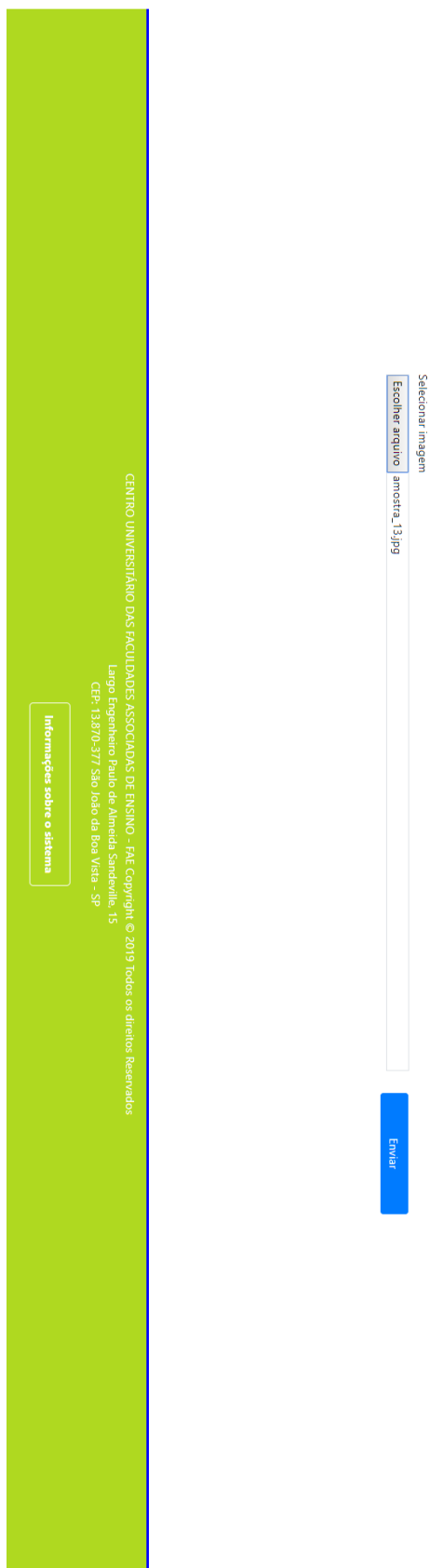
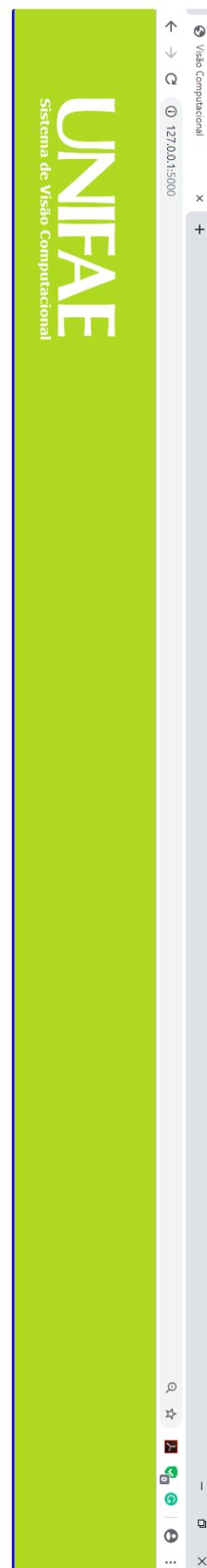






6.4. Telas gráficas

Páginas Web (telas gráficas) resultantes do desenvolvimento do sistema.




Visão Computacional

127.0.0.1:5000/relatorio_imagem

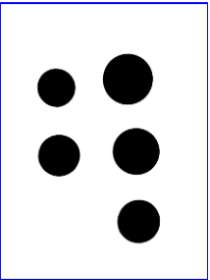
UNIFAE

Relatório dimensional


Original



Binária



Objetos segmentados



Nome do arquivo: amostra_13.jpg

Formato: JPEG

Quantidade de linhas: 3474 linhas

Quantidade de colunas: 4632 colunas

Quantidade de pixels: 16091568,0 pixels

Quantidade de canais de cores: 3 canais

Padrão das cores: RGB

Quantidade de objetos segmentados: 6 objetos

	Objeto 1	Objeto 2	Objeto 3	Objeto 4	Objeto 5	Objeto 6
Dímetro em pixels (px)	4631	647	699	859	786	731
Dímetro (mm)	143,15	20,0	21,61	26,55	24,3	22,6

Voltar

CENTRO UNIVERSITÁRIO DAS FACULDADES ASSOCIADAS DE ENSINO - FAE Copyright © 2019 Todos os direitos Reservados

Largo Engenheiro Paulo de Almeida Sandeville, 15

CEP: 13.870-377 São João da Boa Vista - SP

Informações sobre o sistema

