UNVERSIDADE FEDERAL DE UBERLÂNDIA

FACULDADE DE ENGENHARIA MECÂNICA

LUCAS AUGUSTO GONZAGA

APLICAÇÕES DA VISÃO COMPUTACIONAL

UTILIZANDO PYTHON

Uberlândia - MG

2017

LUCAS AUGUSTO GONZAGA

APLICAÇÕES DA VISÃO COMPUTACIONAL UTILIZANDO PYTHON

Projeto de Conclusão de Curso apresentado ao Curso de Graduação em Engenharia Mecatrônica da Universidade Federal de Uberlândia, como parte dos requisitos para obtenção do título de **BACHAREL EM ENGENHARIA MECATRÔNICA.**

Áreas de concentração: Robótica, Programação Aplicada a Engenharia, Visão Computacional

Orientador: Prof. Dr. Luciano Vieira Lima

Uberlândia – MG
2017

APLICAÇÕES DA VISÃO COMPUTACIONAL UTILIZANDO PYTHON

Projeto de Conclusão de Curso **APROVADO** pelo Programa de Graduação em Engenharia Mecatrônica da Universidade Federal de Uberlândia.

Áreas de concentração: Robótica, Programação Aplicada a Engenharia, Visão Computacional

Banca Examinadora:

____

Prof. Dr. Luciano Vieira Lima

____

Profa. Dra. Vera Lúcia D. S. Franco

____

Prof. Dr. Keiji Yamanaka

UBERLÂNDIA – MG
2017

AGRADECIMENTOS

Agradeço primeiramente à Deus, pelo dom da vida.

Agradeço à minha Patrícia Augusta de Faria por tudo que me ensinou, pelo exemplo de pessoa, pelo enorme amor e dedicação à família.

Agradeço ao meu pai Alexandre Luiz Gonzaga pelo suporte, pelo amor, pela motivação, pelo exemplo de pessoa, pelos conselhos.

Ao orientador, professor Luciano Vieira, pela confiança, pelo empenho, pelos conselhos e por me ajudar a tornar este trabalho possível.

À professora Vera Lúcia D. Sousa Franco pelos conselhos, pelas conversas e pela confiança.

À secretária Ivone Santos de Almeida, da coordenação do curso de Engenharia Mecatrônica, pelos conselhos e conversas.

Aos meus amigos, por estarem sempre presente, me motivando e me apoiando em todos os momentos.

GONZAGA, Lucas Augusto, APLICAÇÕES DA VISÃO COMPUTACIONAL UTILIZANDO PYTHON, 2017, Monografia de conclusão de curso, Laboratório de Robótica e Inteligência Artificial, Universidade Federal de Uberlândia, MG, Brasil.

RESUMO

Neste trabalho são apresentadas várias técnicas que são utilizadas em aplicações da Visão Computacional utilizando a linguagem PYTHON, começando com uma introdução geral ao assunto e depois seguindo com as técnicas e por fim algumas aplicações. Estas começam em níveis básicos para que alunos que possuam interesse, mas não possuem muita experiência com a linguagem, possa começar a trabalhar na área. É desejável que a pessoa que use este trabalho, possua alguma noção de programação, para melhor entendimento dos procedimentos. Os códigos apresentados podem ser aplicados em qualquer máquina que possua PYHTON, mas todos os códigos aqui apresentados foram testados em uma máquina com UBUNTU 4.0 instalado, além das bibliotecas de OpenCV, NumPy, SciPy. Foi utilizado o PYTHON 2.7 para as aplicações neste trabalho. Aplicações como desenho de formas, identificação de formas, histogramas, detector de movimentos entre outras aplicações podem ser encontradas aqui. Anexado ao trabalho existe todos os códigos utilizados neste trabalho.

Palavras chaves: Visão Computacional, Python, Códigos Didáticos, Aplicação de Visão Computacional, sensor de movimento, Histogramas

GONZAGA, Lucas Augusto, APLICATIONS OF COMPUTER VISION USING PYTHON, 2017, Conclusion Course Monograph, Robotics and IA Laboratory, Federal University of Uberlandia, MG, Brazil.

ABSTRACT

This paper reports many techniques and applications of Computer Vision using the PYTHON language, starting with an introduction of the subject and then going for the code techniques and applications. They start on basics levels, for those who have interest in the area, but do not have many experience with the language, can start working on it. It is desirable that the person who use this paper have some notion of programming to help the understanding of the subject. The codes presented here, can be created in any machine who can use PYTHON and all codes here were tested in a machine with the following items installed: UBUNTO 4.0, OpenCV, NumPy, SciPy libraries. For all applications, was used PYTHON 2.7. Applications as shape drawing, shape identification, histograms, movement sensor and others can be found in this paper. Annexed to the paper are all the codes used on it.

Keywords: Computer Vision, Python, Didactic Codes, Computer Vision applications, ´movement sensor, Histograms

LISTA DE FIGURAS

Figura 1- Reconhecimento Facial 11
Figura 2- Sistema de Segurança........ 12
Figura 3- Kinect 12
Figura 4- Linhas iniciais do código 1 14
Figura 5- Parte final do código 1 15
Figura 6- Salvando uma imagem 15
Figura 7 - Linha de comando 15
Figura 8- Sistema de coordenadas 17
Figura 9- Início do código 2 17
Figura 10- Acessando pixels 18
Figura 11- Selecionando um conjunto de pixels 18
Figura 12 - Comando para o código 2 19
Figura 13- Linhas iniciais do código 3 19
Figura 14-Linhas de código para desenhos 20
Figura 15- Criando retângulos 20
Figura 16- Desenhando círculos 21
Figura 17 –Código para realizar Translação 22
Figura 18- Inicio do código 5 23
Figura 19- Aplicando a Translação 23
Figura 20- Código para rotação 24
Figura 21- Adicionando rotação ao imultis 25
Figura 22- Rotacionando a imagem 25
Figura 23 – Redimensionamento 26
Figura 24- Redimensionamento no imultis 27
Figura 25- Imagem após a técnica de giro 28
Figura 26- Código para realizar a técnica Giro 28
Figura 27- Escolhendo a direção de giro 28
Figura 28 - Código para realizar a técnica Corte 29
Figura 29 - arithmetic.py 30
Figura 30- Efeito da adição e subtração de pixels em imagens 31
Figura 31- Resultado da adição e subtração de pixels em uma imagem 31

Figura 32 - bitwise.py 32
Figura 33- Imagens criadas.. 32
Figura 34- Operações Binárias 33
Figura 35 - Resultado das operações Bitwise 34
Figura 36- Criação e aplicação de máscara 34
Figura 37- código masking.py 35
Figura 38- Máscara circular 35
Figura 39- Resultado da máscara circular 36
Figura 40- split_and_merging.py 36
Figura 41- Resultado da separação de canais 37
Figura 42- Separação por cores 37
Figura 43 - Separação por cores 37
Figura 44- grayscale_histogram.py 39
Figura 45- Criando histograma 39
Figura 46- Histograma pronto 40
Figura 47 - Criando um histograma colorido 40
Figura 48- Resultado dos histogramas coloridos 41
Figura 49- Histograma multidimensional 42
Figura 50- Resultado do histograma multidimensional 43
Figura 51- Borrando a imagem usando a média 43
Figura 52- Imagem borrada utilizando um kernel de 3 x 3 (esquerda) e um 5 x 5 (meio) e um 7 x 7(direita) 44
Figura 53 - Borrando utilizando o método Guassiano 44
Figura 54- código para usar Gaussian blurring 44
Figura 55- Implementação do método da mediana 45
Figura 56- Resultado do método da mediana 45
Figura 57- Método bilateral 46
Figura 58- Resultado do método bilateral 46
Figura 59- Aplicação do Thresholding 47
Figura 60- Thresholding 47
Figura 61- Resultado do Thresholding 48
Figura 62- Thresholding adaptável 49
Figura 63 - Resultado dos Thresholdings 50

Figura 64- otsu_and_riddler.py 50
Figura 65- Aplicando o thresholding 51
Figura 66- A esquerda a figura original, ao meio o método de Otsu e a direita o método de Riddler-Calvard 51
Figura 67- sobel_and_laplacian.py 52
Figura 68- Resultado utilizando Laplaciano 53
Figura 69- Gradiente de Sobel 53
Figura 70- Resultados usando Sobel 54
Figura 71- canny.py 55
Figura 72- Resultados da detecção Canny 55
Figura 73 - counting_coins.py 56
Figura 74- Contagem de contornos 57
Figura 75- Resultado da contagem de modas 58
Figura 76- Selecionando apenas as moedas 58
Figura 77 - Resultado do corte da moeda 59
Figura 78- motion_detector.py 60
Figura 79 - Código para sensor de movimento 61
Figura 80- continuação do código de sensor de movimento 62
Figura 81 - Frame Delta e sua diferença com o frame inicial 62
Figura 82 - final do código de sensor de movimento 63
Figura 83- Resultados do sensor de movimento 63
Figura 84- Importação e inicialização do programa 64
Figura 85- Criação da cascata 64
Figura 86- Leitura da Imagem 64
Figura 87- Reconhecimento Facial 64
Figura 88- Loop para encontrar as faces 65
Figura 89- Apresentação do resultado 65
Figura 90 - Código para executar o Reconhecimento Facial 65
Figura 91- Imagem após a análise de reconhecimento facial 66
Figura 92- Imagem após a análise de reconhecimento facial 66

SUMÁRIO

1. INTRODUÇÃO ... 11

1.1 - OBJETIVO GERAL ... 13

1.2 - OBJETIVOS ESPECÍFICOS ... 13

1.3 - PYTHON E PACOTES NECESSÁRIOS ... 13

2. FUNDAMENTAÇÃO TEÓRICA ... 14

2.1 - SALVAR, CARREGAR E MOSTRAR ... 14

2.2 - CONCEITOS BÁSICOS SOBRE IMAGENS ... 16

2.3 - DESENHANDO FORMAS ... 19

2.4 - PROCESSAMENTO DE IMAGEM ... 21

2.5 - HISTOGRAMAS ... 38

2.6 - SUAVISAR E BORRAR ... 43

2.7 - THRESHOLDING ... 47

2.8 - GRADIENTES E DETECÇÃO DE CANTOS ... 52

2.9 - CONTORNOS ... 56

2.10 - SUBTRAÇÃO DE PLANO DE FUNDO (BACKGROUND) ... 59

2.11 - DETECÇÃO DE MOVIMENTO ... 59

3. APLICAÇÕES DA VISÃO COMPUTACIONAL ... 63

3.1 RECONHECIMENTO FACIAL ... 63

4 - CONCLUSÃO ... 67

5 - REFERÊNCIAS BIBLIOGRÁFICAS ... 68

CAPÍTULO I

1. INTRODUÇÃO

O objetivo da visão computacional é entender a história por trás de uma imagem. Para o ser humano, isso é bem simples, mas para computadores, pode ser uma tarefa extremamente complicada. (Coldewey, Devin, 2016)

Devido ao fato de que imagens estão por todas as partes, aprender sobre Visão Computacional é muito importante.

Imagens estão sendo cada vez mais utilizadas em nosso dia a dia. Redes sociais, como Facebook, Snapchat, Instagram ou vídeos do Youtube, todos eles são extremamente focados em imagens e atualmente precisamos de métodos para analisar, categorizar e quantificar essas imagens.

Um exemplo bem simples da visão computacional pode ser visto no Facebook, quando você marca alguém em uma foto, e o Facebook consegue detectar para você as faces na imagem. Isso só pode ser feito através de algoritmos de reconhecimento de formas, e com uma melhora neste algoritmo, temos um reconhecimento facial, podendo assim comparar com as várias outras imagens do site e assim identificar de quem é aquela face.

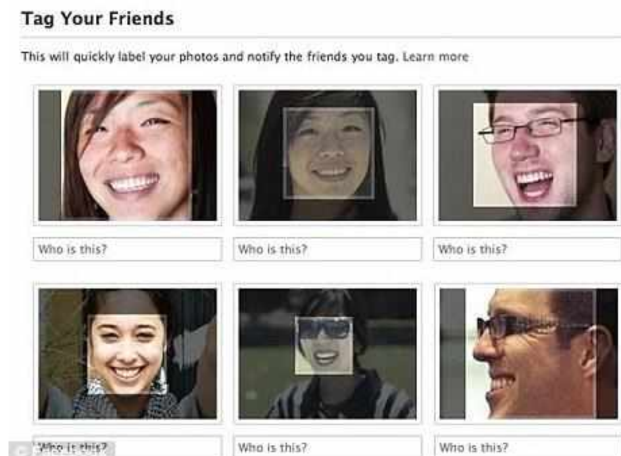

Figura 1- Reconhecimento Facial

Outras aplicações da visão computacional estão na área da segurança. Análise de vídeos de segurança para procurar suspeitos após um roubo, detectar movimento suspeito em uma área.

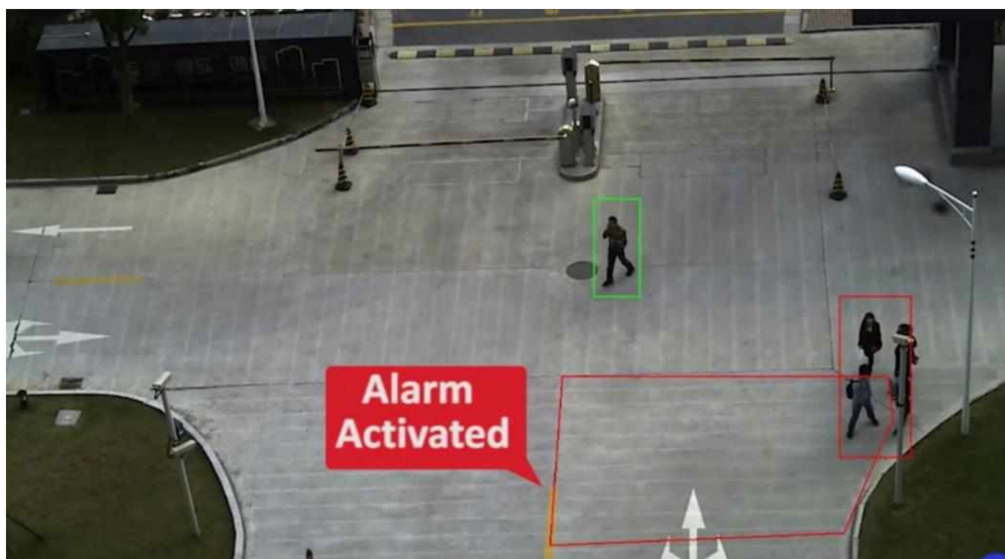

Figura 2- Sistema de Segurança

No campo da medicina, a visão computacional está sendo utilizada na análise de Raios-X, impressões de MRI e estruturas celulares. Métodos de análise para histologia mamária para evitar câncer também estão sendo utilizados, assim, o que antes era necessário um patologista com anos de experiência, pode agora ser feito rapidamente por uma máquina (ROSEBROOK, ADRIAN, 2016).

Atualmente, o sucesso mais famoso da visão computacional é o X-360 Kinect (e suas versões mais atuais). O Kinect utiliza algoritmos capazes de identificar a profundidade de uma imagem, reconhecer poses humanas.

Figura 3- Kinect

Filmes, jogos de futebol, reconhecimento de gestos (para linguagem de sinais), placas de licença de carros, medicina, cirurgia, militarismo e várias outras áreas adotam a cada momento o uso de algoritmos de visão computacional.

1.1 - OBJETIVO GERAL

O presente trabalho tem o objetivo geral de trazer uma referência para qualquer um que esteja interessado em iniciar na área da visão computacional, explicando teorias básicas utilizadas e criando exemplos práticos para aprendizagem.

1.2 - OBJETIVOS ESPECÍFICOS

O trabalho tem como objetivos específicos:

- Estudo a linguagem PYTHON
- Estudo da Visão Computacional, desde os princípios básicos de manuseio de imagens até a criação de algoritmos mais avançados para análise de imagens e vídeos.
- Criação de um material mais acessível e em língua nativa a estudantes para uso do Laboratório de Robótica e Inteligência Artificial
- Criação de um material mais acessível e em língua nativa a estudantes interessados na área da visão computacional

1.3 - PYTHON E PACOTES NECESSÁRIOS

Para se utilizar este trabalho, foi necessário a utilização de alguns pacotes e bibliotecas. O laboratório de Robótica e Inteligência Artificial da FEELT já possuia uma máquina virtual com todos os pacotes e bibliotecas instaladas e pronta para uso.

Os requisitos para uso do trabalho são:

- Linguagem Python
- Bibliotecas NumPy e SciPy
- OpenCV
- Mahotas

Nas referências deste trabalho, item 2, está o site aonde pode-se realizar o download da máquina virtual com pronta para uso, caso o acesso ao laboratório esteja difícil.

CAPÍTULO II

2 - FUNDAMENTAÇÃO TEÓRICA

Será mostrado nesta seção, técnicas utilizadas para construção de várias aplicações da visão compitacional.

2.1 - SALVAR, CARREGAR E MOSTRAR

Começando pelos princípios básicos, usando Python e o OpenCV, o código 1 do anexo tem como objetivo carregar uma imagem, mostra-la na tela e salva-la em um formato diferente.

Foi criado um arquivo como nome de *loa d_display_save.py* para iniciar.

```
1 from __future__ import print_function
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
```

Figura 4- Linhas iniciais do código 1

Inicialmente é necessário importar os pacotes que precisamos para este exemplo. A importação da função *print_function* de *__future__* é utilizada para que o código também funcione em Python 3 e facilite com possíveis atualizações.

Utilizou-se *argeparse* para analisar os argumentos da nossa linha de comando. Importamos *cv2*, nossa biblioteca para o OpenCV, esta que contém nossas funções para processamento de imagens.

Nas linhas de 5 a 8, analisamos os argumentos da linha de comando. O único argumento que precisamos é *–image:* “o diretório da imagem no disco rígido”. Então, salvamos os argumentos analisados.

```
 9 image = cv2.imread(args["image"])
10 print("width: {} pixels".format(image.shape[1]))
11 print("height: {} pixels".format(image.shape[0]))
12 print("channels: {}".format(image.shape[2]))
13
14 cv2.imshow("Image", image)
15 cv2.waitKey(0)
```

Figura 5- Parte final do código 1

Após adquirida a localização de nossa imagem, foi carregada a imagem utilizando a função *cv2.imread* na linha 8. Ela retornará uma matriz NumPy representando a imagem.

