

1. 画像処理の基礎

本田 康祐

2020/04/17

0 python+OpenCV のプログラミング環境構築

今回の課題では、Anacondaを利用してPython環境を構築した。

Anacondaの仮想環境にOpenCVの環境を構築するため、以下のcondaコマンドを入力した。

```
conda install -c menpo opencv
```

また、統合開発環境はJupyter Notebookを使用した。

そのため、以後添付するソースコードはJupyter Notebook上のコードから1つずつ切り取ったものであり、単体では動かない。

最後に今回の実験環境を以下に示す。

- Windows 10 64-bit
- Anaconda3 20.02 Python3.7
- numpy 1.18.1
- opencv 3.4.2
- matplotlib 3.1.3

1 numpyを使った行列の四則演算

行列 A , B およびスカラー値 k を定義して、以下の5つの演算を試みる。

- 行列の和
- 行列の差
- 行列の積
- スカラー積
- アダマール積(要素ごとの積)

今回, A , B , k の値を以下の式(1)のように定める.

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, B = \begin{pmatrix} 5 & 6 \\ 7 & 8 \end{pmatrix}, k = 2 \quad (1)$$

それぞれの5つの演算の結果は以下の式(2)から式(6)のようになる

- 行列の和

$$A + B = \begin{pmatrix} 6 & 8 \\ 10 & 12 \end{pmatrix} \quad (2)$$

- 行列の差

$$A - B = \begin{pmatrix} -4 & -4 \\ -4 & -4 \end{pmatrix} \quad (3)$$

- 行列の積

$$AB = \begin{pmatrix} 19 & 22 \\ 43 & 50 \end{pmatrix} \quad (4)$$

- スカラー積

$$kA = \begin{pmatrix} 2 & 4 \\ 6 & 8 \end{pmatrix} \quad (5)$$

- アダマール積(要素ごとの積)

$$A \otimes B = \begin{pmatrix} 5 & 12 \\ 21 & 32 \end{pmatrix} \quad (6)$$

1.1 ソースコード

```
1 import numpy as np
2
3 A = np.array([[1,2],[3,4]])
4 B = np.array([[5,6], [7,8]])
5 k = 2
6
7 # 行列の和
8 print("A + B = \n{}".format(A + B))
9 # 行列の差
10 print("A - B = \n{}".format(A - B))
11 # 行列の積
12 print("A * B = \n{}".format(np.dot(A, B)))
13 # スカラー積
14 print("k * A = \n{}".format(k * A))
15 # アダマール積
16 print("A ⊗ B = \n{}".format(A * B))
```

1.2 実行結果

```
A + B =
[[ 6  8]
 [10 12]]
A - B =
[[-4 -4]
 [-4 -4]]
A * B =
[[19 22]
 [43 50]]
k * A =
[[2 4]
 [6 8]]
A @ B =
[[ 5 12]
 [21 32]]
```

