

## ➤ もくじ

1. 画像の読み込みと表示
2. リサイズ
3. 色補完・グレースケール
4. ヒストグラム
  - ✓ ヒストグラム均一化
5. ガンマ変換
6. 2値化
7. アフライン変換
8. 透視変換
9. 畳み込み
  - ✓ 畳み込みの基礎
  - ✓ 平滑化
  - ✓ ガウシアンフィルタ
  - ✓ メディアンフィルタ
  - ✓ バイラテラルフィルタ

10. 画像の微分
  - ✓ Sobelフィルタ
  - ✓ ラプラシアンフィルタ
  - ✓ ラプラシアンガウシアンフィルタ
  - ✓ エッジ検出 (Canny)
11. 直線と円の検出
  - ✓ 直線の検出
  - ✓ 円の検出
12. モルフォロジー演算
  - ✓ オープニング
  - ✓ クロージング
13. 特徴抽出
  - ✓ KAZE
  - ✓ AKAZE
  - ✓ ORB
14. 色検出
15. インペイント+画像切り抜き

1. 画像の読み込みと表示
2. リサイズ

```
[146] from google.colab.patches import cv2_imshow
import cv2

img = cv2.imread("./train/bridge/bridge_000.jpeg")
```

```
[147] # 高さ、幅、カラー数となっている
img.shape
```

```
(3024, 4032, 3)
```

```
[148] # resizeの際は、幅、高さの順とする
size = (400, 300)
```

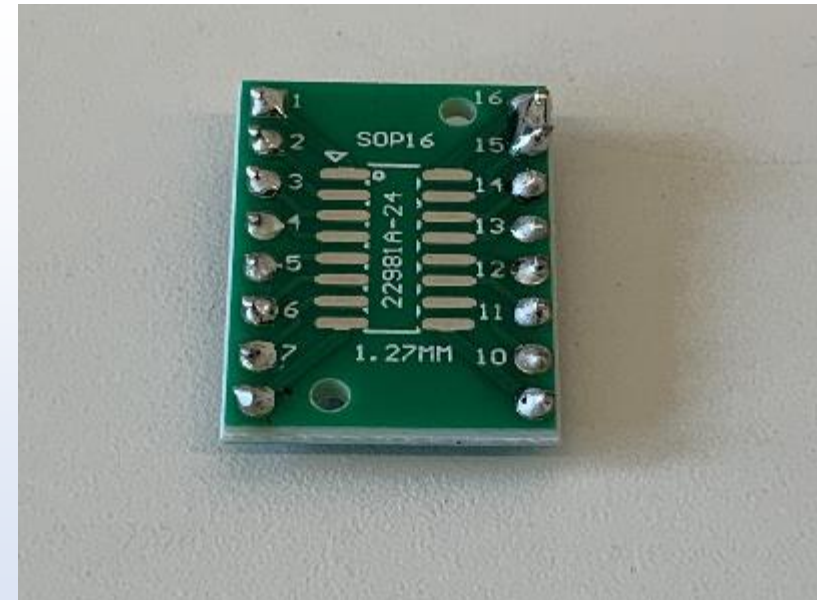
```
[149] img_resize = cv2.resize(img, size)
```

```
[150] img_resize.shape
```

```
(300, 400, 3)
```

```
▶ # 小さくなった
cv2_imshow(img_resize)
```

Google Colabの場合は、`cv2.imshow`は使用できない。  
`cv2\_imshow`をimportしないといけない。



1. 画像の読み込みと表示
2. リサイズ



# resizeの種類には色々あるらしい

```
img_area = cv2.resize(img, size, interpolation = cv2.INTER_AREA)
```

```
img_linear = cv2.resize(img, size, interpolation = cv2.INTER_LINEAR)
```