Nas linhas de 10 a 12, foi examinado as dimensões da imagem. Por elas serem representadas em uma matriz, foi utilizado o atributo *shape* para examinar sua altura, largura e número de canais.

Finalmente, nas linhas 14 e 15, é mostrado a imagem na tela. O primeiro parâmetro é uma string, o "nome" da janela. O segundo parâmetro é uma referência a imagem que foi carregada na linha 9. Para finalizar, usamos *cv2.waitKey(0)* para rodar o código até que apertamos uma tecla. O parâmetro 0 indica que qualquer tecla que apertemos vai fazer com que o código avance.

Finalmente, foi salvo a imagem no formato JPG.

```
16    cv2.imwrite("newimage.jpg", image)
```

Figura 6- Salvando uma imagem

Foram colocados o arquivo final (primeiro argumento) e a imagem que a ser salva (segundo argumento).

Para rodar o código, bastou abrir o terminal de comando e executar a linha de comando abaixo:

```
$ python load_display_save.py --image ../images/trex.png
```

Figura 7 - Linha de comando

2.2 - CONCEITOS BÁSICOS SOBRE IMAGENS

Uma imagem é composta por pixels. Eles são os blocos de construção de uma imagem. Não existe nada menor que um pixel. Se tive1rmos uma imagem com resolução de 500 x 300, isso quer dizer que nossa imagem possui 500 linhas e 300 colunas de pixels ou seja, 500 x 300 = 150.000 pixel na imagem.

A maioria dos pixels são representados de duas formas: coloridos ou tom de cinza. Em uma imagem em tom de cinza (grayscale), cada pixel possui um valor de 0 a 255, aonde 0 corresponde ao "preto" e 255 ao "branco". Os valores entre 0 e 255 representam tons de cinza, aonde os valores mais próximos ao 0 são mais escuros que os mais próximos ao 255.

Pixels coloridos são geralmente representados em um espaço RGB. Um para a componente Vermelho (R – Red), um para o Verde (G – Green) e outro para o Azul (B – Blue). Outros espaços para cor existem, mas o RGB é o mais básico.

Cada uma das cores é representada por um número inteiro de 0 a 255, que indica "quanto" dessa cor existe. Geralmente é utilizado inteiros de 8-bits para representar as intensidades de cores.

Combina-se então esses valores em uma tupla RGB na forma (vermelho,verde, azul). Esta tupla representará nossa cor. A cor branca por exemplo, seria criada da seguinte maneira (255, 255, 255) e a cor preta seria (0 ,0 ,0). Para criar uma cor pura, como por exemplo vermelho puro, colocaríamos (255, 0, 0).

Algumas combinações de cores são:

- Preto: (0, 0, 0)
- Branco: (255, 255, 255)
- Vermelho (255, 0, 0)
- Verde (0, 255, 0)
- Azul (0, 0, 255)
- Aqua (0, 255, 255)
- Rosa: (255, 0, 255)
- Marrom: (128, 0, 0)

- Azul Escuro: (0, 0 ,128)

Pode-se então criar um sistema de coordenadas nas imagens. O ponto (0,0) corresponderia ao ponto da esqueda-superior da nossa imagem. Ao deslocar-se para baixo e para direita, os valores de x e y aumentam.

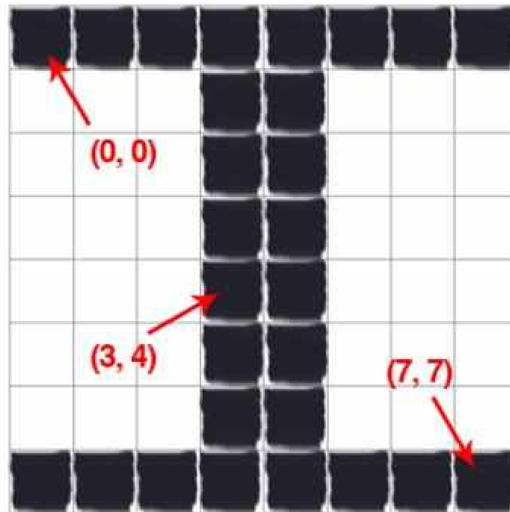

Figura 8- Sistema de coordenadas

A linguagem PYTHON é zero-indexada, ou seja, Começa-se sempre contando à partir do zero.

Assim, podemos construir um código capaz de manipular pixels em uma imagem. Criaremos um arquivo com o nome *getting_and_setting.py*.

```
1  from __future__ import print_function
2  import argparse
3  import cv2
4
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
```

Figura 9- Início do código 2

As linhas de 1 a 8 geralmente se repetirão, pois, o processo de carregar e mostrar a imagem sempre será utilizado. As linhas 10 e 11 realizam o carregamento da imagem.

Para acessar um determinado pixel, precisamos fornecer as coordenadas x e y do pixel que queremos. Algo importante a ser lembrado é que o OpenCV salva nossas tuplas RGB em ordem inversa. Ela salva as tuplas na forma (Azul, Verde, Vermelho).

```
12 (b, g, r) = image[0, 0]
13 print("Pixel at (0, 0) - Red: {}, Green: {}, Blue: {}".format(r,
       g, b))
14
15 image[0, 0] = (0, 0, 255)
16 (b, g, r) = image[0, 0]
17 print("Pixel at (0, 0) - Red: {}, Green: {}, Blue: {}".format(r,
       g, b))
```

Figura 10- Acessando pixels

Na linha 12, foi pego o pixel na posição (0, 0). O pixel é representado por uma tupla. Na linha 13, acessamos os valores das cores do pixel.

Na linha 15, foi manipulado o pixel da imagem, trocando o valor de suas cores, deixando puro azul. Após realizar isso, nas linhas 16 e 17 nós pegamos os valores novamente para confirmar que realmente mudamos o pixel.

Assim, podemos utilizar a mesma ideia para pegar não um pixel, mas um conjunto de pixels.

```
18 corner = image[0:100, 0:100]
19 cv2.imshow("Corner", corner)
20
21 image[0:100, 0:100] = (0, 255, 0)
22
23 cv2.imshow("Updated", image)
24 cv2.waitKey(0)
```

Figura 11- Selecionando um conjunto de pixels

Na linha 18, foi pego 100 x 100 pixels da imagem, neste caso, o canto superior-esquerdo da imagem. Para pegar um pedaço da imagem, NumPy espera que seja indexado quatro elementos:

- Y inicial: O valor y do pixel inicial, neste caso, 0

- Y final: O valor y do pixel final, neste caso, 100
- X inicial: O valor x do pixel inicial, neste caso, 0
- X final: O valor de x do pixel final, neste caso, 100

Na linha 19 foi mostrado o resultado do *cropping* (será explicado mais à frente no trabalho).

Na linha 21, foi acessado novamente a região, mas agora trocando a cor para um verde puro, onde foi observado o resultado utilizando as linhas 23 e 24.

Para exercutar o código, basta seguir o comando abaixo:

```
$ python getting_and_setting.py --image ../images/trex.png
```

Figura 12 - Comando para o código 2

2.3 - DESENHANDO FORMAS

Inicialmente foi necessário criar uma tela aonde foi realizado os desenhos. Até, sempre carregamos uma imagem para trabalhar, mas também poderíamos criar um espaço para o desenho. Foi criado o arquivo *drawing.py* para este exemplo.

```
1 import numpy as np
2 import cv2
3
4 canvas = np.zeros((300, 300, 3), dtype = "uint8")
```

Figura 13- Linhas iniciais do código 3

Nas linhas 1 e 2, foi importado as bibliotecas necessárias. Foi criado um atalho para o numpy, o redutor “np”. Na linha 4 foi iniciada nossa imagem. Criou-se uma matriz NumPy usando *np.zeros* com 300 colunas e 300 linhas, ou seja, 300 x 300 pixels. Alocou-se três espaços para os canais, um para o Vermelho, Verde e Azul. Este método inicia a matriz com valores zero.

Como foi representada uma imagem em RGB que utiliza inteiros entre 0 e 255, usaremos inteiros de 8-bits, ou seja, *uint8*.

Com o espaço criado, pode-se iniciar os desenhos.

```
5  green = (0, 255, 0)
6  cv2.line(canvas, (0, 0), (300, 300), green)
7  cv2.imshow("Canvas", canvas)
8  cv2.waitKey(0)
9
10 red = (0, 0, 255)
11 cv2.line(canvas, (300, 0), (0, 300), red, 3)
12 cv2.imshow("Canvas", canvas)
13 cv2.waitKey(0)
```

Figura 14-Linhas de código para desenhos

Na linha 5, foi definido a tupla para o verde. Então, desenhou-se uma linha verde do ponto (0,0) até o ponto (300, 300) na linha 6.

Para desenhar uma linha, utilizou-se a função *cv2.line*. O primeiro argumento para essa função é a imagem em que iremos desenhar. Em seguida foram colocados o ponto inicial e o ponto final da reta e por último, a cor. Linhas 7 e 8 mostram o resultado na tela e deixa o programa em espera por um aperto de tecla para continuar.

Nas linhas 10 a 13, deve-se levar em consideração a espessura da linha. Foi definido uma tupla vermelha, desenhou-se uma linha e, como último argumento, colocou-se o número 3, que define a espessura da linha em 3 pixels.

Para realizar o desenho de um retângulo, foi utilizado outra função.

```
14 cv2.rectangle(canvas, (10, 10), (60, 60), green)
15 cv2.imshow("Canvas", canvas)
16 cv2.waitKey(0)
17
18 cv2.rectangle(canvas, (50, 200), (200, 225), red, 5)
19 cv2.imshow("Canvas", canvas)
20 cv2.waitKey(0)
21
22 blue = (255, 0, 0)
23 cv2.rectangle(canvas, (200, 50), (225, 125), blue, -1)
24 cv2.imshow("Canvas", canvas)
25 cv2.waitKey(0)
```

Figura 15- Criando retângulos

Na linha 14 utilizou-se a função *cv2.rectangle* que atua igual ao *cv2.line*, construindo um retângulo. O primeiro argumento é a imagem aonde foi desenhada, o segundo é o ponto inicial (x,y), no caso, o ponto (10, 10). O terceiro argumento é o ponto final (x, y), no caso, (60,

60), definindo uma região 50 x 50 pixels. O último argumento é a cor. Na linha 18, foi repetido o processo, mas agora levando em conta a espessura da linha do retângulo, que foi ajustado para uma espessura de 5 pixels. Para preencher o retângulo, basta indicar espessuras negativas.

Na linha 23, desenhou-se um retângulo sólido. Cor azul pura, iniciando em (200, 50) e terminando em (225, 125). Colocando o valor da espessura em -1, o retângulo ficará sólido.

Para desenhar círculos, foi utilizado uma nova função com parâmetros um pouco diferentes.

```
26 canvas = np.zeros((300, 300, 3), dtype = "uint8")
27 (centerX, centerY) = (canvas.shape[1] // 2, canvas.shape[0] // 2)
28 white = (255, 255, 255)
29
30 for r in range(0, 175, 25):
31     cv2.circle(canvas, (centerX, centerY), r, white)
32
33 cv2.imshow("Canvas", canvas)
34 cv2.waitKey(0)
```

Figura 16- Desenhando círculos

Na linha 26, reiniciou-se a tela. Na linha 27, calculou-se duas variáveis, *centerX* e *centerY*, Essas duas variáveis representam (x,y) coordenadas do centro da imagem. Basta utilizar o tamanho da imagem e dividir por 2 para encontrar o centro. Na linha 28 define-se um pixel branco.

Na linha 30, foi criado um loop começando em 0, indo até 175, variando em 25. Na linha 31, desenha-se o círculo. O primeiro parâmetro é a imagem. O segundo é o centro da imagem e o terceiro é o raio do círculo que queremos desenhar e por último, a cor. Linhas 33 e 34 mostramos o resultado na tela.

2.4 - PROCESSAMENTO DE IMAGEM

Primeiramente será mostrado as técnicas básicas de transformação em imagens. Elas são: Translação, Rotação, redimensionamento, giro e corte. Depois será mostrado técnicas mais avançadas como aritmética em imagens, operações bitwise e máscara. E por último, como separar imagens em seus respectivos canais e depois uni-las novamente.

O primeiro método é a translação, que nada mais é que mudar a posição da imagem nos eixos x e y. Usando a translação, podemos movimentar a imagem para cima, baixo, esquerda ou direita. Criaremos o arquivo *translation.py*

```
1  import numpy as np
2  import argparse
3  import imutils
4  import cv2
5
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required = True,
8      help = "Path to the image")
9  args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
13
14 M = np.float32([[1, 0, 25], [0, 1, 50]])
15 shifted = cv2.warpAffine(image, M, (image.shape[1], image.shape
       [0]))
16 cv2.imshow("Shifted Down and Right", shifted)
17
18 M = np.float32([[1, 0, -50], [0, 1, -90]])
19 shifted = cv2.warpAffine(image, M, (image.shape[1], image.shape
       [0]))
20 cv2.imshow("Shifted Up and Left", shifted)
```

Figura 17 –Código para realizar Translação

Linhas 1 a 4 temos a inicialização das bibliotecas. A única nova é a *imutils*. Esta não é uma biblioteca do NumPy ou do OpenCV. Ela será uma biblioteca que criaremos com as funções básicas como translação, rotação e redimensionamento. Linhas 6 a 12 encontramos a imagem e a carregamos.

Linhas 14 a 16 é que realizamos a translação. Primeiro definimos nossa matriz de translação M. Esta matriz nos diz quantos pixels para esquerda, direita, cima ou baixo a imagem será transladada.

A primeira linha da matriz é [1, 0, Tx], onde Tx é o número de pixels que transladaremos para esquerda ou direita. Valores de Tx negativos levarão a imagem para esquerda e positivos para direita. A segunda linha da matriz é [1, 0, Ty] onde Ty é o número de pixels que transladaremos a imagem para cima ou para baixo. Valores negativos de Ty levarão a imagem

para cima e valores positivos para baixo. Na linha 14, o Tx = 25 e Ty = 50, ou seja, a imagem irá transladar 25 pixels para direita e 50 pixels para baixo.

Com a matriz de translação criada, na linha 15 foi utilizado a função *cv2.warpAffine* para transladar. O primeiro argumqento é a imagem que queremos transladar e o segundo é nossa matriz M de translação. O terceiro argumento são a altura e comprimento da imagem. Na linha 16 temos o resultado.

Nas linhas de 18 a 20, é feita outra translação. Colocamos o Tx =-50 e o Ty = -90, ou seja, a imagem irá transladar 50 pixels para a esquerda e 90 pixels para cima.

Vamos então criar um arquivo chamado *imutils.py* para que possamos realizar essas funções mais rapidamente.

```
1 import numpy as np
2 import cv2
3
4 def translate(image, x, y):
5     M = np.float32([[1, 0, x], [0, 1, y]])
6     shifted = cv2.warpAffine(image, M, (image.shape[1], image.
          shape[0]))
7
8     return shifted
```

Figura 18- Inicio do código 5

A função *translate* necessitará de três parâmetros. A imagem a ser transladada, o número de pixels a se transladar no eixo x e o número de pixels a se transladar no eixo y.

Na linha 5 da figura 18, definimos nossa matriz M de translação e aplicamos a translação na linha 6 da figura 18. Na linha 8 da figura 18 temos o resultado da translação.

Voltando ao código principal, podemos aplicar a translação agora.

```
21 shifted = imutils.translate(image, 0, 100)
22 cv2.imshow("Shifted Down", shifted)
23 cv2.waitKey(0)
```

Figura 19- Aplicando a Translação

Utilizando essa nova função, podemos transladar uma imagem em 100 pixels para baixo apenas utilizando uma linha de código.

A segunda técnica é a rotação. Podemos utilizar essa técnica para rotacionar uma imagem em um anglo Θ. Também será implementado a função *rotate* a biblioteca *imutils*. Foi criado o código *rotate.py*, mostrado na figura

```
1  import numpy as np
2  import argparse
3  import imutils
4  import cv2
5
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required = True,
8      help = "Path to the image")
9  args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
13
14 (h, w) = image.shape[:2]
15 center = (w // 2, h // 2)
16
17 M = cv2.getRotationMatrix2D(center, 45, 1.0)
18 rotated = cv2.warpAffine(image, M, (w, h))
19 cv2.imshow("Rotated by 45 Degrees", rotated)
20
21 M = cv2.getRotationMatrix2D(center, -90, 1.0)
```

Figura 20- Código para rotação

Linhas 1 a 4 importam os pacotes necessários. Linhas 6 a 12 fazem o carregamento da imagem e mostramos ela na tela. Para rotacionar, é necessário dizer o ponto em que rotacionaremos a imagem. Nas linhas 14 e 15 pegamos os valores da altura e comprimento da imagem, dividimos por dois, para encontrarmos o centro da imagem. É usado a divisão de inteiros "//" para garantir que o valor final seja inteiro.

Devemos agora estabelecer uma matriz de rotação. Ao invés de criar a matriz manualmente, podemos utilizar a função *cv2.getRotationMatrix2D* como é feito na linha 17. Esta função requer três argumentos: O ponto em que queremos rotacionar a imagem (nesse caso o centro da imagem), o ângulo que rotacionaremos, o valor em graus (neste caso rotacionaremos em 45°) e o último parâmetro é a escala. Foi utilizado uma escala de 1.0, mas se fosse necessário duplicar a imagem, basta mudar o valor para 2.0. Para diminui-la em 50%, basta mudar o valor para 0.5.

Com a matriz de rotação pronta, podemos aplicar a rotação utilizando *cv2.warpAffine* na linha 18. O primeiro argumento para essa função é a imagem a ser rotacionada, o segundo a matriz de rotação e as dimensões da imagem. A linha 19 mostra o resultado.