2 画像の表示，縮小拡大，回転，二値化

OpenCV を用いて画像の表示，拡大，縮小，回転，二値化を行った。画像は図 1 を使用した。

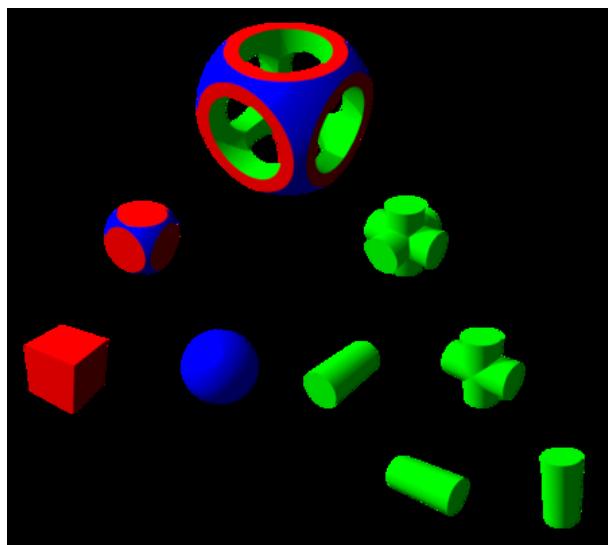


Figure 1: 画像処理に用いる画像

2.1 ソースコード

```

1 import cv2
2 from matplotlib import pyplot as plt
3 from IPython.display import Image, display
4
5 # 画面表示のメソッド
6 # jupyter_notebook 用
7 def show_img(img):
8     height, width = img.shape[:2]
9     img = cv2.imencode('.png', img)[1]
10    display(Image(img))
11    print("size:{}x{}".format(width, height))
12
13 # Python 用
14 # def show_img(img):
15 #     height, width = img.shape[:2]
16 #     print("size: ({}, {})".format(width, height))
17
18 # # RGB 画像
19 # if len(img.shape) == 3:
20 #     img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
21 #     plt.imshow(img)
22
23 # # グレースケール画像
24 # else:
25 #     plt.imshow(img, cmap="gray")
26
27 # plt.show()
28 # plt.close()
29
30 # 画像の読み込み
31 ori_img = cv2.imread("Csg_tree.png")
32 # サイズ取得
33 height, width = ori_img.shape[:2]
34 # 画像を2倍に拡大
35 big_img = cv2.resize(ori_img, (width*2,height*2))
36 # 画像を2分の1に縮小
37 small_img = cv2.resize(ori_img,(width//2, height//2))
38 # 画像を反時計回りに 90度回転
39 rotate_img = cv2.rotate(ori_img, cv2.ROTATE_90_COUNTERCLOCKWISE)
40
41 ## 二値化
42 # グレースケール画像に変換
43 gray_img = cv2.cvtColor(ori_img, cv2.COLOR_RGB2GRAY)
44 # 閾値
45 threshold = 128
46 # 二値化
47 _, binary_img = cv2.threshold(gray_img, threshold, 255, cv2.
        THRESH_BINARY)
48
49 # 画像表示
50 show_img(ori_img)
51 show_img(big_img)
52 show_img(small_img)
53 show_img(rotate_img)
54 show_img(binary_img)