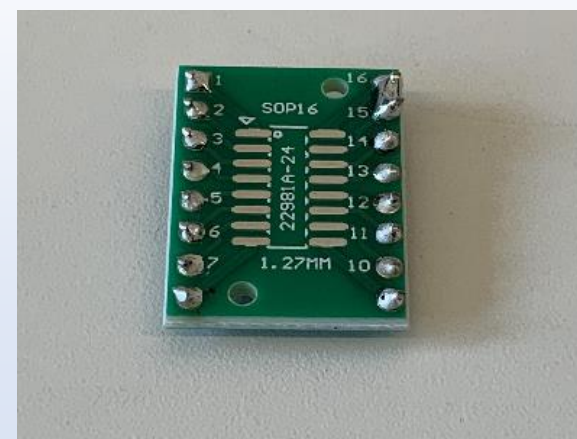
```
cv2.imshow(img_area)
```

```
cv2.imshow(img_linear) # ややちかちか
```

INTER\_AREA



INTER\_LINEAR



わかりづらいが、アップにすると違いがあるのが良くわかる。  
INTER\_LINEARの方がややちかちかする。区切りがはっきり？

## 3. 色補完・グレースケール

RGB以外にもHSVなど様々な色の表し方がある

< HSV >

- H : Hue (色相) 0-359
- S : Saturation (輝度) 0-100%
- V : Value (明度) 0-100%

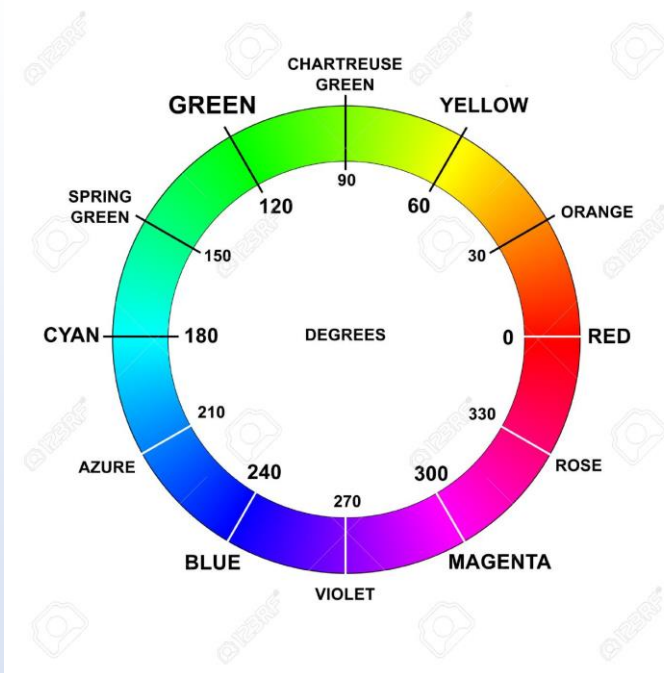
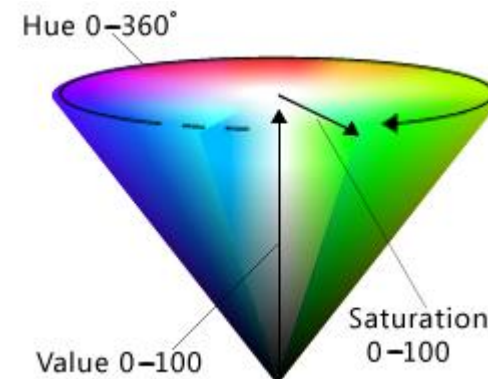
OpenCVでは

- H : Hue (色相) 0-179
  - S : Saturation (輝度) 0-255
  - V : Value (明度) 0-255
- となっている点に注意

メリットとして、色が表しやすい。  
数値範囲で取り出しやすかったりする。

<グレースケール>

Gray = 0.2989R + 0.5870G + 0.1140B  
演算負荷を抑えられる。2値化がやりやすい



## 3. 色補完・グレースケール

```
[152] img_hsv = cv2.cvtColor(img_resize, cv2.COLOR_BGR2HSV)  
      img_gray = cv2.cvtColor(img_resize, cv2.COLOR_BGR2GRAY)
```

```
[154] # 3は、hsvの3  
      img_hsv.shape
```