Nas linhas 21 a 23, foi realizado outra rotação. O código é idêntico ao das linhas 17 a 19. Mas dessa vez foi rotacionado em -90° ao invés de 45°. Adicionando o código de rotação ao código geral *imultis*.

```
27 def rotate(image, angle, center = None, scale = 1.0):
28     (h, w) = image.shape[:2]
29 
30     if center is None:
31         center = (w / 2, h / 2)
32 
33     M = cv2.getRotationMatrix2D(center, angle, scale)
34     rotated = cv2.warpAffine(image, M, (w, h))
35 
36     return rotated
```

Figura 21- Adicionando rotação ao imultis

A função requer quatro parâmetros. O primeiro é a imagem. O segundo é o ângulo que a ser rotacionado. Deixando duas opções, *center* ou *scale*. O parâmetro centro irá rotacionar a imagem em relação ao seu centro. Se nada for preenchido, será colocado *center* como padrão na linha 30 e 31. Por fim, o parâmetro *scale* controla a escala que a imagem final receberá. Nas linhas 33 e 34 ocorre a rotação e na linha 36 temos o resultado.

Aplicando a função no código original:

```
24 rotated = imutils.rotate(image, 180)
25 cv2.imshow("Rotated by 180 Degrees", rotated)
26 cv2.waitKey(0)
```

Figura 22- Rotacionando a imagem

A terceira técnica é o redimensionamento. Como o próprio nome diz, foi utilizado para redimensionar a imagem. Criou-se um arquivo *resize.py* para mostrar esta técnica.

```python
import numpy as np
import argparse
import imutils
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
    help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

r = 150.0 / image.shape[1]
dim = (150, int(image.shape[0] * r))

resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
cv2.imshow("Resized (Width)", resized)
```

Figura 23 – Redimensionamento

Linhas 1 a 12 realizam os mesmos processos que já foi visto anteriormente. Para redimensionar, precisa-se de um valor que seria a relação entre altura e comprimento da imagem. Isso é necessário para que a imagem possamos redimensionar a imagem de maneira correta. Na linha 14 calculamos este valor. Nesta linha, foi redefinido que o comprimento da nova imagem seria de 150 pixels, assim, pare ter este valor, bastou dividir os 150 pelo valor do comprimento antigo da imagem.

Tendo este valor, pode-se calcular os novos valores da dimensão da imagem, feito na linha 15. O redimensionamento é feito na linha 17. O primeiro argumento é a imagem a ser redimensionalizada e o segundo são os valores das dimensões da nova imagem. O último parâmetro é o método de *interpolation*. Este é o algoritmo que irá realizar o redimensionamento da imagem. Nos testes realizados, cv2.*INTER_AREA* foi o que teve o melhor resultado para redimensionalizar, mas podemos usar também *cv2.INTER_LINEAR, cv2.INTER_CUBIC, cv2.INTER_NEAREST.* Na linha 18 mostramos a imagem redimensionada.

Adicionando esta técnica ao código *imultis*:

```
9  def resize(image, width = None, height = None, inter = cv2.
        INTER_AREA):
10     dim = None
11     (h, w) = image.shape[:2]
12 
13     if width is None and height is None:
14         return image
15 
16     if width is None:
17         r = height / float(h)
18         dim = (int(w * r), height)
19 
20     else:
21         r = width / float(w)
22         dim = (width, int(h * r))
23 
24     resized = cv2.resize(image, dim, interpolation = inter)
25 
26     return resized
```

Figura 24- Redimensionamento no imultis

O primeiro argumento é a imagem a ser redimensionada. Depois definiu-se duas palavras-argumento, *width* e *height.* Estes argumentos não podem ser nulos, pois sem eles não é possível redimensionar a imagem. Também colocu-se *inter* que é o método de interpolação e o padrão será *cv2.INTER_AREA*.

É feito uma verificação nas linhas 13 e 14 para que haja um valor inteiro para altura e comprimento. E nas linhas 16 a 22 calcula-se o valor de redimensionamento, seja ele dado pela altura ou pelo comprimento, na linha 24 realizou-se o redimensionamento e na linha 26 é mostrado o resultado.

A quarta técnica é uma especificação da rotação. O giro é uma rotação da imagem em pontos diferentes. A figura 25 explica melhor como é feito. Criou-se o código *flipping.py*

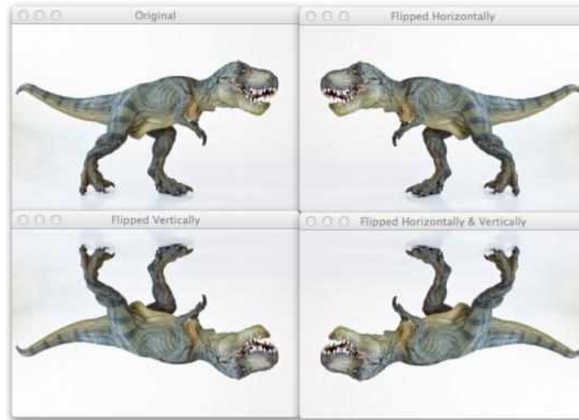

Figura 25- Imagem após a técnica de giro

```python
1  import argparse
2  import cv2
3
4  ap = argparse.ArgumentParser()
5  ap.add_argument("-i", "--image", required = True,
6      help = "Path to the image")
7  args = vars(ap.parse_args())
8
9  image = cv2.imread(args["image"])
10 cv2.imshow("Original", image)
11
12 flipped = cv2.flip(image, 1)
13 cv2.imshow("Flipped Horizontally", flipped)
14
```

Figura 26- Código para realizar a técnica Giro

Linhas 1 a 10 foi realizado os mesmos processos. Para girar uma imagem, bastou utilizar a função *cv2.flip* como foi feito na linha 12. Esta função requer dois argumentos. O primeiro argumento é a imagem a ser girada e o segundo argumento é um parâmetro para saber como será feito o giro.

```python
15 flipped = cv2.flip(image, 0)
16 cv2.imshow("Flipped Vertically", flipped)
17
18 flipped = cv2.flip(image, -1)
19 cv2.imshow("Flipped Horizontally & Vertically", flipped)
20 cv2.waitKey(0)
```

Figura 27- Escolhendo a direção de giro

Usar 1 como argumento, faz girar a imagem horizontalmente em torno do eixo y (linha 12). O argumento 0 indica que se quer girar a imagem verticalmente em relação ao eixo x (linha 15). Finalmente, utilizando um valor negativo irá girar a imagem nos dois eixos (linha 18).

A quinta técnica é o corte ou em inglês *cropping*. Quando se corta uma imagem, a idéia é remover partes da imagem que não desejadas. Já foi realizado um corte no código 2, representado na figura 6.

Criando o arquivo *crop.py* para aplicar o corte:

```
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 cropped = image[30:120 , 240:335]
14 cv2.imshow("T-Rex Face", cropped)
15 cv2.waitKey(0)
```

Figura 28 - Código para realizar a técnica Corte

As linhas iniciais são as mesmas e por isso foram omitidas aqui. Lembrando que todos os códigos completos podem ser encontrados anexados a este trabalho.

O corte se inicia a partir da linha 13. Foi escolhido matrizes NumPy retangulares, começando em (240, 30) e terminando em (355, 120). Lembrando que o NumPy tem como referência primeiro a altura e depois o comprimento. Para realizar o corte, são necessários quatro parâmetros:

- Y inicial: coordenada y do ponto inicial. Neste caso y = 30
- Y final: coordenada y do ponto final. Neste caso y = 120
- X inicial: coordenada x do ponto inicial. Neste caso x = 240
- X final: coordenada x do ponto final. Neste caso x = 335

Executando o código, realizou-se o corte na imagem.

Uma das técnicas um pouco mais avançadas é a aritmética em imagens. Somar valores ou subtrair valores entre uma imagem ou outra pode causar vários problemas se não levar em conta o espaço de cor ou tipo de dados.

Por exemplo, imagens em RGB possuem pixels entre 0 e 255 e se adicionarmos 10 ao valor de 250, teríamos que, ou realizar um processo de checagem para evitar isso ou realizar um arredondamento modular, que transformaria o 260 no valor 4. Logo, dependerá da maneira que você necessita realizar as operações. NumPy realizará esta soma através do arredondamento, enquanto o OpenCV realizará uma checagem, evitando valores acima de 255 ou abaixo de 0.

Criando o arquivo *arithmetic.py* para realizar um exemplo da técnica.

```
1 from __future__ import print_function
2 import numpy as np
3 import argparse
4 import cv2
5
6 ap = argparse.ArgumentParser()
7 ap.add_argument("-i", "--image", required = True,
8     help = "Path to the image")
9 args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 cv2.imshow("Original", image)
13
14 print("max of 255: {}".format(cv2.add(np.uint8([200]), np.uint8
      ([100]))))
15 print("min of 0: {}".format(cv2.subtract(np.uint8([50]), np.uint8
      ([100]))))
16
17 print("wrap around: {}".format(np.uint8([200]) + np.uint8([100]))
      )
18 print("wrap around: {}".format(np.uint8([50]) - np.uint8([100])))
```

Figura 29 - arithmetic.py

Nas linhas de 1 a 12 temos o procedimento padrão. Na linha 14, foi definido duas matrizes NumPy que são de inteiros de 8 bits. A primeira possui um valor de 200 e a segunda um valor de 100. Usando a função *cv2.add* foi realizado a soma das duas matrizes.

Se fosse utilizado as regras aritméticas, o valor final deveria ser 300, mas como foi somado usando o OpenCV, ele irá somar até chegar em seu valor máximo de 255, ou seja, a soma dessas matrizes, usando essa função, retornará o valor 255.

A linha 15 realiza a subtração usando a função *cv2.subtract*. Novamente, define-se duas matrizes de 8 bits com os valores de 50 e 100. Realizando a operação, teríamos que receber o valor de -50, mas o OpenCV irá entregar o valor mais baixo para ele, no caso, 0.

Se fosse usado o NumPy para realizar essas operações, teremos resultados diferentes. As linhas 17 e 18 realizam este teste.

Definiu-se duas matrizes, uma com o valor de 200 e a outra com 100. Quando realizar a soma, no NumPy, ao chegarmos ao valor de 255, reinicia a contagem a 0 e continua a partir daí, assim, recebemos o resultado de 44 nesta soma. O mesmo acontece ao subtrair dois números. Se resultar em um número negativo, ao chegar no zero, o NumPy irá reiniciar a contagem a partir do 255.

Foi realizado a subtração e adição em uma imagem.

```
19 M = np.ones(image.shape, dtype = "uint8") * 100
20 added = cv2.add(image, M)
21 cv2.imshow("Added", added)
22
23 M = np.ones(image.shape, dtype = "uint8") * 50
24 subtracted = cv2.subtract(image, M)
25 cv2.imshow("Subtracted", subtracted)
26 cv2.waitKey(0)
```

Figura 30- Efeito da adição e subtração de pixels em imagens

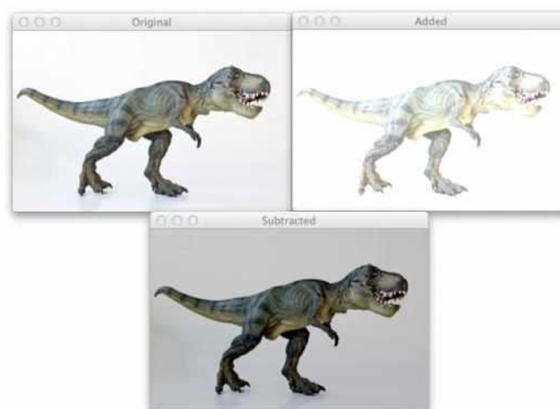

Figura 31- Resultado da adição e subtração de pixels em uma imagem

Ao analisar a figura 28, foi observado que, quando se adiciona 100 em todos os pixels de uma figura, o resultado será uma imagem mais apagada, como se o brilho tivesse aumentado e quando realizar a subtração de 100 em cada pixel, ela se torna mais nítida e escura.

Podemos também realizar operações Bitwise com imagens, isto é, aplicar as operações de E (AND), OU (OR), ou-exclusivo (XOR) e não (NOT). Estas operações trabalham de maneira binária e são representadas quando utilizamos uma escala de tons de cinza. Um pixel estará “desligado” se possuir valor zero e estará “acesso” se possuir valor maior que zero.

Foi criado o arquivo *bitwise.py* para exemplificar a técnica.

```
1  import numpy as np
2  import cv2
3
4  rectangle = np.zeros((300, 300), dtype = "uint8")
5  cv2.rectangle(rectangle, (25, 25), (275, 275), 255, -1)
6  cv2.imshow("Rectangle", rectangle)
7
8  circle = np.zeros((300, 300), dtype = "uint8")
9  cv2.circle(circle, (150, 150), 150, 255, -1)
10 cv2.imshow("Circle", circle)
```

Figura 32 - bitwise.py

Nas primeiras duas linhas, as importações necessárias são feitas. Inicializamos uma imagem de retângulo 300 x 300 na linha 4. Então desenhamos um retângulo 250 x 250 branco no centro da imagem.

Na linha 8, foi iniciado uma nova imagem contendo um círculo que é desenhado na linha 9, e centrado no centro da imagem, com raio de 150 pixels.

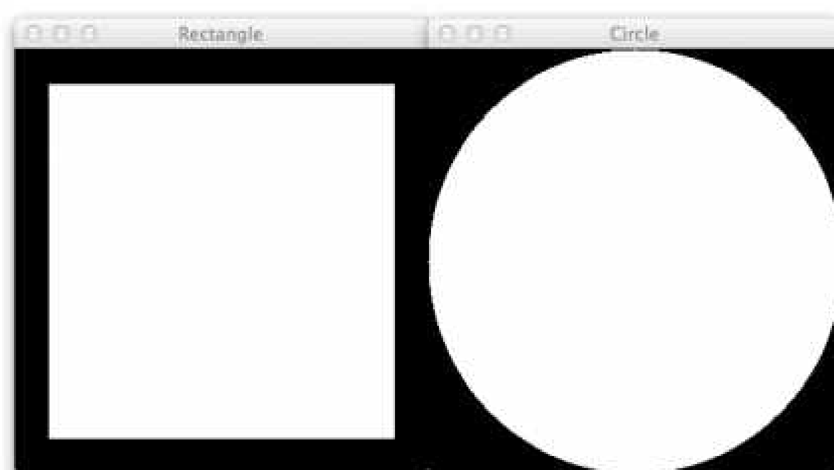

Figura 33- Imagens criadas

Foi realizado as operações binárias então com essas duas imagens.

```
11 bitwiseAnd = cv2.bitwise_and(rectangle, circle)
12 cv2.imshow("AND", bitwiseAnd)
13 cv2.waitKey(0)
14
15 bitwiseOr = cv2.bitwise_or(rectangle, circle)
16 cv2.imshow("OR", bitwiseOr)
17 cv2.waitKey(0)
18
19 bitwiseXor = cv2.bitwise_xor(rectangle, circle)
20 cv2.imshow("XOR", bitwiseXor)
21 cv2.waitKey(0)
22
23 bitwiseNot = cv2.bitwise_not(circle)
24 cv2.imshow("NOT", bitwiseNot)
25 cv2.waitKey(0)
```

Figura 34- Operações Binárias

Relembrando as operações binárias:

- E (AND): Será verdadeiro se ambos os pixels forem maiores que zero.
- OU (OR): Será verdadeiro se um ou outro pixel for maior que zero.
- OU-EXCLUSIVO (XOR): Será verdadeiro se e somente se os dois pixels forem maiores que zero, mas não se os dois forem ao mesmo tempo.
- NÃO (NOT): Irá inverter os valores de “acesso” e “desligado” dos pixels.

Na linha 11 foi aplicado a operação E usando a função cv2.*bitwise_and*. A imagem 32 mostra o resultado, na parte superior esquerda da figura. Os parâmetros utilizados são as duas imagens.

Na linha 15 foi aplicado a operação OU usando a função cv2.*bitwise_or*. A imagem 32 mostra o resultado, na parte superior direita da figura. Os parâmetros utilizados são as duas imagens.

Na linha 19 foi aplicado a operação OU-EXCLUSIVO usando a função cv2.*bitwise_xor*. A imagem 32 mostra o resultado, na parte inferior esquerda da figura. Os parâmetros utilizados são as duas imagens.

Na linha 23 foi aplicado a operação NOT usando a função cv2.*bitwise_not*. A imagem 32 mostra o resultado, na parte inferior direta da figura. O parâmetro utilizado é a imagem.

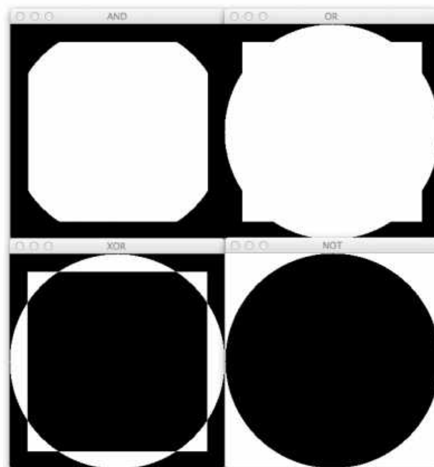

Figura 35 - Resultado das operações Bitwise

Outra técnica muito importante é a Máscara. Utilizar uma máscara nos permite focar nas porções importantes da imagem. Por exemplo, se é necessário criar um sistema para reconhecer rostos, a única parte que é interessante da foto são as partes aonde há rostos. O resto pode-se descartar. A figura 33 mostra o uso prático de uma máscara.