```

2.2 実行結果

ソースコードの 49 行～54 行で、

1. 元画像
2. 2 倍に拡大した画像
3. 2 分の 1 に縮小した画像
4. 反時計回りに 90 度回転した画像
5. 二値化した画像

の 5 つを表示させているが、4 と 5 について図 2 と 3 に示す。

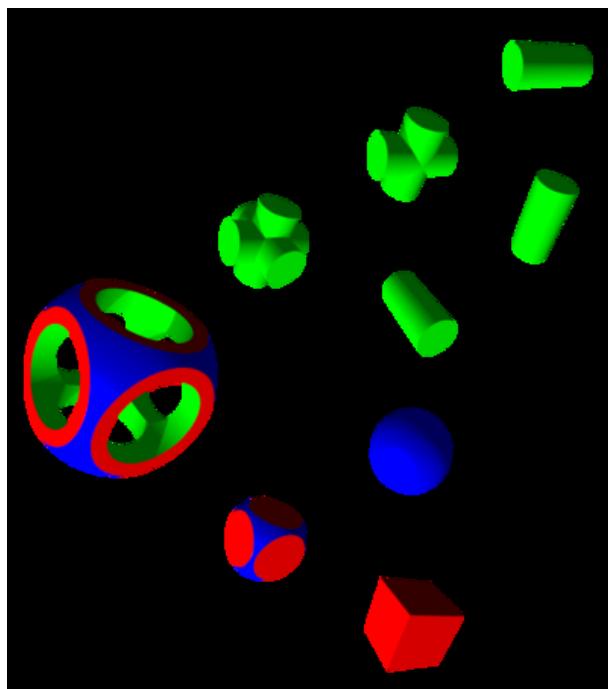


Figure 2: 反時計回りに 90 度回転した画像

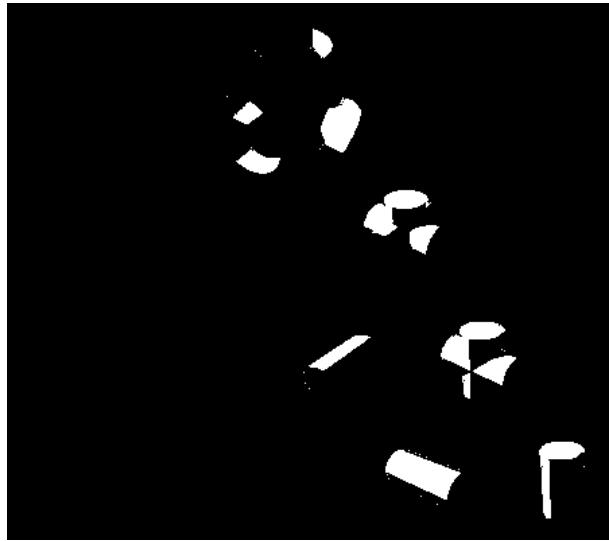


Figure 3: 二値化した画像

3 2枚の異なる画像の差分画像作成

2枚の異なる画像から差分画像を算出した。ここでは差分画像は、2つの画像の行列 A, B の差の演算結果から、負の値になった要素を 0 に置換することで算出した。

使用した画像を以下の図 4 と図 5 に示す。

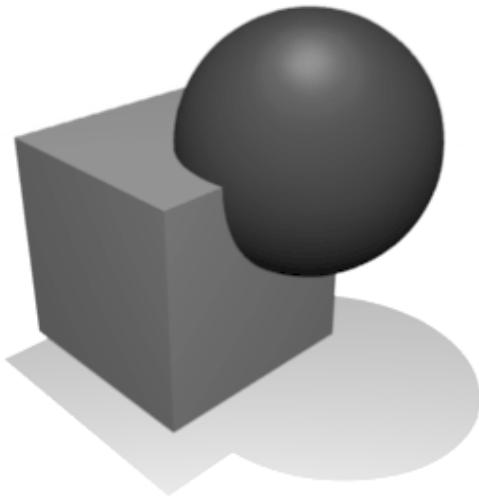


Figure 4: 画像 1

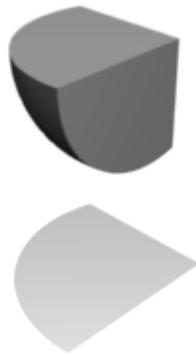


Figure 5: 画像 2

3.1 ソースコード

```
1 # 画像の読み込み (グレースケール画像に変換する)
```

```

2 | img1 = cv2.imread("Boolean_union.png", 0)
3 | img2 = cv2.imread("Boolean_intersect.png", 0)
4 |
5 | # 差分を取る(マイナスは0に置換)
6 | sub_img = img1 - img2
7 | sub_img = np.where(sub_img<0, 0, sub_img)
8 |
9 | # 画像表示
10| show_img(img1)
11| show_img(img2)
12| show_img(sub_img)

```

3.2 出力結果

出力された差分画像を図 6 に示す.