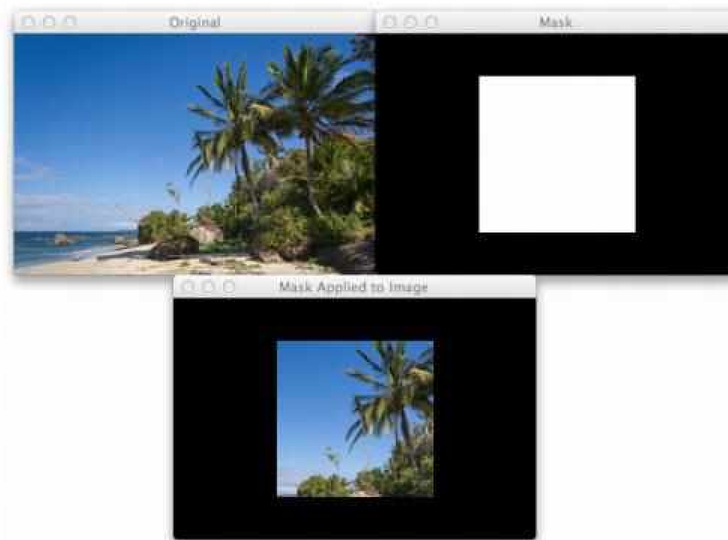

Figura 36- Criação e aplicação de máscara

O código *masking.py* mostra como foi realizado esse tipo de procedimento.

```
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 cv2.imshow("Original", image)
12
13 mask = np.zeros(image.shape[:2], dtype = "uint8")
14 (cX, cY) = (image.shape[1] // 2, image.shape[0] // 2)
15 cv2.rectangle(mask, (cX - 75, cY - 75), (cX + 75 , cY + 75), 255,
        -1)
16 cv2.imshow("Mask", mask)
17
18 masked = cv2.bitwise_and(image, image, mask = mask)
19 cv2.imshow("Mask Applied to Image", masked)
20 cv2.waitKey(0)
```

Figura 37- código masking.py

Na linha 13, foi construída uma matriz NumPy com zeros. Para desenhar o retângulo branco, foi necessário dividir os valores da altura e comprimento por dois para o centro da imagem e na linha 15 foi feito o retângulo.

Para aplicar a máscara, usou-se a função *cv2.bitwise_and*. Os dois primeiros parâmetros são a própria imagem. O retorno será verdadeiro para todos os pixels. A parte importante é a utilização da palavra-chave *mask*, assim, apenas analisando os pontos que estiverem na máscara. Neste caso, apenas os pixels dentro do quadrado branco.

Outro exemplo é realizar uma máscara circular.

```
21 mask = np.zeros(image.shape[:2], dtype = "uint8")
22 cv2.circle(mask, (cX, cY), 100, 255, -1)
23 masked = cv2.bitwise_and(image, image, mask = mask)
24 cv2.imshow("Mask", mask)
25 cv2.imshow("Mask Applied to Image", masked)
26 cv2.waitKey(0)
```

Figura 38- Máscara circular

Na linha 21, reiniciamos nossa máscara para ela ser cheia de zeros e das mesmas dimensões que a nossa imagem. Desenhamos então um círculo branco em nossa máscara, começando no centro e com raio de 100 pixels. Aplicando a máscara na linha 23, teremos os resultados abaixo.

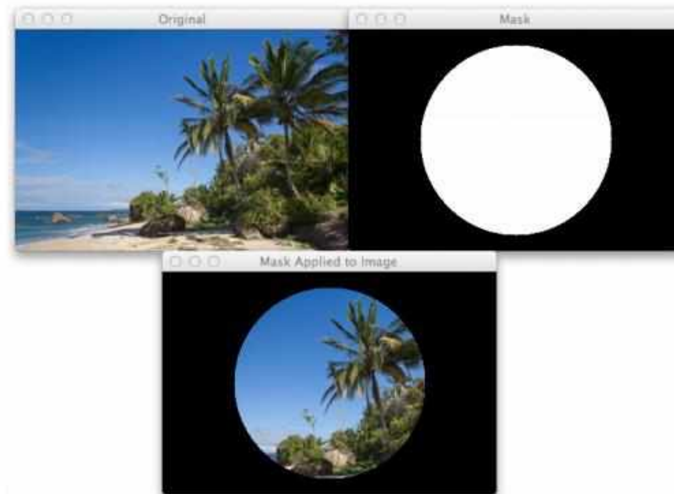

Figura 39- Resultado da máscara circular

E por último temos a separação e união de canais. As cores de uma imagem dependem de múltiplos canais: um canal, vermelho, um verde e um azul.

Para realizar a separação desses canais, usaremos a função *cv2.split* que pode ser vista na figura 40.

```
1  import numpy as np
2  import argparse
3  import cv2
4
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 (B, G, R) = cv2.split(image)
12
13 cv2.imshow("Red", R)
14 cv2.imshow("Green", G)
15 cv2.imshow("Blue", B)
16 cv2.waitKey(0)
17
18 merged = cv2.merge([B, G, R])
19 cv2.imshow("Merged", merged)
20 cv2.waitKey(0)
21 cv2.destroyAllWindows()
```

Figura 40- split_and_merging.py

As linhas de 1 a 10 importam os pacotes e carregam a imagem. Na linha 11 utilizou-se a separação de canais.

Linhas 13 a 16 mostram cada canal individualmente. O resultado está na figura abaixo:

Figura 41- Resultado da separação de canais

É possível uni-las novamente usando a função *cv2.merge*. Apenas especifica-se os canais, na ordem BGR e a função executa a união novamente (Linha 18).

Outro método de separação, é o mostrado na figura 39.

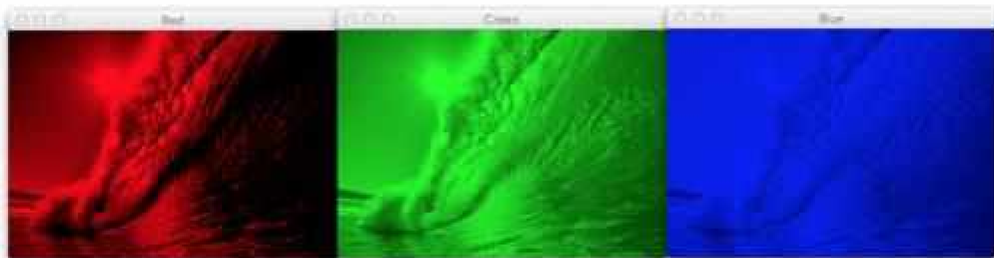

Figura 42- Separação por cores

Para realizar a separação desta maneira, basta, primeiramente executar a separação por canais e depois colocar todos os pixels, menos o da cor do canal como zero como feito na figura 43.

```
22 zeros = np.zeros(image.shape[:2], dtype = "uint8")
23 cv2.imshow("Red", cv2.merge([zeros, zeros, R]))
24 cv2.imshow("Green", cv2.merge([zeros, G, zeros]))
25 cv2.imshow("Blue", cv2.merge([B, zeros, zeros]))
26 cv2.waitKey(0)
```

Figura 43 - Separação por cores

Na linha 22 construímos uma matriz de zeros com o mesmo comprimento e altura da imagem original. Então, para realizar o canal vermelho, foi utilizado a função *cv2.merge*, mas especificando os zeros para os canais verde e azul e repetiu-se o processo para os outros canais (linhas 24 e 25).

2.5 - HISTOGRAMAS

Um histograma representa a distribuição das intensidades de pixel em uma imagem. Ele pode ser visualizado em um gráfico que nos dá uma intuição da distribuição da intensidade. Assumindo um espaço de cor RGB por exemplo, quando mostrarmos o histograma, o eixo X do gráfico servirá como nossas "caixas". Se for construído um histograma com 256 caixas, será contado o número de vezes que cada valor de pixel aparece. Mas se for utilizado apenas 2 caixas, será contado o número de vezes que o valor de pixel aparece entre [(0,128) e (128,255)]. O número de pixels que está nessa faixa será mostrado então no eixo y.

Simplesmente analisando um histograma, é possivel ter uma noção geral do contraste, brilho e da distribuição de intensidade.

Para construir um histograma, usou-se a função *cv2.calcHist*. Antes, de implementar, será mostrado como a função trabalha.

cv2.calcHist (images, channels, mask, histSize, ranges)

- **Images**: É a imagem de que será analisada o histograma.
- **Channels:** É a escolha de qual canal será utilizado. Para usar um canal de tons de cinza, selecione [0]. Para histogramas para os três canais, vermelho, verde, azul, basta selecionar [0 ,1 ,2]
- **Mask:** Se quiser aplicar uma máscara para analisar em um local específico da imagem
- **histSize:** É o número de "caixas" que será usado para computar o histograma. Uma lista é criada, um para cada canal computado. Não é necessário que todos eles sejam do mesmo tipo de dados. Por exemplo, 32 "caixas" para cada canal: [32, 32, 32]
- **Ranges:** Especifica a faixa de valores de pixel possíveis. Normalmente é de [0,256] para cada canal, mas se for utilizado outros espaços de cor, pode alterar.

Primeiramente foi realizado um histograma em tons de cinza (grayscale).

```
1 from matplotlib import pyplot as plt
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
```

Figura 44- grayscale_histogram.py

Foi importado bibliotecas e inicializou a imagem a ser tratada. Foi utilizada a biblioteca *matlplotlib* para facilitar na montagem do histograma.

```
13 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
14 cv2.imshow("Original", image)
15
16 hist = cv2.calcHist([image], [0], None, [256], [0, 256])
17
18 plt.figure()
19 plt.title("Grayscale Histogram")
20 plt.xlabel("Bins")
21 plt.ylabel("# of Pixels")
22 plt.plot(hist)
23 plt.xlim([0, 256])
24 plt.show()
25 cv2.waitKey(0)
```

Figura 45- Criando histograma

Na linha 13, convertemos o espaço de cor de RGB para grayscale. Linha 16 computamos o histograma. Selecionamos a imagem, utilizamos [0] pois grayscale possui apenas um canal, não possuímos máscara, usaremos 256 caixas e o valores podem ir de 0 a 256.

Chamamos a função *plt.plot()* para mostrar o histograma feito, que pode ser visto na figura 46.

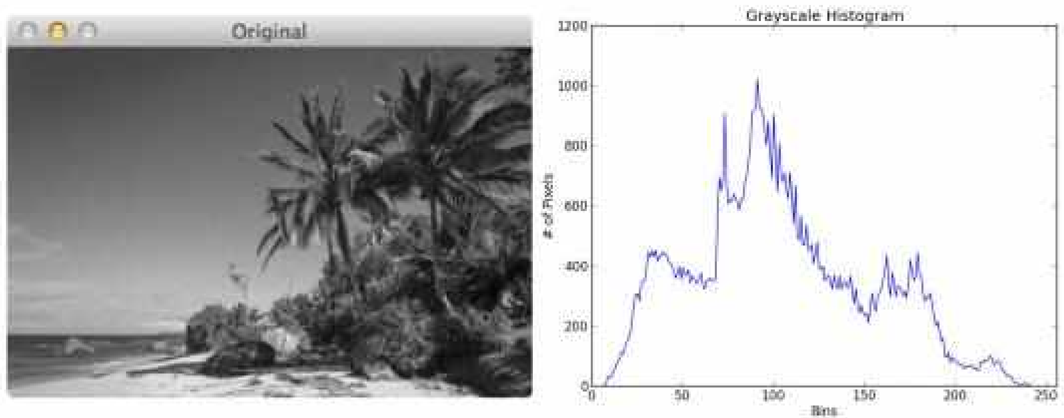

Figura 46- Histograma pronto

Repete-se o processo para o canal RGB agora. Foi criado o arquivo *color_histograms.py* para demonstrar a técnica sendo aplicada.

```
14 chans = cv2.split(image)
15 colors = ("b", "g", "r")
16 plt.figure()
17 plt.title("'Flattened' Color Histogram")
18 plt.xlabel("Bins")
19 plt.ylabel("# of Pixels")
20
21 for (chan, color) in zip(chans, colors):
22     hist = cv2.calcHist([chan], [0], None, [256], [0, 256])
23     plt.plot(hist, color = color)
24     plt.xlim([0, 256])
```

Figura 47 - Criando um histograma colorido

Primeiro foi necessário separar a imagem nos três canais, vermelho, verde e azul. Linhas 14 e 15 realizam esse processo.

As linhas 16 a 19 preparam a figura para ser mostrada. Na linha 21, é feito a análise e criação do histograma, unindo os três histogramas e um só. O resultado está na figura logo abaixo.

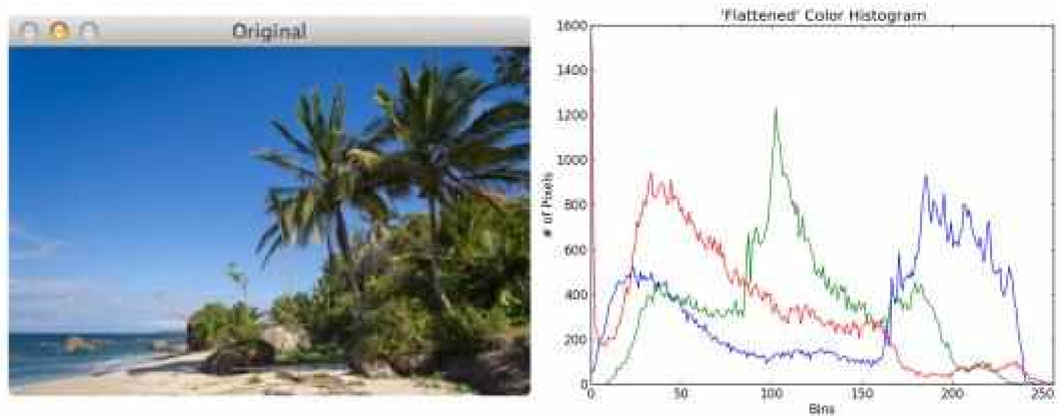

Figura 48- Resultado dos histogramas coloridos

Olhando para o histograma, pode-se observar um pico de verde no valor de 100. Isso indica um valor alto de verde mais escuro, que vem das árvores e da vegetação da praia.

Também é possível se ver vários pixels azuis entre 170 e 225. Como são pixels mais claros, sabemos que são os pixels do céu azul da imagem. Existe também uma concentração de pixels azuis na faixa de 25 a 50, pixels muito mais escuros que são as águas do oceano no canto inferior-esquerdo da figura 48.

Até agora, foi considerado apenas um canal por vez nas análises do histograma, mas pode-se fazer análises multidimensionais também. Este tipo de histograma responde perguntas como: "Quantos pixels possuem valor em vermelho de 10 e valores de azul de 30? ", "Quantos pixels possuem valor de verde em 200 e valor de vermelho em 130? ".

```
25 fig = plt.figure()
26
27 ax = fig.add_subplot(131)
28 hist = cv2.calcHist([chans[1], chans[0]], [0, 1], None,
29     [32, 32], [0, 256, 0, 256])
30 p = ax.imshow(hist, interpolation = "nearest")
31 ax.set_title("2D Color Histogram for G and B")
32 plt.colorbar(p)
33
34 ax = fig.add_subplot(132)
35 hist = cv2.calcHist([chans[1], chans[2]], [0, 1], None,
36     [32, 32], [0, 256, 0, 256])
37 p = ax.imshow(hist, interpolation = "nearest")
38 ax.set_title("2D Color Histogram for G and R")
39 plt.colorbar(p)
40
41 ax = fig.add_subplot(133)
42 hist = cv2.calcHist([chans[0], chans[2]], [0, 1], None,
43     [32, 32], [0, 256, 0, 256])
44 p = ax.imshow(hist, interpolation = "nearest")
45 ax.set_title("2D Color Histogram for B and R")
46 plt.colorbar(p)
47
48 print("2D histogram shape: {}, with {} values".format(
49     hist.shape, hist.flatten().shape[0]))
```

Figura 49- Histograma multidimensional

Neste código, mostrado na figura 49, realizamos a análise bidimensional para todas as combinações do RGB. Não podemos usar 256 caixas dessa vez, pois dessa maneira, quando analisamos o bidimensional, teríamos 256 x 265 = 65.536 contadores de pixel. Estaremos gastando recursos atoa, além de não ser prático. A maioria das aplicações multidimensionais, utilizam de 8 a 64 bits. As linhas 28 e 29 mostram que usamos 32 bits ao invés de 256.

O Resultado dos histogramas pode ser visto logo abaixo:

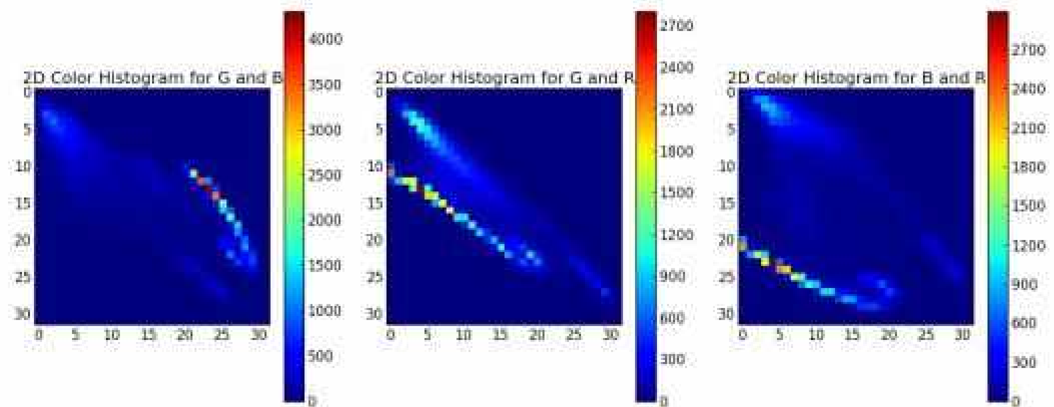

Figura 50- Resultado do histograma multidimensional

2.6 - SUAVISAR E BORRAR

Quando olhamos para fotografias, geralmente queremos evitar os borrões, mas para a visão computacional eles são indispensáveis, pois melhoram nas tarefas de detecção de cantos e detecção de formas. Borrar nada mais é que misturar os pixels que estão próximos na figura.

O primeiro método de borrar é o método da média (averaging). Foi definido uma janela k x k em cima da nossa imagem, aonde k será um número ímpar. O pixel no centro desta janela (k deve ser ímpar para que justamente tenhamos um centro definido por um pixel) será a média de todos os pixels ao seu redor. Chamamos essa janela de “ convolucional kernel” ou só “kernel”. Veremos que, quanto maior o kernel, mais borrada a imagem ficará, isso será implementado de acordo com a figura 51.

```
12 blurred = np.hstack([
13     cv2.blur(image, (3, 3)),
14     cv2.blur(image, (5, 5)),
15     cv2.blur(image, (7, 7))])
16 cv2.imshow("Averaged", blurred)
17 cv2.waitKey(0)
```

Figura 51- Borrando a imagem usando a média

Figura 52- Imagem borrada utilizando um kernel de 3 x 3 (esquerda) e um 5 x 5 (meio) e um 7 x 7(direita)

Para usar a média, foi utilizado a função *cv2.blur* que precisa de dois argumentos: a imagem e o tamanho do kernel. As linhas 13 a 15 mostram que foi borrou-se a imagem com tamanhos crescentes de kernel. Quanto maior o kernel, maior o borrão.

Utilizou-se a função *np.hstack* para juntar as imagens. Este método une as imagens horizontalmente. Isto é útil para que não seja criado três janelas separadas usando o *cv2.imshow*.

Outro método para borrar é o método Guassiano (Gaussian blurring). Este método é similar ao médio, mas ao invés de usar uma simples média, usa-se uma média com pesos, onde os pixels vizinhos contribuem mais do que a maioria. O resultado é uma imagem menos borrada, mas com um borrão mais natural, representado na figura 53 e implementado na figura 54.

Figura 53 - Borrando utilizando o método Guassiano

```
18 blurred = np.hstack([
19     cv2.GaussianBlur(image, (3, 3), 0),
20     cv2.GaussianBlur(image, (5, 5), 0),
21     cv2.GaussianBlur(image, (7, 7), 0)])
22 cv2.imshow("Gaussian", blurred)
23 cv2.waitKey(0)
```

Figura 54- código para usar Gaussian blurring

Foi utilizado a função *cv2.GaussianBlur* nas linhas 19 a 21. O primeiro argumento é a imagem. Então, colocou-se uma tupla representando o tamanho do kernel (igual ao *cv2.blur*).

O último parâmetro é o “σ”, o desvio padrão na direção do eixo x. Colocando este valor em 0, o OpenCV vai automaticamente computa-lo, variando de acordo com o tamanho do kernel escolhido.

Outro método é o método da mediana. Este é o melhor método para remover os ruídos de sal-e-pimenta. Este tipo de ruído faz com que pequenos pontos pretos e brancos apareçam nas imagens.

Para implementar esse método, primeiro é necessário definir o tamanho do kernel. Então, considera-se todos os pixels vizinhos dentro de k x k, mas ao invés de trocar o pixel central pela média, troca-se pela mediana de seus vizinhos.

```
24 blurred = np.hstack([
25     cv2.medianBlur(image, 3),
26     cv2.medianBlur(image, 5),
27     cv2.medianBlur(image, 7)])
28 cv2.imshow("Median", blurred)
29 cv2.waitKey(0)
```

Figura 55- Implementação do método da mediana

Para utilizar o método da mediana, utiliza-se a função *cv2.medianBlur* e requer dois parâmetros: a imagem e o tamanho do kernel. Nas linhas 25 a 27, inicia-se com o tamanho do kernel em 3 e aumentamos para 5 e depois 7. O resultado está na figura 56.

Figura 56- Resultado do método da mediana

O último método é o bilateral. Quando borramos uma imagem, geralmente se perde os cantos e traços da imagem para melhorar os ruídos, mas se é necessário manter os cantos, utiliza-se este método. Este método consegue isso aplicando duas distribuições Guassianas.

A primeira considera apenas os pixels vizinhos espaciais, aqueles que estão próximos nas coordenadas espaciais. A segunda considera a intensidade dos pixels vizinhos, garantindo que apenas pixels com intensidade similar são incluídos.

No geral, esse método consegue manter os cantos da imagem e retira os ruídos. O lado ruim é que ele é consideravelmente mais lento que os outros métodos. A figura 57 mostra a implementação do método.

```
30 blurred = np.hstack([
31     cv2.bilateralFilter(image, 5, 21, 21),
32     cv2.bilateralFilter(image, 7, 31, 31),
33     cv2.bilateralFilter(image, 9, 41, 41)])
34 cv2.imshow("Bilateral", blurred)
35 cv2.waitKey(0)
```

Figura 57- Método bilateral

Figura 58- Resultado do método bilateral

Aplicou-se o método chamando a função *cv2.bilateralFilter* nas linhas 31 a 33. O primeiro parâmetro é a imagem. Então definiu-se o diâmetro da nossa vizinhança de pixels. O terceiro argumento é a cor σ. Um valor alto para a cor σ quer dizer que mais cores na vizinhança serão consideradas. E por último, deve-se que considerar um espaço para σ. Um valor mais alto de espaço quer dizer que pixels mais afastados do pixel central terão uma maior influência no cálculo do borrão.

2.7 - THRESHOLDING

Thresholding é a binarização da imagem. Um simples exemplo seria selecionar um pixel de valor p e então colocar todas as intensidades dos outros pixels abaixo de p em zero. Dessa maneira, cria-se uma representação binária da imagem.

Geralmente usamos thresholding para focar em objetos e áreas que estamos interessados na imagem. Aplicar métodos simples de thresholding requer intervenção humana. É necessário estabelecer um valor T. Todos os pixels abaixo de T serão colocados em 0 e todos acima de T serão colocados em 255. Também pode-se aplicar o inverso.

```
1  import numpy as np
2  import argparse
3  import cv2
4
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 blurred = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Image", image)
```

Figura 59- Aplicação do Thresholding

Nas linhas de 1 a 10, foi importado pacotes e iniciou-se a imagem. Na linha 11, mudamos o espaço de cor de RGB para grayscale. Aplicou-se o método Gaussian blurring na linha 12 com raio de 12. Aplicar esse método ajuda a remover alguns cantos de frequência alta na imagem.

```
14 (T, thresh) = cv2.threshold(blurred, 155, 255, cv2.THRESH_BINARY)
15 cv2.imshow("Threshold Binary", thresh)
16
17 (T, threshInv) = cv2.threshold(blurred, 155, 255, cv2.
       THRESH_BINARY_INV)
18 cv2.imshow("Threshold Binary Inverse", threshInv)
19
20 cv2.imshow("Coins", cv2.bitwise_and(image, image, mask =
       threshInv))
21 cv2.waitKey(0)
```

Figura 60- Thresholding

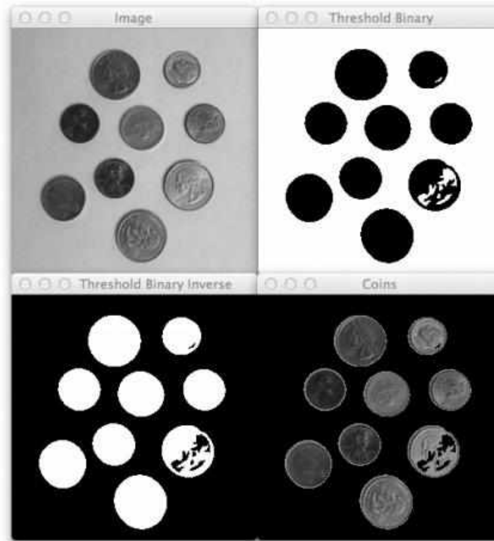

Figura 61- Resultado do Thresholding

Após borrarmos a imagem, computou-se o Thresholding na linha 14 usando a função *cv2.threshold*. Este método requer quatro argumentos. O primeiro é a imagem em grayscale. Depois, foi escolhido manualmente o valor T = 155. O terceiro argumento é o valor máximo aplicado no thresholding. Qualquer pixel acima do valor de T será colocado neste valor. Neste exemplo, qualquer pixel acima de 155 é colocado em 255 e qualquer valor abaixo é colocado em 0, como mostrado na figura 61.

E finalmente, precisa-se definir um método de thresholding. Usamos o *cv2.*THRESH_*BINARY* que indica que pixels acima de T são colocados no valor máximo (terceiro argumento).

O *cv2.threshold* retorna dois valores. O primeiro é T e o segundo é a imagem com o método aplicado.

Na linha 17, aplica-se um thresholding inverso usando *cv2.TRESHOLD_BINARY_INV* como método que faz com que agora, as moedas sejam brancas e o fundo preto. A última coisa que foi feita é revelar as moedas e nada mais.

Na linha 20, criou-se uma máscara usando *cv2.bitwise_and*. Colocamos a imagem original como os dois primeiros argumentos e então a nossa imagem com thresholding de máscara.

Um segundo método de Thresholding é o adaptativo. Um dos grandes problemas do thresholding simples é a necessidade de se colocar o valor T manualmente. Este método considera os pixels vizinhos para criar um valor para T.

```
1 import numpy as np
2 import argparse
3 import cv2
4
5 ap = argparse.ArgumentParser()
6 ap.add_argument("-i", "--image", required = True,
7     help = "Path to the image")
8 args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 blurred = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Image", image)
14
15 thresh = cv2.adaptiveThreshold(blurred, 255,
16     cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 11, 4)
17 cv2.imshow("Mean Thresh", thresh)
18
19 thresh = cv2.adaptiveThreshold(blurred, 255,
20     cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 15, 3)
21 cv2.imshow("Gaussian Thresh", thresh)
22 cv2.waitKey(0)
```

Figura 62- Thresholding adaptável

As linhas de 1 a 10 importam as bibliotecas e inicializam a imagem. As linhas 11 e 12 transformam as cores em grayscale e aplicam borrão. Aplicou-se o thresholding adaptável usando a função *cv2.adaptativeThreshold* na linha 15. O Primeiro parâmetro é a imagem. Depois foi colocado o valor máximo de 255, similar ao thresholding simples.

O terceiro argumento é o método para computar a vizinhança de pixels. Usando *cv2.ADAPTATIVA_THRESH_MEAN_C*, indica-se que se quer computar a média dos pixels vizinhos e trata-la como T.

O próximo argumento é o método de threshold. Utilizou-se o método *cv2.THRESH_BINARY_INV*.

O próximo parâmetro é o tamanho da vizinhança. Este inteiro deve ser ímpar e indica o quão grande a vizinhança de pixels será. Adotou-se o valor de 11, ou seja, examinaremos regiões de 11 x 11 da imagem.

E por fim, foi adotado o parâmetro C. Este valor é um inteiro que é subtraído da média, permitindo afinar nosso thresholding. Usamos C = 4 neste exemplo.

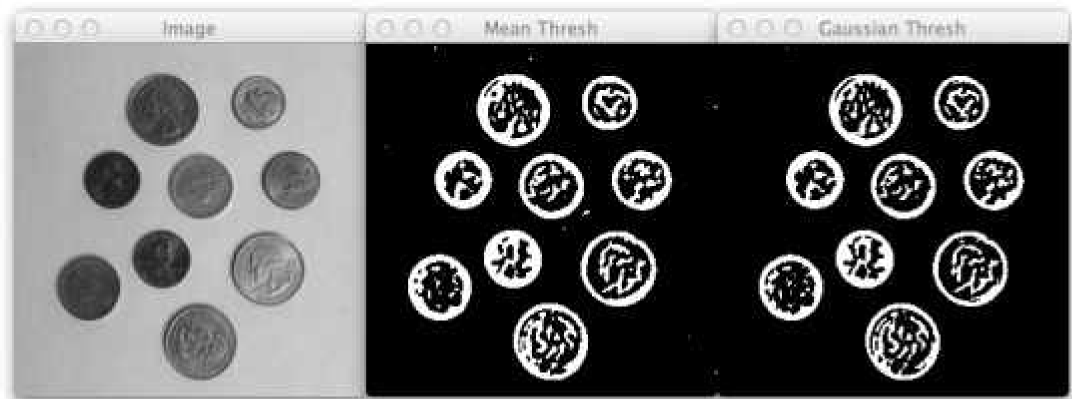

Figura 63 - Resultado dos Thresholdings

Mas ao invés de aplicar uma média padrão no thresholding, pode-se também aplicar um thresholding gaussiano como é feito na linha 17. A ordem dos parâmetros é a mesma, mas trocaremos algum deles. Foi utilizado *cv2.ADAPTAVIE_THRESH_GAUSSIAN_C*, usando uma vizinhança de 15 x 15 e foi mudado valor de C para 3. Os resultados podem ser vistos na figura 60.

No geral, escolher entre os dois métodos requer experimentos para ver qual ficará melhor.

Outra maneira de computar o valor de T é utilizando o método de Otsu e Riddler-Calvard. Este método assume que há dois picos no histograma de tons de cinza da imagem. Então, ele tenta encontrar o melhor valor para separar esses picos, que seria nosso valor T.

```
1  from __future__ import print_function
2  import numpy as np
3  import argparse
4  import mahotas
5  import cv2
6
7  ap = argparse.ArgumentParser()
8  ap.add_argument("-i", "--image", required = True,
9      help = "Path to the image")
10 args = vars(ap.parse_args())
11
12 image = cv2.imread(args["image"])
13 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
14 blurred = cv2.GaussianBlur(image, (5, 5), 0)
15 cv2.imshow("Image", image)
16
17 T = mahotas.thresholding.otsu(blurred)
18 print("Otsu's threshold: {}".format(T))
```

Figura 64- otsu_and_riddler.py

Nas linhas de 1 a 5, realizou-se as importações necessárias, utilizando a biblioteca mahotas (O OpenCV possui suporte para o método Otsu, mas o do mahotas é mais bem desenvolvido e por isso foi escolhido ele).

Linhas 7 a 12 realizam o carregamento da imagem. Realizou-se a troca para grayscale e então foi utilizado a função *T = mahotas.thresholding.otsu(blurred)* para adquirir nosso valor T.

```
19 thresh = image.copy()
20 thresh[thresh > T] = 255
21 thresh[thresh < 255] = 0
22 thresh = cv2.bitwise_not(thresh)
23 cv2.imshow("Otsu", thresh)
24
25 T = mahotas.thresholding.rc(blurred)
26 print("Riddler-Calvard: {}".format(T))
27 thresh = image.copy()
28 thresh[thresh > T] = 255
29 thresh[thresh < 255] = 0
30 thresh = cv2.bitwise_not(thresh)
31 cv2.imshow("Riddler-Calvard", thresh)
32 cv2.waitKey(0)
```

Figura 65- Aplicando o thresholding

Nas linhas 19 a 22 aplicou-se o thresholding. Também é possivel utilizar o método de Riddler-Calvard, bem semelhante ao método de Otsu. Na linha 25 foi aplicado este método. O resultado pode ser visto na figura 66.

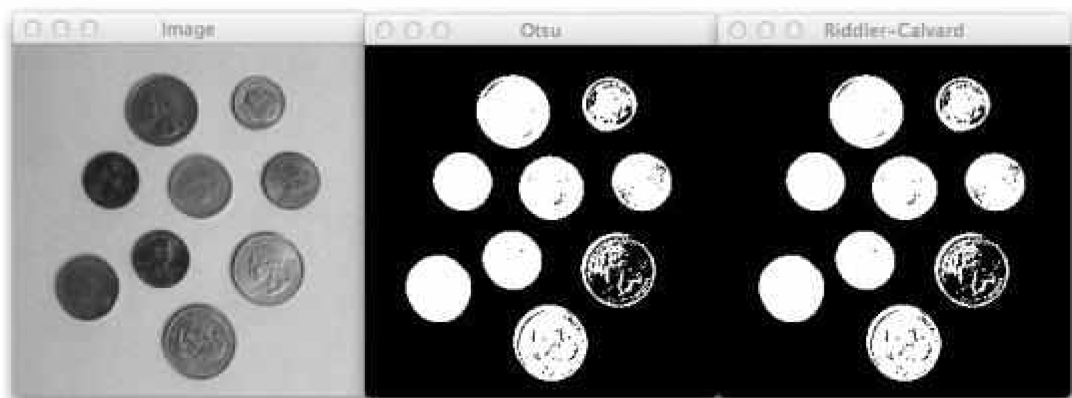

Figura 66- A esquerda a figura original, ao meio o método de Otsu e a direita o método de Riddler-Calvard

2.8 - GRADIENTES E DETECÇÃO DE CANTOS

Para detectar cantos nas imagens, aplicou-se métodos matemáticos capazes de identificar mudanças drásticas nos pixels da imagem.

Deve-se encontrar o “gradiente” da imagem em grayscale, para que seja possível encontrar regiões de canto nas coordenadas x e y.

O primeiro método para detecção de cantos é o método de Lapace e Sobel.

```
1  import numpy as np
2  import argparse
3  import cv2
4
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 cv2.imshow("Original", image)
13
14 lap = cv2.Laplacian(image, cv2.CV_64F)
15 lap = np.uint8(np.absolute(lap))
16 cv2.imshow("Laplacian", lap)
17 cv2.waitKey(0)
```

Figura 67- sobel_and_laplacian.py

Geralmente, quando se calcula o gradiente, utiliza-se apenas um canal, neste caso, estamos usando grayscale. Na linha 14, foi utilizado a função *cv2.Laplacian* para calcular esse gradiente. O primeiro argumento é a imagem em grayscale e o segundo argumento é o tipo de dado da imagem que final terá.

Neste caso, utilizou-se 64 bits por um único fator. Quando é realizado uma transformação do preto-para-branco, ela é considerada positiva, mas quando se realiza branco-para-preto, esta é considerada negativa. Números inteiros de 8 bits não conseguem representar números negativos e por isso utilizamos 64 bits. Para se ter certeza que seja detectado todos os cantos, utilizou-se um número decimal (float) e foi pego o valor absoluto do gradiente e

converteu-se de voltar para 8 bits. A linha 15 da figura 67 mostra esse processo. Se isso não for feito, é possível que não se detecte todos os cantos da imagem.

Figura 68- Resultado utilizando Laplaciano

```
18 sobelX = cv2.Sobel(image, cv2.CV_64F, 1, 0)
19 sobelY = cv2.Sobel(image, cv2.CV_64F, 0, 1)
20
21 sobelX = np.uint8(np.absolute(sobelX))
22 sobelY = np.uint8(np.absolute(sobelY))
23
24 sobelCombined = cv2.bitwise_or(sobelX, sobelY)
25
26 cv2.imshow("Sobel X", sobelX)
27 cv2.imshow("Sobel Y", sobelY)
28 cv2.imshow("Sobel Combined", sobelCombined)
29 cv2.waitKev(0)
```

Figura 69- Gradiente de Sobel

Usando o operador de Sobel, pode-se calcular o gradiente nos eixos x e y, permitindo encontrar os cantos horizontais e verticais. Nas linhas 18 e 19 foi realizado esse processo, usando a função *cv2.Sobel*. O primeiro argumento é a imagem, o segundo é o tipo do dado, utilizou-se o tipo *float* novamente, e as derivativas para os eixos x e y. Especificar um valor de 1 e 0 faz encontrar os cantos verticais e 0 e 1 faz encontrar os horizontais.