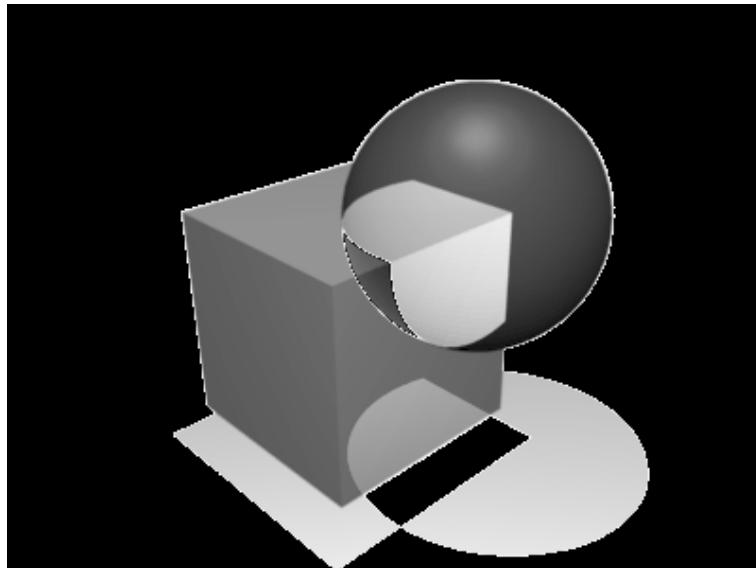


Figure 6: 1 と 2 の差分画像

4 画像の特徴量抽出と図示

4.1 ヒストグラム

画像のヒストグラムとは、画像内の R 値、B 値、G 値の度数分布を表したものである。

今回、OpenCV の calcHist 関数によりヒストグラムを作成し、Matplotlib でグラフ表示した。

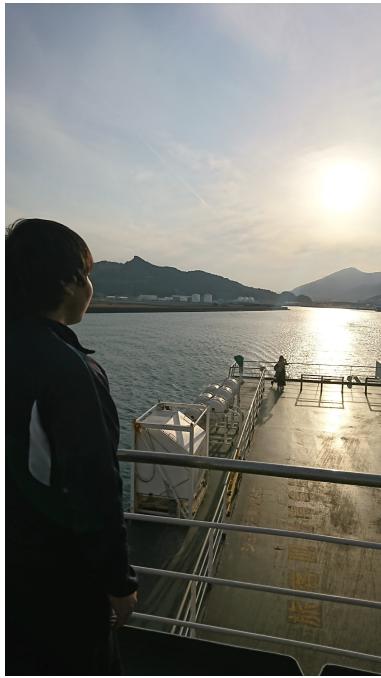


Figure 7: ヒストグラムを作成する画像

4.1.1 ソースコード

```
1 # 画像読み込み
2 photo = cv2.imread('photo.jpg')
3 # 画像表示
4 show_img(photo)
5 # ヒストグラム表示
6 color = ('b', 'g', 'r')
7 for i,col in enumerate(color):
8     histr = cv2.calcHist([photo], [i], None, [256], [0,256])
9     plt.plot(histr, color = col)
10    plt.xlim([0, 256])
11 plt.show()
12 plt.close()
```

4.1.2 出力結果

出力結果を図 8 に示す。

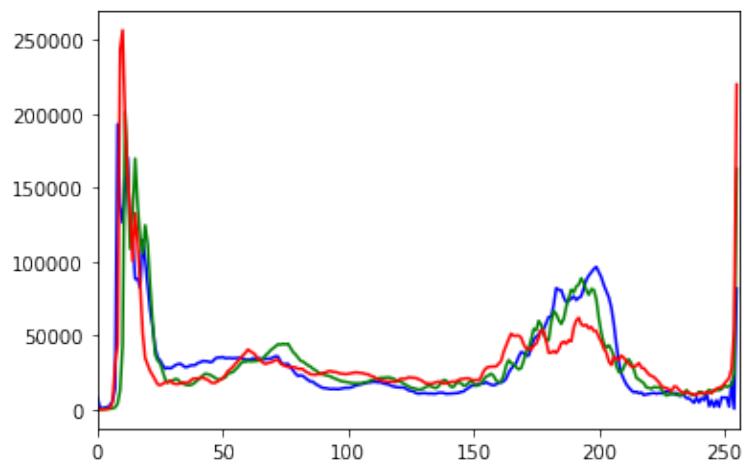


Figure 8: ヒストグラム (赤が R 値, 緑が G 値, 青が B 値)

4.2 特徴量抽出

画像から OpenCV により特徴量抽出を行う。

特徴量抽出には SURF と A-KAZE の 2 つのアルゴリズムを使用した。

画像は以下の図 9 を使用した。

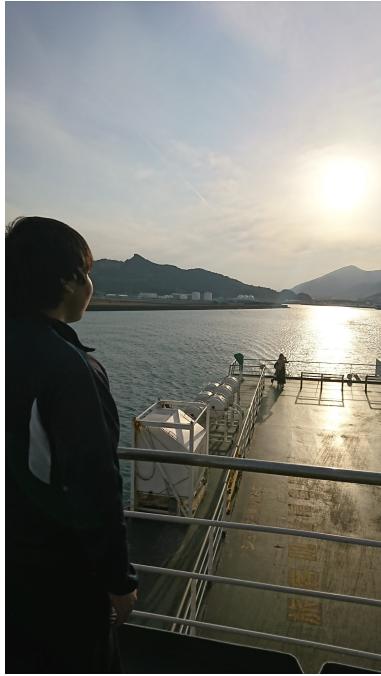


Figure 9: 特徴量抽出する画像

4.2.1 ソースコード

```
1 # SURF による特徴量抽出
2 surf = cv2.xfeatures2d.SURF_create()
3 surf_kp = surf.detect(photo)
4 surf_img = cv2.drawKeypoints(photo, surf_kp, None, flags=4)
5
6 # A-KAZE による特徴量抽出
7 akaze = cv2.AKAZE_create()
8 akaze_kp = akaze.detect(photo)
9 akaze_img = cv2.drawKeypoints(photo, akaze_kp, None, flags=4)
10
11 # 画像表示
12 show_img(surf_img)
13 show_img(akaze_img)
```

4.2.2 出力結果

特徴点はキーポイント (画像上の 1 ピクセル) と各キーポイントに対応する特徴ベクトルから構成されている。抽出された特徴量を視覚的にとらえるため、今回 OpenCV の drawKeypoints 関数により特徴ベクトルのノルムの大きさに合った円を表示させる。