Nas linhas 21 e 22 garantiu-se a detecção de todos os cantos usando o valor absoluto e para combinar os gradientes dos dois eixos, utilizou-se *cv2.bitwise_or*. Nas linhas de 26 a 29 é mostrado os gradientes. Na figura 70 temos os resultados.

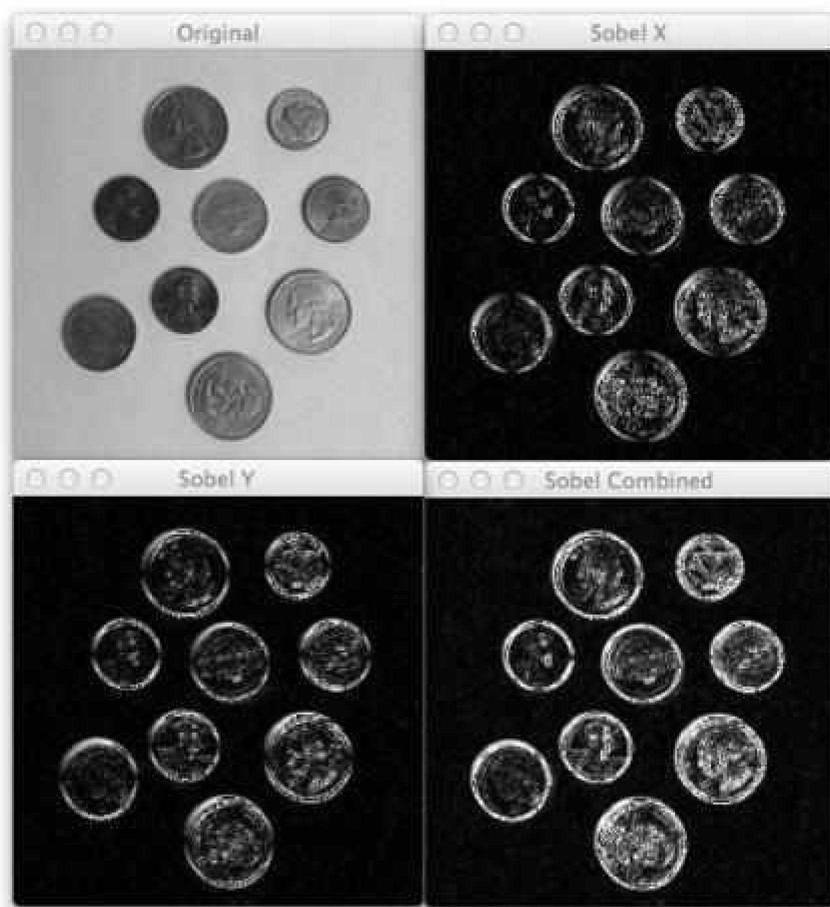

Figura 70- Resultados usando Sobel

Pode-se notar que as delimitações dos cantos estão com muito ruído. Para consertar isto, foi utilizado uma outra técnica, Canny edge detector.

Esta técnica é um processo de múltiplos degraus. Iremos aplicar borrão a imagem, computar o gradiente de Sobel e depois aplicamos um thresholding para saber se o pixel é um canto ou não.

```
1  import numpy as np
2  import argparse
3  import cv2
4
5  ap = argparse.ArgumentParser()
6  ap.add_argument("-i", "--image", required = True,
7      help = "Path to the image")
8  args = vars(ap.parse_args())
9
10 image = cv2.imread(args["image"])
11 image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
12 image = cv2.GaussianBlur(image, (5, 5), 0)
13 cv2.imshow("Blurred", image)
14
15 canny = cv2.Canny(image, 30, 150)
16 cv2.imshow("Canny", canny)
17 cv2.waitKey(0)
```

Figura 71- canny.py

Importou-se os pacotes, foi carregada a imagem, aplicou-se grayscale e foi utilizado um borrão Gaussiano. Aplicando este método, será reduzido os ruídos dos cantos. Na linha 15 foi aplicado a detecção de cantos Canny usando a função cv2.*canny*. O primeiro argumento é a imagem, e logo após é necessário dos valores, *threshold1* e *threshold2*. Qualquer valor acima de *threshold2* será considerado um canto. Qualquer valor abaixo de *threshold1* será considerado um não-canto. Valores dentro da faixa são considerados cantos os não cantos de acordo com a intensidade dos pixels, neste caso, abaixo de 30 e acima de 150. Na linha 16 é mostrado os resultados, que podem ser vistos na figura 72.

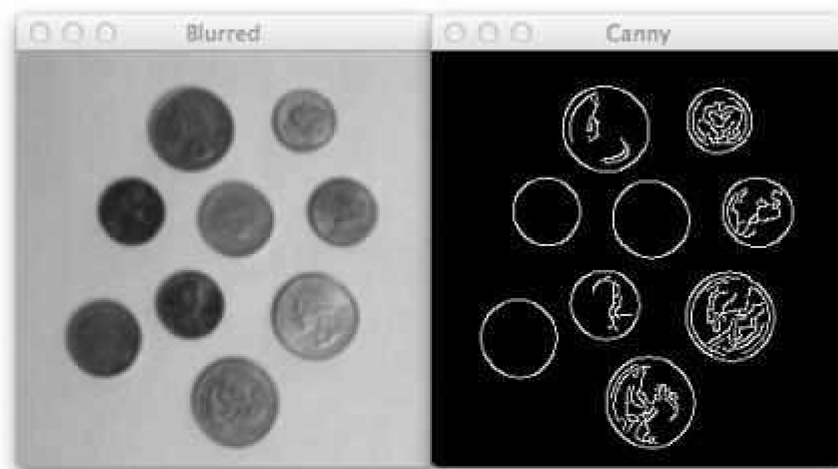

Figura 72- Resultados da detecção Canny

2.9 - CONTORNOS

Com a técnica de detectar cantos, é possível utiliza-la para encontrar contornos e assim, contar o número de moedas em uma figura.

O OpenCV possui métodos para encontrar "curvas" em uma imagem, chamadas contornos. Um contorno é uma curva de pontos, sem falhas na curva. Contornos são extremamente úteis quando se faz necessário detectar formas e analisa-las.

Para encontrar os contornos, precisa-se binarizar a imagem, usando thresholding ou detecção de cantos.

```
1  from __future__ import print_function
2  import numpy as np
3  import argparse
4  import cv2
5
6  ap = argparse.ArgumentParser()
7  ap.add_argument("-i", "--image", required = True,
8      help = "Path to the image")
9  args = vars(ap.parse_args())
10
11 image = cv2.imread(args["image"])
12 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
13 blurred = cv2.GaussianBlur(gray, (11, 11), 0)
14 cv2.imshow("Image", image)
15
16 edged = cv2.Canny(blurred, 30, 150)
17 cv2.imshow("Edges", edged)
```

Figura 73 - counting_coins.py

Nas linhas de 1 a 10 inicializou-se o processo. De 11 a 13, aplicou-se Guassian blur e foi convetida a imagem para grayscale. Na linha 16 foi aplicada a detecção Canny.

```
18 (_, cnts, _) = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
       cv2.CHAIN_APPROX_SIMPLE)
19
20 print("I count {} coins in this image".format(len(cnts)))
21
22 coins = image.copy()
23 cv2.drawContours(coins, cnts, -1, (0, 255, 0), 2)
24 cv2.imshow("Coins", coins)
25 cv2.waitKey(0)
```

Figura 74- Contagem de contornos

Detectando os cantos, pode-se encontrar os contornos na imagem. Usando a função *cv2.findContours* na linha 18, tem-se uma tupla com três retornos, a imagem com os contornos detectados, os contornos em si e a hierarquia de contornos. O primeiro argumento é a imagem. Importante saber que, este processo irá destruir a imagem, deixando apenas os contornos. O segundo argumento é o tipo de contorno que procurado. Usou-se *cv2.RETR_EXTERNAL* para selecionar apenas os contornos mais de fora. Utilizou-se *cv2.RETR_LIST* para selecionar todos os contornos. O último argumento é como aproxima-se o contorno. Foi usado *cv2.CHAIN_APPROX_SIMPLE* para um melhor contorno.

Os contornos "cnts" serão parte de uma lista Python. Pode-se utilizar a função *len* para contar o número de contornos. Foi executado esse processo na linha 20.

Na linha 22, foi feita uma cópia da imagem para não se perder a imagem no processo de criar os contornos. A função *cv2.drawContours* desenhará os contornos na imagem. O primeiro argumento é a imagem, o segundo é a lista de contornos. Especificando o index dos contornos em -1, indica-se que se quer que desenhe todos os contornos. Se especificar um número inteiro, será desenhado apenas o contorno indicado. O último argumento é a espessura da linha. Neste caso, adotou-se 2 pixels.

Na linha 24 os contornos são desenhados e a figura 75 mostra o resultado.

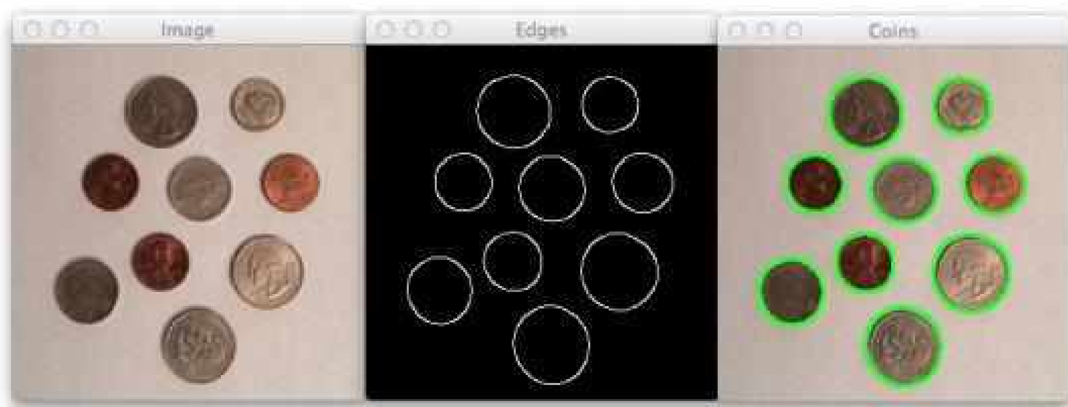

Figura 75- Resultado da contagem de modas

Também é possivel utilizar a técnica de corte para retirar apenas as moedas da imagem.

```
26 for (i, c) in enumerate(cnts):
27     (x, y, w, h) = cv2.boundingRect(c)
28
29     print("Coin #{}".format(i + 1))
30     coin = image[y:y + h, x:x + w]
31     cv2.imshow("Coin", coin)
32
33     mask = np.zeros(image.shape[:2], dtype = "uint8")
34     ((centerX, centerY), radius) = cv2.minEnclosingCircle(c)
35     cv2.circle(mask, (int(centerX), int(centerY)), int(radius),
           255, -1)
36     mask = mask[y:y + h, x:x + w]
37     cv2.imshow("Masked Coin", cv2.bitwise_and(coin, coin, mask =
           mask))
38     cv2.waitKey(0)
```

Figura 76- Selecionando apenas as moedas

Na linha 26, começamos um loop para selecionar todos os contornos. Utilizou-se *cv2boundingRect* no contorno selecionado. Este método faz com que se crie um quadrado ao redor do contorno, permitindo corta-lo e separa-lo. Essa função requer apenas um parâmetro, o contorno e irá retornar uma tupla de e posições x e y aonde o retângulo começará, além da altura e comprimento do retângulo.

Realiza-se o corte na imagem nas linhas 30 e mostra-se o resultado na linha 31.

Também é possivel encontrar um círculo ao redor do contorno. Inicializa-se uma matriz NumPy na linha 33 para criar uma máscara e utiliza-se a função *cv2.minEnclosingCircle* na

linha 34 que irá encaixar um círculo no contorno. Colocou-se uma variável que será o círculo (no caso “c”), as coordenadas x e y do círculo e seu raio.

Utilizando as coordenadas, pode-se criar um círculo na máscara que representará a moeda. Na linha 36 nós cortou-se a máscara do mesmo modo que a moeda.

Para mostrar apenas a moeda e ignorar o background da imagem, chamou-se a função para a operação binária E (AND) e apresentou-se o resultado na linha 37, mostrado na figura 77.

Figura 77 - Resultado do corte da moeda

2.10 - SUBTRAÇÃO DE PLANO DE FUNDO (BACKGROUND)

Esta é uma técnica muito importante na visão computacional e muito utilizada em várias outras aplicações. Pode-se utilizar para detecção de movimento, contagem de carros que passam por um determinado local, número de pessoas que passam por uma determinada loja e várias outras aplicações. Os dois métodos principais são o modelo Guassiano e a segmentação do plano de fundo. Nas referências bibliográficas deste trabalho, os links VII e VIII levam a artigos que falam mais sobre estes métodos caso queira aprofundar mais nos algoritmos.

2.11 - DETECÇÃO DE MOVIMENTO

Quando se fala de detecção de movimento, muitas variáveis entram em questão. Geralmente adota-se a ideia de que o plano de fundo será totalmente estático e não mudará (quando analisamos um vídeo). Se houver muitas alterações, será considerado essas alterações como movimento.

Logicamente, no mundo real, essa premissa com certeza falhará. Isso por causa do movimento das sombras, reflexo, condições da luz e várias outras coisas que o ambiente pode afetar. O método implementado visa uma introdução ao assunto, e muitas vezes pode não funcionar se o plano de fundo se modificar de mais ou o contraste de cores for muito parecido. È possível realizar ajustes para tentar melhora-lo e será necessário caso seja muito diferente a iluminação local.

Criou-se um arquivo chamado *motion_detector.py* para exemplificar a detecção de movimento.

Basic motion detection and tracking with Python and OpenCV — Python

```
1  # import the necessary packages
2  import argparse
3  import datetime
4  import imutils
5  import time
6  import cv2
7
8  # construct the argument parser and parse the arguments
9  ap = argparse.ArgumentParser()
10 ap.add_argument("-v", "--video", help="path to the video file")
11 ap.add_argument("-a", "--min-area", type=int, default=500, help="minimum area size
12 args = vars(ap.parse_args())
13
14 # if the video argument is None, then we are reading from webcam
15 if args.get("video", None) is None:
16     camera = cv2.VideoCapture(0)
17     time.sleep(0.25)
18
19 # otherwise, we are reading from a video file
20 else:
21     camera = cv2.VideoCapture(args["video"])
22
23 # initialize the first frame in the video stream
24 firstFrame = None
```

Figura 78- motion_detector.py

Nas linhas de 2 a 6 foi importado os pacotes necessários. Utilizou-se o *imultis* já visto neste trabalho. Nas linhas de 9 a 12 realizou-se o carregamento do arquivo a ser analisado. Pode-se adicionar um arquivo de vídeo para ser analisado. Nas linhas 15 a 21, foi criado um método para que, se não houver arquivo, que se inicie a webcam do computador.

```
Basic motion detection and tracking with Python and OpenCV                Python
26 # loop over the frames of the video
27 while True:
28     # grab the current frame and initialize the occupied/unoccupied
29     # text
30     (grabbed, frame) = camera.read()
31     text = "Unoccupied"
32 
33     # if the frame could not be grabbed, then we have reached the end
34     # of the video
35     if not grabbed:
36         break
37 
38     # resize the frame, convert it to grayscale, and blur it
39     frame = imutils.resize(frame, width=500)
40     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
41     gray = cv2.GaussianBlur(gray, (21, 21), 0)
42 
43     # if the first frame is None, initialize it
44     if firstFrame is None:
45         firstFrame = gray
46         continue
```

Figura 79 - Código para sensor de movimento

Tendo adquirido a referência ou a câmera, iniciou-se o loop na linha 27. Este loop irá acontecer até que o vídeo acabe ou, no caso da câmera, até que a webcam seja desativada. Nas linhas 39 a 41 é que se inicia o processamento de imagem.

Primeiramente redimensiona-se para 500 pixels, pois não há motivo para processarmos imagens grandes. Converteu-se a imagem para grayscale. Dois frames de vídeo nunca serão iguais e por isso, deve-se reduzir o ruído aplicando um borrão com Guassian blurring.

Como é necessário um modelo para o plano de fundo, foi utilizado o frame inicial como modelo, e se algo se diferenciar dele, será então detectado algum movimento no local.

```
Basic motion detection and tracking with Python and OpenCV                    Python
48        # compute the absolute difference between the current frame and
49        # first frame
50        frameDelta = cv2.absdiff(firstFrame, gray)
51        thresh = cv2.threshold(frameDelta, 25, 255, cv2.THRESH_BINARY)[1]
52
53        # dilate the thresholded image to fill in holes, then find contours
54        # on thresholded image
55        thresh = cv2.dilate(thresh, None, iterations=2)
56        (cnts, _) = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
57            cv2.CHAIN_APPROX_SIMPLE)
58
59        # loop over the contours
60        for c in cnts:
61            # if the contour is too small, ignore it
62            if cv2.contourArea(c) < args["min_area"]:
63                continue
64
65            # compute the bounding box for the contour, draw it on the frame,
66            # and update the text
67            (x, y, w, h) = cv2.boundingRect(c)
68            cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
69            text = "Occupied"
```

Figura 80- continuação do código de sensor de movimento

Realiza-se então uma comparação entre os frames. Para isso, basta subtrair um frame do outro. Pega-se o valor absoluto dessa subtração e armazenamos como *delta*. Aplicou-se um thresholding ao *frameDelta* na linha 51 para revelar apenas as regiões da imagem que sofreram mudanças significativas.

Figura 81 - Frame Delta e sua diferença com o frame inicial

Feito o tresholding, aplicou-se a detecção de contorno para encontrar as linhas da região branca (linha 56).