SURF と A-KAZE による特徴量の抽出結果を次の図 10, 11 に示す。

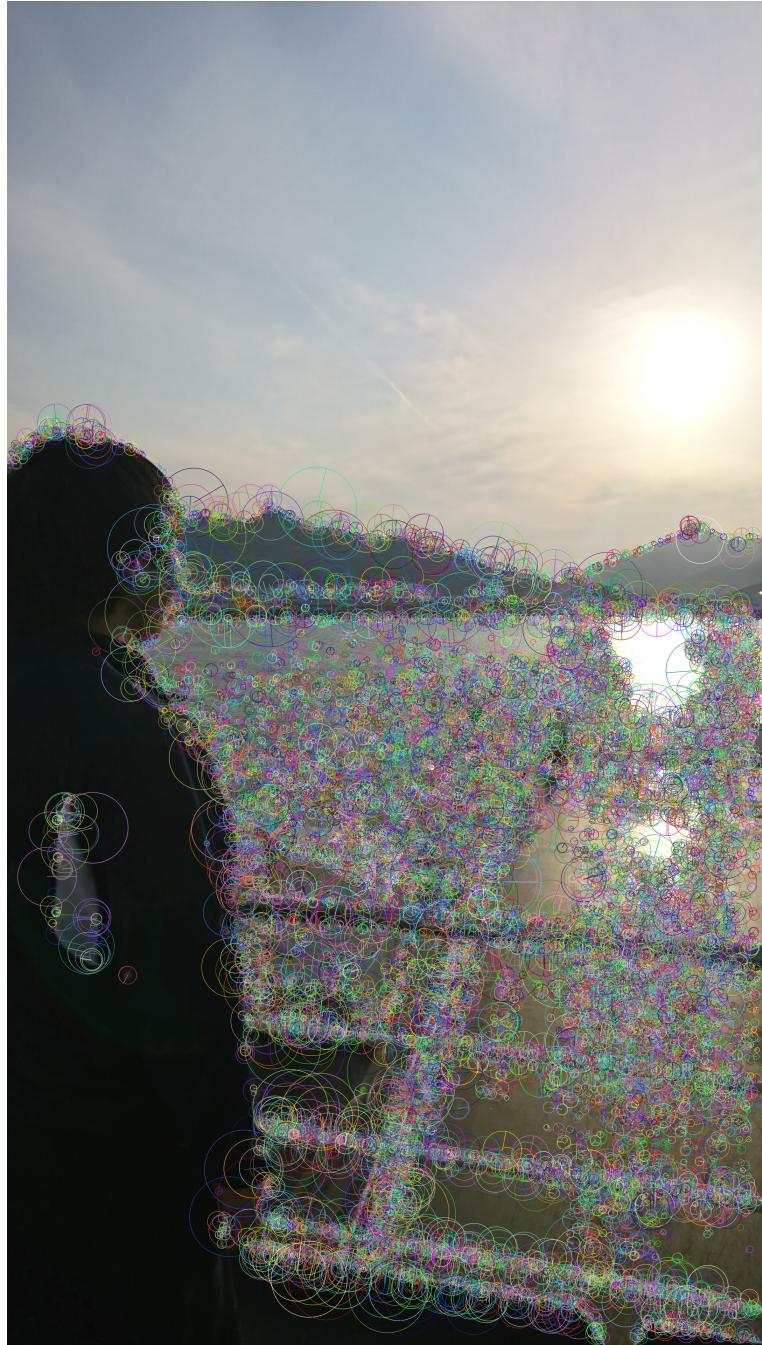


Figure 10: SURF 特徴量の抽出結果

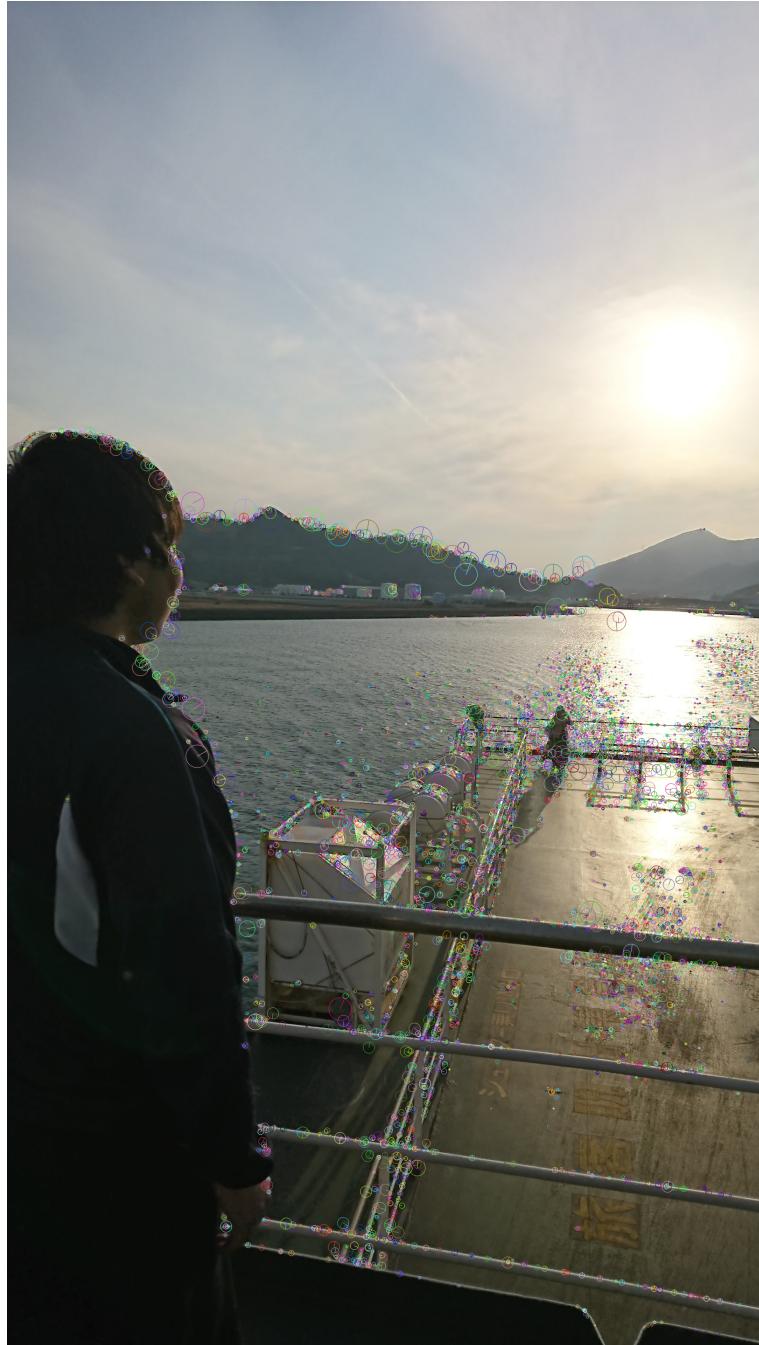


Figure 11: A-KAZE 特徴量の抽出結果

SURF の特徴点は A-KAZE よりとても多かった。A-KAZE の方がエッジの抽出が正確であると考えられる。

4.3 特徴点のマッチング

特微量抽出を用いて、入力画像からターゲット画像を検出する。

今回使用するターゲット画像および入力画像を以下の図 12 と 13 に示す。



Figure 12: ターゲット画像

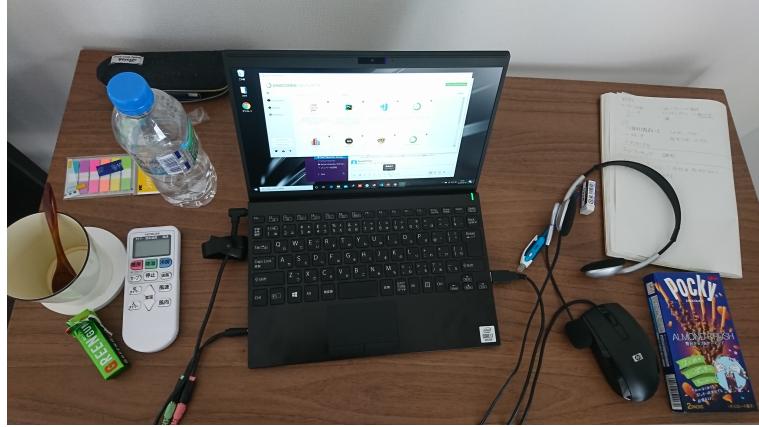


Figure 13: 入力画像

ターゲット画像であるポッキーの箱を、入力画像から検出することを目標とする。

OpenCVにより、抽出した特徴量から、総当たり(Brute-Force)アルゴリズムとFLANNアルゴリズムを用いて以下の手順によりマッチングをする。

1. ターゲット画像および入力画像から特徴量抽出を行う。
2. 総当たりアルゴリズムにより、ターゲット画像での各特徴点に対して入力画像での全ての特徴点を組み合わせて、各組の2点の特徴ベクトルの距離を算出する。
3. FLANNアルゴリズムにより、ターゲット画像の各特徴点の特徴ベクトルに最もマッチングした入力画像の上位 k 個の特徴点を返す。
4. ratio test と呼ばれるいわゆるデータの間引きを行う。割合 $ratio$ を定め、先程の上位 k 個の特徴点の組全てに対し、1位の特徴ベクトルの距離が2位の距離 $\times ratio$ 以下であるものを採用する。

以上の手順から、最後まで残ったターゲット画像と入力画像の特徴点の組をマッチング結果とする。

今回は $k = 2$, $ratio = 0.6$ で実験した。

4.3.1 ソースコード

```

1 # 特徴点のマッチングのメソッド
2 # target: ターゲット画像
3 # input_img: 入力画像
4 # algo: 特徴量抽出のアルゴリズム
5 def feature_matching(target, input_img, algo):
6     # 特徴量抽出
7     kp1, des1 = algo.detectAndCompute(target, None)
8     kp2, des2 = algo.detectAndCompute(input_img, None)

```

```

9  # 総当たりマッチング
10 bf = cv2.BFMatcher()
11 # ターゲット画像の各キーポイントの特徴量に,最もマッチングした入力画像の
12     上位k 個の特徴量を返す
13 matches = bf.knnMatch(des1, des2, k=2)
14
15 ## ratio test
16 # 1位の距離が 2位の距離のratio 以下であるものを採用, それ以外を間引き
17 ratio = 0.6
18 good = []
19 for m, n in matches:
20     if m.distance < ratio * n.distance:
21         good.append([m])
22 print("{}->{} points".format(len(matches), len(good)))
23
24 # 出力
25 output = cv2.drawMatchesKnn(target, kp1, input_img, kp2, good,
26     None, flags=2)
27 show_img(output)
28
29 # 画像の読み込み
30 target = cv2.imread("target.JPG")
31 input_img = cv2.imread("input.JPG")
32
33 # SURF による特徴マッチング
34 feature_matching(target, input_img, surf)
35 # A-KAZE による特徴マッチング
36 feature_matching(target, input_img, akaze)

```

4.3.2 出力結果

SURF と A-KAZE による特徴マッチングの結果を, 以下の図 14 と 15 に示す.