Inicializa-se um loop em todos os contornos (linha 60), filtrando os contornos não relevantes nas linhas 62 e 63. Se o contorno for maior que o valor estipulado, quer dizer que existiu um movimento, e atualiza-se a frase para “Room Status: Occupied” (linhas 67 e 68).

```
Basic motion detection and tracking with Python and OpenCV                Python
71        # draw the text and timestamp on the frame
72        cv2.putText(frame, "Room Status: {}".format(text), (10, 20),
73            cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
74        cv2.putText(frame, datetime.datetime.now().strftime("%A %d %B %Y %I:%M:%S%p"),
75            (10, frame.shape[0] - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.35, (0, 0, 255), 1)
76
77        # show the frame and record if the user presses a key
78        cv2.imshow("Security Feed", frame)
79        cv2.imshow("Thresh", thresh)
80        cv2.imshow("Frame Delta", frameDelta)
81        key = cv2.waitKey(1) & 0xFF
82
83        # if the `q` key is pressed, break from the lop
84        if key == ord("q"):
85            break
86
87    # cleanup the camera and close any open windows
88    camera.release()
89    cv2.destroyAllWindows()
```

Figura 82 - final do código de sensor de movimento

Nas linhas 77 a 80 mostra-se na tela o resultado da detecção. Se for um vídeo, será realizado até o fim dele. Se a câmera estiver acionada, o código continuará funcionando até que a câmera seja desativada ou até apertarmos a tecla “q” (linha 84).

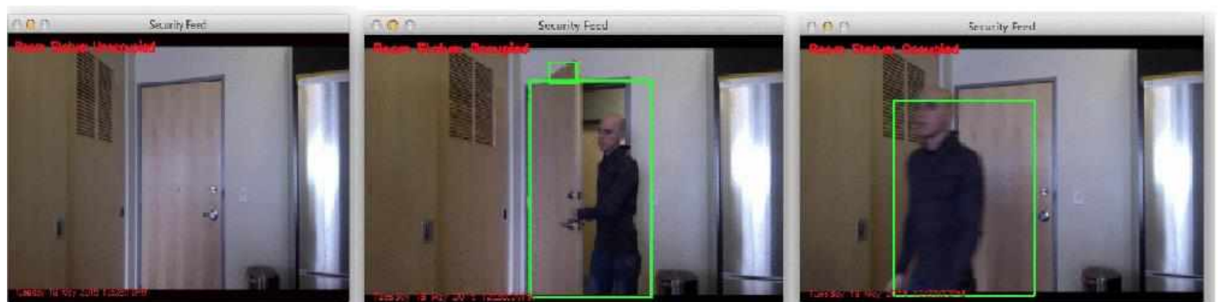

Figura 83- Resultados do sensor de movimento

2. APLICAÇÕES DA VISÃO COMPUTACIONAL

Neste capítulo será apresentada uma aplicação simples, mas bem útil, que pode ser desenvolvida com as teorias apresentadas e utilizando a linguagem PYTHON.

2.1 RECONHECIMENTO FACIAL

Utilizaremos as funções do OpenCV para realizar este código. Estas funções utilizam as técnicas vistas até agora, facilitando a implementação das aplicações.

```python
1  import cv2
2  import sys
3
4  # Get user supplied values
5  imagePath = sys.argv[1]
6  cascPath = "haarcascade_frontalface_default.xml"
```

Figura 84- Importação e inicialização do programa

Importa-se o OpenCV e a biblioteca sys para iniciar, como visto na figura 84. Utilizaremos uma teoria de cascata para esse código. Utilizar cascata pode parecer complicado, mas na verdade é bem simples. Foi feito uma sobreposição de arquivos XML contendo o código OpenCV para detectar os objetos.

Definiu-se a imagem e o nome das cascatas nas linhas 5 e 6.

```python
8  # Create the haar cascade
9  faceCascade = cv2.CascadeClassifier(cascPath)
```

Figura 85- Criação da cascata

Na linha 9, mostrava na figura 85, realizou-se a criação da cascata que será utilizada no reconhecimento. O arquivo "haarcascade_frontalface_default.xml" pode ser obtido na referência XI deste trabalho, referido no tópico "Referênciass Bibliográficas".

```python
11  # Read the image
12  image = cv2.imread(imagePath)
13  gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
```

Figura 86- Leitura da Imagem

Na figura 86, temos a inicialização da imagem e a conversão para escala grayscale, como feito em várias técnicas já vistas anteriormente.

```python
15  # Detect faces in the image
16  faces = faceCascade.detectMultiScale(
17      gray,
18      scaleFactor=1.1,
19      minNeighbors=5,
20      minSize=(30, 30),
21      flags = cv2.cv.CV_HAAR_SCALE_IMAGE
22  )
```

Figura 87- Reconhecimento Facial

Na figura 87, temos a inicialização do reconhecimento facial. A função detectMultiScale é uma função geral de detecção de objetos. Como foi instanciado uma cascata de face, esta função irá detectar faces. O primeiro argumento é a imagem a ser analisada.

O segundo argumento é o fator de escala. Como algumas faces podem estar mais próximas da câmera, esse fator irá compensar essa diferença de proximidade com a câmera.

O terceiro argumento define quantos objetos serão detectados próximos um ao outro. O algoritmo utilizado nesse procedimento, utiliza a criação de janelas em torno da face, e por isso foi utilizado "minSize" para determinar o tamanho mínimo dessa janela. A função irá retornar uma lista de retângulos que irá mostrar as faces da imagem.

```
24   print("Found {0} faces!".format(len(faces)))
25
26   # Draw a rectangle around the faces
27   for (x, y, w, h) in faces:
28       cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)
```

Figura 88- Loop para encontrar as faces

Na figura 88, realizou-se um loop para encontrar todas as faces na figura. Esta função retorna a localização x e y dos retângulos, além das dimensões deles. Com esses dados, foi desenhado retângulos nessas coordenadas.

```
30   cv2.imshow("Faces found", image)
31   cv2.waitKey(0)
```

Figura 89- Apresentação do resultado

Na figura 89, foi realizado a apresentação dos resultados da implementação do código. Utilzando o código na figura 90, podemos executar o código para verificar os resultados, que podem ser vistos na figura 91 e na figura 92, realizado para duas imagens diferentes.

```
$ python face_detect.py abba.png haarcascade_frontalface_default.xml
```

Figura 90 - Código para executar o Reconhecimento Facial

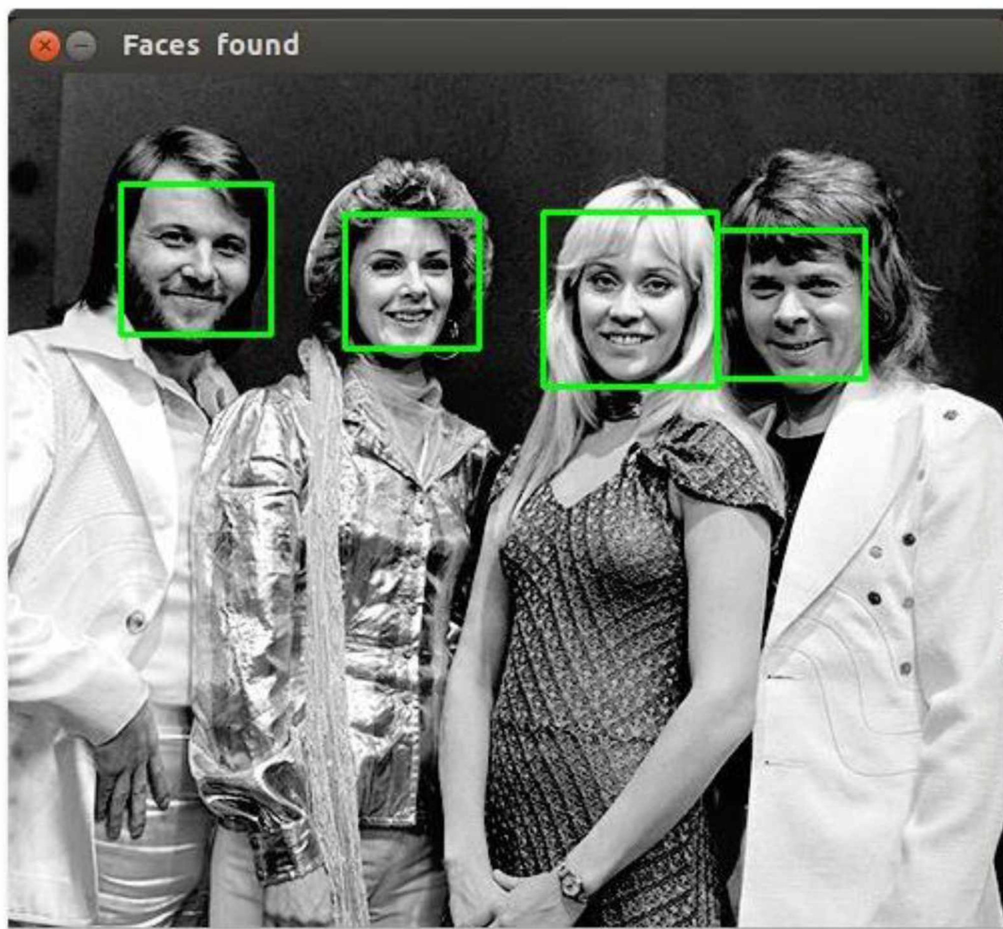

Figura 91- Imagem após a análise de reconhecimento facial

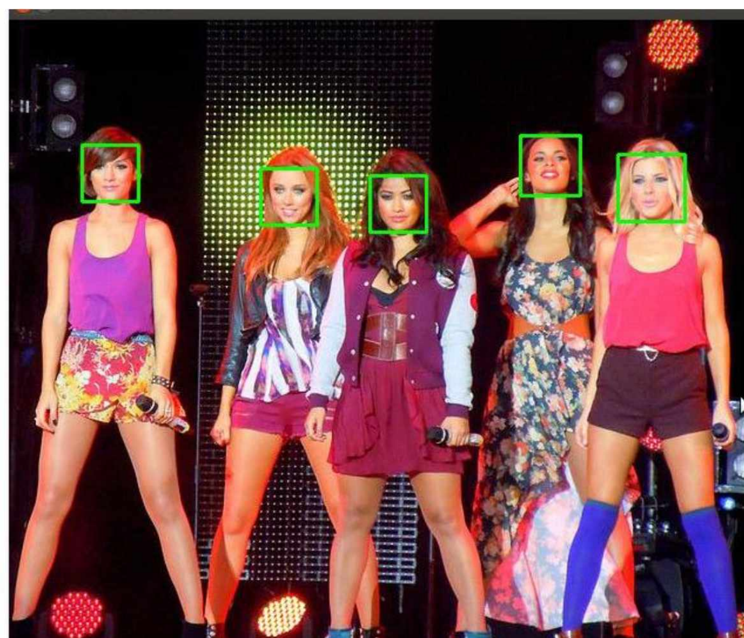

Figura 92- Imagem após a análise de reconhecimento facial

4 - CONCLUSÃO

A visão computacional, a cada dia que passa, será mais utilizada em nosso cotidiano e as teorias básicas e técnicas de análise de imagens são muito importantes para se avançar nesta área. Muitas das aplicações que podemos ver hoje em dia, nada mais são que a utilização de várias técnicas básicas em conjunto. Técnicas mais avançadas como o reconhecimento de face ou reconhecimento de movimentos, todos eles começam com princípios básicos de reconhecimento de cantos, formas, contornos.

Analisar um vídeo nada mais é que analisar várias imagens e retirando conclusões delas. Se unirmos estas técnicas básicas com algoritmos de aprendizagem, podemos realizar várias aplicações úteis no mundo atual.

Com a pesquisa bibliográfica, percebeu-se que muitas pesquisas estão sendo realizadas nessa área, e que a substituição de sensores por máquinas que utilizam a visão computacional já está acontecendo. O entendimento do conteúdo deste trabalho fará com que, alunos e interessados sobre o assunto possam iniciar na área, tendo exemplos práticos e uma boa base para técnicas mais avançadas na área da visão computacional.

Pessoalmente, este trabalho trouxe uma experiência muito proveitosa em uma área que previamente eu nunca havia trabalhado. Ao trabalhar com a visão computacional, além da experiência na linguagem de programação e nas teorias da visão, consegui observar a grande variedade de projetos que podem ser implementados e/ou melhorados com a visão computacional, sendo para mim, uma ótima maneira de otimizar várias aplicações atuais.

5 - REFERÊNCIAS BIBLIOGRÁFICAS

[I] http://www.pyimagesearch.com/practicalpython-opencv/ - acessado dia 06/09/2017 às 19:45

[II] http://www.pyimagesearch.com/2015/05/25/basic-motion-detection-and-[I] tracking- with-python-and-opencv/ - acessado dia 06/09/2017 às 19:45

[III] http://www.pyimagesearch.com/ - acessado dia 06/09/2017 às 19:45

[IV] http://datascienceacademy.com.br/blog/o-que-e-visao-computacional/ - acessado dia 06/09/2017 às 19:45

[V]ROSEBROOK, ADRIAN. Practical Python and OpenCV

[VI] ROSEBROOK, ADRIAN. Computer Vison on 21 days, May 5, 2016

[VII] http://www.ee.surrey.ac.uk/CVSSP/Publications/papers/KaewTraKulPong-AVBS01.pdf - acessado dia 06/09/2017 às 19:49

[VIII] http://www.zoranz.net/Publications/zivkovic2004ICPR.pdf - acessado dia 06/09/2017 às 19:49

[IX] http://goldberg.berkeley.edu/pubs/acc-2012-visual-tracking-final.pdf - acessado dia 06/09/2017 às 19:50

[X]https://github.com/shantnu/FaceDetect/blob/master/haarcascade_frontalface_default.xml - acessado 21/12/2017 às 23:09

ANEXO A

Código I – load_display_save.py

Código 1 – load_display_save.py

```
# Como usar
# python load_display_save.py --image ../images/trex.png

import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
        help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
print "width: %d pixels" % (image.shape[1])
print "height: %d pixels" % (image.shape[0])
print "channels: %d" % (image.shape[2])

cv2.imshow("Image", image)
cv2.waitKey(0)

cv2.imwrite("newimage.jpg", image)
```

ANEXO B

Código II – getting_and_setting.py

Código 2 – getting_and_setting.py

```
# Como usar
# python getting_and_setting.py --image ../images/trex.png

import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
        help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

 (b, g, r) = image[0, 0]
print "Pixel at (0, 0) - Red: %d, Green: %d, Blue: %d" % (r, g, b)

image[0, 0] = (0, 0, 255)
(b, g, r) = image[0, 0]
print "Pixel at (0, 0) - Red: %d, Green: %d, Blue: %d" % (r, g, b)

corner = image[0:100, 0:100]
cv2.imshow("Corner", corner)

image[0:100, 0:100] = (0, 255, 0)

cv2.imshow("Updated", image)
cv2.waitKey(0)
```

ANEXO C

Código III – drawing.py

Código 3 – drawing.py

```
# Como usar
# python drawing.py
import numpy as np
import cv2

canvas = np.zeros((300, 300, 3), dtype = "uint8")
green = (0, 255, 0)
cv2.line(canvas, (0, 0), (300, 300), green)
cv2.imshow("Canvas", canvas)
cv2.waitKey(0)

red = (0, 0, 255)
cv2.line(canvas, (300, 0), (0, 300), red, 3)
cv2.imshow("Canvas", canvas)
cv2.waitKey(0)

cv2.rectangle(canvas, (10, 10), (60, 60), green)
cv2.imshow("Canvas", canvas)
cv2.waitKey(0)

cv2.rectangle(canvas, (50, 200), (200, 225), red, 5)
cv2.imshow("Canvas", canvas)
cv2.waitKey(0)

blue = (255, 0, 0)
cv2.rectangle(canvas, (200, 50), (225, 125), blue, -1)
cv2.imshow("Canvas", canvas)
cv2.waitKey(0)

canvas = np.zeros((300, 300, 3), dtype = "uint8")
(centerX, centerY) = (canvas.shape[1] / 2, canvas.shape[0] / 2)
white = (255, 255, 255)

for r in xrange(0, 175, 25):
	cv2.circle(canvas, (centerX, centerY), r, white)
cv2.imshow("Canvas", canvas)
cv2.waitKey(0)

for i in xrange(0, 25):
```

```
        # randomly generate a radius size between 5 and 200,
        # generate a random color, and then pick a random
        # point on our canvas where the circle will be drawn
        radius = np.random.randint(5, high = 200)
        color = np.random.randint(0, high = 256, size = (3,)).tolist()
        pt = np.random.randint(0, high = 300, size = (2,))
        cv2.circle(canvas, tuple(pt), radius, color, -1)
cv2.imshow("Canvas", canvas)
cv2.waitKey(0)
```

ANEXO D

Código IV – translation.py

Código IV – translation.py

```
#Como usar
# python translation.py --image ../images/trex.png

import numpy as np
import argparse
import imutils
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

M = np.float32([[1, 0, 25], [0, 1, 50]])
shifted = cv2.warpAffine(image, M, (image.shape[1], image.shape[0]))
cv2.imshow("Shifted Down and Right", shifted)


M = np.float32([[1, 0, -50], [0, 1, -90]])
shifted = cv2.warpAffine(image, M, (image.shape[1], image.shape[0]))
cv2.imshow("Shifted Up and Left", shifted)

shifted = imutils.translate(image, 0, 100)
cv2.imshow("Shifted Down", shifted)
cv2.waitKey(0)
```

ANEXO E

Código V – imultis.py

Código V – imultis.py

```
import numpy as np
import cv2

def translate(image, x, y):
	# Define the translation matrix and perform the translation
	M = np.float32([[1, 0, x], [0, 1, y]])
	shifted = cv2.warpAffine(image, M, (image.shape[1], image.shape[0]))

	return shifted

def rotate(image, angle, center = None, scale = 1.0):
	(h, w) = image.shape[:2]

	if center is None:
		center = (w / 2, h / 2)

	M = cv2.getRotationMatrix2D(center, angle, scale)
	rotated = cv2.warpAffine(image, M, (w, h))

	return rotated

def resize(image, width = None, height = None, inter = cv2.INTER_AREA):
	dim = None
	(h, w) = image.shape[:2]

	if width is None and height is None:
		return image

	if width is None:
		# calculate the ratio of the height and construct the
		# dimensions
		r = height / float(h)
		dim = (int(w * r), height)

	else:
		# calculate the ratio of the width and construct the
		# dimensions
		r = width / float(w)
```

```
        dim = (width, int(h * r))

    resized = cv2.resize(image, dim, interpolation = inter)
return resized
```

ANEXO F

Código VI – rotate.py

Código Vi – rotate.py

```
# Como usar
# python rotate.py --image ../images/trex.png

import numpy as np
import argparse
import imutils
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
        help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

 (h, w) = image.shape[:2]
center = (w / 2, h / 2)

M = cv2.getRotationMatrix2D(center, 45, 1.0)
rotated = cv2.warpAffine(image, M, (w, h))
cv2.imshow("Rotated by 45 Degrees", rotated)

M = cv2.getRotationMatrix2D(center, -90, 1.0)
rotated = cv2.warpAffine(image, M, (w, h))
cv2.imshow("Rotated by -90 Degrees", rotated)

rotated = imutils.rotate(image, 180)
cv2.imshow("Rotated by 180 Degrees", rotated)
cv2.waitKey(0)
```

ANEXO G

Código VII – resize.py

Código VII – resize.py

```
# Como usar
# python resize.py --image ../images/trex.png

import numpy as np
import argparse
import imutils
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

r = 150.0 / image.shape[1]
dim = (150, int(image.shape[0] * r))

resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
cv2.imshow("Resized (Width)", resized)

r = 50.0 / image.shape[0]
dim = (int(image.shape[1] * r), 50)

resized = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
cv2.imshow("Resized (Height)", resized)
cv2.waitKey(0)

resized = imutils.resize(image, width = 100)
cv2.imshow("Resized via Function", resized)
cv2.waitKey(0)
```

ANEXO H

Código VIII – flipping.py

Código VIII – flipping.py

```
# Como usar
# python flipping.py --image ../images/trex.png

import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

flipped = cv2.flip(image, 1)
cv2.imshow("Flipped Horizontally", flipped)

flipped = cv2.flip(image, 0)
cv2.imshow("Flipped Vertically", flipped)

flipped = cv2.flip(image, -1)
cv2.imshow("Flipped Horizontally & Vertically", flipped)
cv2.waitKey(0)
```

ANEXO I

Código IX – crop.py

Código IX – crop.py

```
# Como usar
# python crop.py --image ../images/trex.png

import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser(
ap.add_argument("-i", "--image", required = True,
        help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

cropped = image[30:120 , 240:335]
cv2.imshow("T-Rex Face", cropped)
cv2.waitKey(0)
```

ANEXO J

Código X – arithmatic.py

Código X- arithmetic.py

```
# Como usar
# python arithmetic.py --image ../images/trex.png

import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

print "max of 255: " + str(cv2.add(np.uint8([200]), np.uint8([100])))
print "min of 0: " + str(cv2.subtract(np.uint8([50]), np.uint8([100])))

print "wrap around: " + str(np.uint8([200]) + np.uint8([100]))
print "wrap around: " + str(np.uint8([50]) - np.uint8([100]))

M = np.ones(image.shape, dtype = "uint8") * 100
added = cv2.add(image, M)
cv2.imshow("Added", added)

M = np.ones(image.shape, dtype = "uint8") * 50
subtracted = cv2.subtract(image, M)
cv2.imshow("Subtracted", subtracted)
cv2.waitKey(0)
```

ANEXO K

Código XI – bitwise.py

Código XI - bitwise.py

```
#Como usar
# python bitwise.py

import numpy as np
import cv2

rectangle = np.zeros((300, 300), dtype = "uint8")
cv2.rectangle(rectangle, (25, 25), (275, 275), 255, -1)
cv2.imshow("Rectangle", rectangle)

circle = np.zeros((300, 300), dtype = "uint8")
cv2.circle(circle, (150, 150), 150, 255, -1)
cv2.imshow("Circle", circle)

bitwiseAnd = cv2.bitwise_and(rectangle, circle)
cv2.imshow("AND", bitwiseAnd)
cv2.waitKey(0)

bitwiseOr = cv2.bitwise_or(rectangle, circle)
cv2.imshow("OR", bitwiseOr)
cv2.waitKey(0)

bitwiseXor = cv2.bitwise_xor(rectangle, circle)
cv2.imshow("XOR", bitwiseXor)
cv2.waitKey(0)

cv2.imshow("NOT", bitwiseNot)
cv2.waitKey(0)
```

ANEXO L

Código XII – masking.py

Código XII - masking.py

```
# Como usar
# python masking.py --image ../images/beach.png

import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
        help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

mask = np.zeros(image.shape[:2], dtype = "uint8")
(cX, cY) = (image.shape[1] / 2, image.shape[0] / 2)
cv2.rectangle(mask, (cX - 75, cY - 75), (cX + 75 , cY + 75), 255, -1)
cv2.imshow("Mask", mask)

masked = cv2.bitwise_and(image, image, mask = mask)
cv2.imshow("Mask Applied to Image", masked)
cv2.waitKey(0)

mask = np.zeros(image.shape[:2], dtype = "uint8")
cv2.circle(mask, (cX, cY), 100, 255, -1)
masked = cv2.bitwise_and(image, image, mask = mask)
cv2.imshow("Mask", mask)
cv2.imshow("Mask Applied to Image", masked)
cv2.waitKey(0)
```

ANEXO M

Código XIII – Split_and_merging.py

Código XII – Split_and_merging.py

```
# Como usar
# python splitting_and_merging.py --image ../images/wave.png

import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
(B, G, R) = cv2.split(image)

cv2.imshow("Red", R)
cv2.imshow("Green", G)
cv2.imshow("Blue", B)
cv2.waitKey(0)

merged = cv2.merge([B, G, R])
cv2.imshow("Merged", merged)
cv2.waitKey(0)
cv2.destroyAllWindows()

zeros = np.zeros(image.shape[:2], dtype = "uint8")
cv2.imshow("Red", cv2.merge([zeros, zeros, R]))
cv2.imshow("Green", cv2.merge([zeros, G, zeros]))
cv2.imshow("Blue", cv2.merge([B, zeros, zeros]))
cv2.waitKey(0)
```

ANEXO N

Código XIV – grayscale_histogram.py

Código XIV –grayscale_histogram.py

```
# Como usar
# python grayscale_histogram.py --image ../images/beach.png

from matplotlib import pyplot as plt
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Original", image)

hist = cv2.calcHist([image], [0], None, [256], [0, 256])

plt.figure()
plt.title("Grayscale Histogram")
plt.xlabel("Bins")
plt.ylabel("# of Pixels")
plt.plot(hist)
plt.xlim([0, 256])
plt.show()
cv2.waitKey(0)
```

ANEXO O

Código XV – color_histograms.py

Código XV– color_histograms.py

```
# Como usar
# python color_histograms.py --image ../images/beach.png

from matplotlib import pyplot as plt
import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

chans = cv2.split(image)
colors = ("b", "g", "r")
plt.figure()
plt.title("'Flattened' Color Histogram")
plt.xlabel("Bins")
plt.ylabel("# of Pixels")

for (chan, color) in zip(chans, colors):
	# Create a histogram for the current channel and plot it
	hist = cv2.calcHist([chan], [0], None, [256], [0, 256])
	plt.plot(hist, color = color)
	plt.xlim([0, 256])

fig = plt.figure()

ax = fig.add_subplot(131)
hist = cv2.calcHist([chans[1], chans[0]], [0, 1], None,
	[32, 32], [0, 256, 0, 256])
p = ax.imshow(hist, interpolation = "nearest")
ax.set_title("2D Color Histogram for G and B")
```

```
plt.colorbar(p)

ax = fig.add_subplot(132)
hist = cv2.calcHist([chans[1], chans[2]], [0, 1], None,
        [32, 32], [0, 256, 0, 256])
p = ax.imshow(hist, interpolation = "nearest")
ax.set_title("2D Color Histogram for G and R")
plt.colorbar(p)

ax = fig.add_subplot(133)
hist = cv2.calcHist([chans[0], chans[2]], [0, 1], None,
        [32, 32], [0, 256, 0, 256])
p = ax.imshow(hist, interpolation = "nearest")
ax.set_title("2D Color Histogram for B and R")
plt.colorbar(p)

print "2D histogram shape: %s, with %d values" % (
        hist.shape, hist.flatten().shape[0])

hist = cv2.calcHist([image], [0, 1, 2],
        None, [8, 8, 8], [0, 256, 0, 256, 0, 256])
print "3D histogram shape: %s, with %d values" % (
        hist.shape, hist.flatten().shape[0])

plt.show()
```

ANEXO P

Código XVI – blurring.py

Anexo XVI: Código 16 – blurring.py

```
# Como usar
# python blurring.py --image ../images/trex.png
# python blurring.py --image ../images/beach.png

# Import the necessary packages
import numpy as np
import argparse
import cv2


ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
cv2.imshow("Original", image)

blurred = np.hstack([
	cv2.blur(image, (3, 3)),
	cv2.blur(image, (5, 5)),
	cv2.blur(image, (7, 7))])
cv2.imshow("Averaged", blurred)
cv2.waitKey(0)

blurred = np.hstack([
	cv2.GaussianBlur(image, (3, 3), 0),
	cv2.GaussianBlur(image, (5, 5), 0),
	cv2.GaussianBlur(image, (7, 7), 0)])
cv2.imshow("Gaussian", blurred)
cv2.waitKey(0)

blurred = np.hstack([
	cv2.medianBlur(image, 3),
	cv2.medianBlur(image, 5),
```

```
        cv2.medianBlur(image, 7)])
cv2.imshow("Median", blurred)
cv2.waitKey(0)

blurred = np.hstack([
        cv2.bilateralFilter(image, 5, 21, 21),
        cv2.bilateralFilter(image, 7, 31, 31),
        cv2.bilateralFilter(image, 9, 41, 41)])
cv2.imshow("Bilateral", blurred)
cv2.waitKey(0)
```

ANEXO Q

Código XVII – simple_thresholdding.py

Código XVII – simple_thresholding.py

```
# Como usar
# python simple_thresholding.py --image ../images/coins.png

import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
        help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imshow("Image", image)

 (T, threshInv) = cv2.threshold(blurred, 155, 255, cv2.THRESH_BINARY_INV)
cv2.imshow("Threshold Binary Inverse", threshInv)

(T, thresh) = cv2.threshold(blurred, 155, 255, cv2.THRESH_BINARY)
cv2.imshow("Threshold Binary", thresh)

cv2.imshow("Coins", cv2.bitwise_and(image, image, mask = threshInv))
cv2.waitKey(0)
```

ANEXO R

Código XVIII – adaptative_thresholding.py

Código XVIII – adaptative_thresholding.py

```
# Como usar
# python adaptive_thresholding.py --image ../images/coins.png

import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imshow("Image", image)

thresh = cv2.adaptiveThreshold(blurred, 255,
	cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY_INV, 11, 4)
cv2.imshow("Mean Thresh", thresh)

thresh = cv2.adaptiveThreshold(blurred, 255,
	cv2.ADAPTIVE_THRESH_GAUSSIAN_C, cv2.THRESH_BINARY_INV, 15, 3)
cv2.imshow("Gaussian Thresh", thresh)
cv2.waitKey(0)
```

ANEXO S

Código XVIX – otsu_and_riddler.py

Código XVIX– otsu_and_riddler.py

```
#Como usar
# python otsu_and_riddler.py --image ../images/coins.png

import numpy as np
import argparse
import mahotas
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imshow("Image", image)

T = mahotas.thresholding.otsu(blurred)
print "Otsu's threshold: %d" % (T)

thresh = image.copy()
thresh[thresh > T] = 255
thresh[thresh < 255] = 0
thresh = cv2.bitwise_not(thresh)
cv2.imshow("Otsu", thresh)

T = mahotas.thresholding.rc(blurred)
print "Riddler-Calvard: %d" % (T)
thresh = image.copy()
thresh[thresh > T] = 255
thresh[thresh < 255] = 0
thresh = cv2.bitwise_not(thresh)
```

```
cv2.imshow("Riddler-Calvard", thresh)
cv2.waitKey(0)
```

ANEXO T

Código XX – canny.py

Código XX – canny.py

```
# Como usar
# python canny.py --image ../images/coins.png

import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
image = cv2.GaussianBlur(image, (5, 5), 0)
cv2.imshow("Blurred", image)

canny = cv2.Canny(image, 30, 150)
cv2.imshow("Canny", canny)
cv2.waitKey(0)
```

ANEXO U

Código XXI – sobel_and_laplacian.py

Código XXI – sobel_and_laplacian.py

```
# Como usar
# python sobel_and_laplacian.py --image ../images/coins.png

import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
cv2.imshow("Original", image)

lap = cv2.Laplacian(image, cv2.CV_64F)
lap = np.uint8(np.absolute(lap))
cv2.imshow("Laplacian", lap)
cv2.waitKey(0)

sobelX = cv2.Sobel(image, cv2.CV_64F, 1, 0)
sobelY = cv2.Sobel(image, cv2.CV_64F, 0, 1)

sobelX = np.uint8(np.absolute(sobelX))
sobelY = np.uint8(np.absolute(sobelY))

sobelCombined = cv2.bitwise_or(sobelX, sobelY)

cv2.imshow("Sobel X", sobelX)
cv2.imshow("Sobel Y", sobelY)
cv2.imshow("Sobel Combined", sobelCombined)
```

ANEXO V

Código XXII – counting_coins.py

Código XXII – counting_coins.py

```
# Como usar
# python counting_coins.py --image ../images/coins.png

import numpy as np
import argparse
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-i", "--image", required = True,
	help = "Path to the image")
args = vars(ap.parse_args())

image = cv2.imread(args["image"])
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
blurred = cv2.GaussianBlur(gray, (11, 11), 0)
cv2.imshow("Image", image)

edged = cv2.Canny(blurred, 30, 150)
cv2.imshow("Edges", edged)

 (cnts, _) = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)

print "I count %d coins in this image" % (len(cnts))

coins = image.copy()
cv2.drawContours(coins, cnts, -1, (0, 255, 0), 2)
cv2.imshow("Coins", coins)
cv2.waitKey(0)

for (i, c) in enumerate(cnts):
	# We can compute the 'bounding box' for each contour, which is
	# the rectangle that encloses the contour
	(x, y, w, h) = cv2.boundingRect(c)
```

```
print "Coin #%d" % (i + 1)
coin = image[y:y + h, x:x + w]
cv2.imshow("Coin", coin)

mask = np.zeros(image.shape[:2], dtype = "uint8")
((centerX, centerY), radius) = cv2.minEnclosingCircle(c)
cv2.circle(mask, (int(centerX), int(centerY)), int(radius), 255, -1)
mask = mask[y:y + h, x:x + w]
cv2.imshow("Masked Coin", cv2.bitwise_and(coin, coin, mask = mask))
cv2.waitKey(0)
```

ANEXO W

Código XXIII – motion_detector.py

Anexo XXIII: Código 23 – motion_detector.py

```
#Como usar
# python motion_detector.py

import argparse
import datetime
import imutils
import time
import cv2

ap = argparse.ArgumentParser()
ap.add_argument("-v", "--video", help="path to the video file")
ap.add_argument("-a", "--min-area", type=int, default=500, help="minimum area size")
args = vars(ap.parse_args())

if args.get("video", None) is None:
	camera = cv2.VideoCapture(0)
	time.sleep(0.25)

else:
	camera = cv2.VideoCapture(args["video"])

firstFrame = None

while True:
	(grabbed, frame) = camera.read()
	text = "Unoccupied"

	if not grabbed:
		break

	frame = imutils.resize(frame, width=500)
	gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
```

```
	gray = cv2.GaussianBlur(gray, (21, 21), 0)

	if firstFrame is None:
		firstFrame = gray
		continue

	frameDelta = cv2.absdiff(firstFrame, gray)
	thresh = cv2.threshold(frameDelta, 25, 255, cv2.THRESH_BINARY)[1]

	thresh = cv2.dilate(thresh, None, iterations=2)
	(cnts, _) = cv2.findContours(thresh.copy(), cv2.RETR_EXTERNAL,
		cv2.CHAIN_APPROX_SIMPLE)

	for c in cnts:
		# if the contour is too small, ignore it
		if cv2.contourArea(c) < args["min_area"]:
			continue

		(x, y, w, h) = cv2.boundingRect(c)
		cv2.rectangle(frame, (x, y), (x + w, y + h), (0, 255, 0), 2)
		text = "Occupied"

	cv2.putText(frame, "Room Status: {}".format(text), (10, 20),
		cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 0, 255), 2)
	cv2.putText(frame,
datetime.datetime.now().strftime("%A %d %B %Y %I:%M:%S%p"),
	(10, frame.shape[0] - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.35, (0, 0, 255),1)
	cv2.imshow("Security Feed", frame)
	cv2.imshow("Thresh", thresh)
	cv2.imshow("Frame Delta", frameDelta)
	key = cv2.waitKey(1) & 0xFF

	if key == ord("q"):
		break

camera.release()
cv2.destroyAllWindows()
```

ANEXO Y

Código XXIV – face_detect.py

Código XXIV – face_detect.py

```
import cv2
import sys

# Get user supplied values
imagePath = sys.argv[1]
cascPath = "haarcascade_frontalface_default.xml"

# Create the haar cascade
faceCascade = cv2.CascadeClassifier(cascPath)

# Read the image
image = cv2.imread(imagePath)
gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)

# Detect faces in the image
faces = faceCascade.detectMultiScale(
    gray,
    scaleFactor=1.1,
    minNeighbors=5,
    minSize=(30, 30),
    flags = cv2.cv.CV_HAAR_SCALE_IMAGE
)

print("Found {0} faces!".format(len(faces)))

# Draw a rectangle around the faces
for (x, y, w, h) in faces:
    cv2.rectangle(image, (x, y), (x+w, y+h), (0, 255, 0), 2)

cv2.imshow("Faces found", image)
cv2.waitKey(0)
```