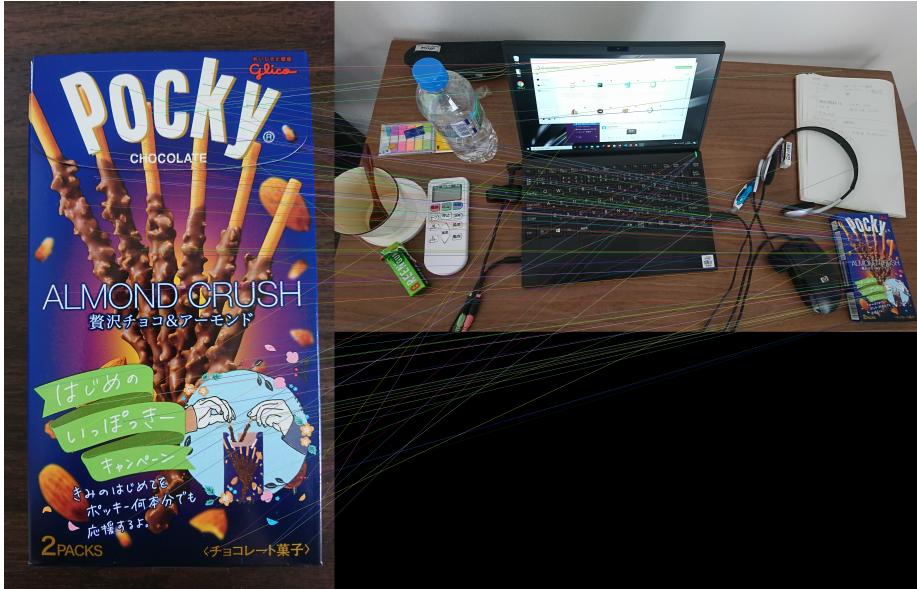


Figure 14: SURF による特徴マッチングの結果

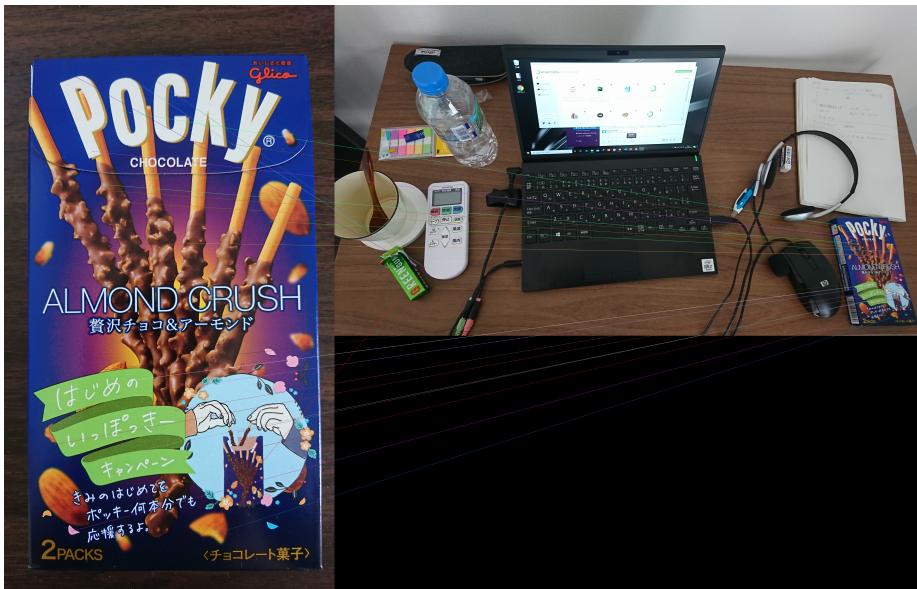


Figure 15: A-KAZE による特徴マッチングの結果

ターゲット画像と入力画像でマッチした特徴点の組の数は、SURF では 111 組、A-KAZE では 22 組となった。入力画像中のポッキーの箱との特徴点がマッ

チした割合は同じくらいであるように感じた。