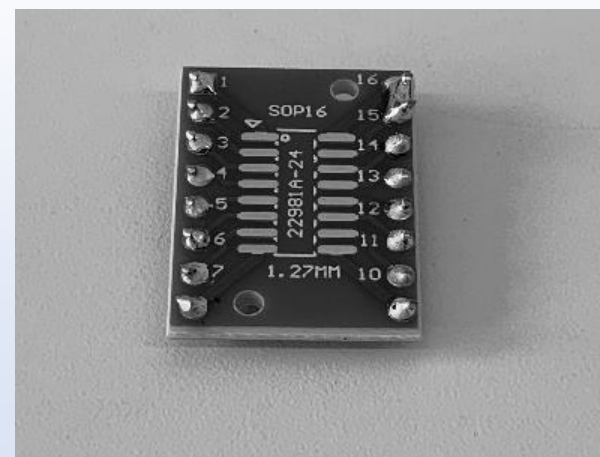
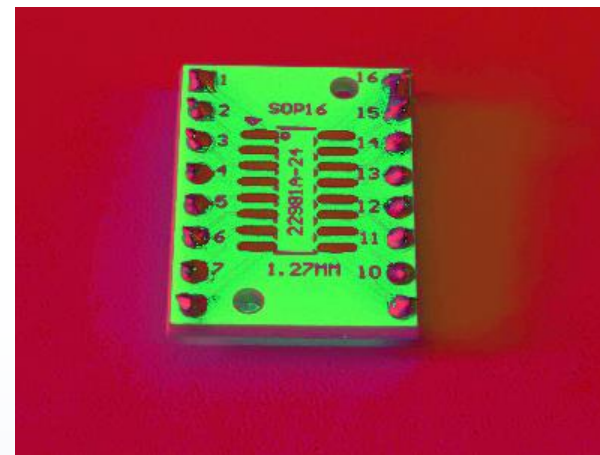
```
(300, 400, 3)
```

```
[153] # グレースケールはshapeだけの情報になっている  
      img_gray.shape
```

```
(300, 400)
```

```
▶ cv2.imshow(img_gray)  
   cv2.imshow(img_hsv)
```

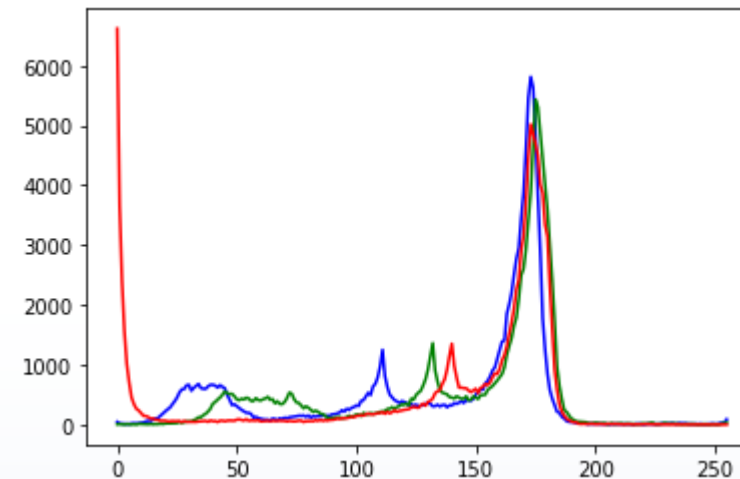
```
[ ] # 引数0にして、読み込み時にグレースケールにする方法もある  
     img_gray2 = cv2.imread("./train/bridge/bridge_000.jpeg", 0)
```



## 4. ヒストグラム

```
[156] import matplotlib.pyplot as plt
```

```
▶ color_list = ["blue", "green", "red"]  
  
for i, j in enumerate(color_list):  
    # 第3引数: マスク画像, 4-5: binの数  
    hist = cv2.calcHist([img_resize], [i], None, [256], [0, 256])  
    plt.plot(hist, color=j)
```



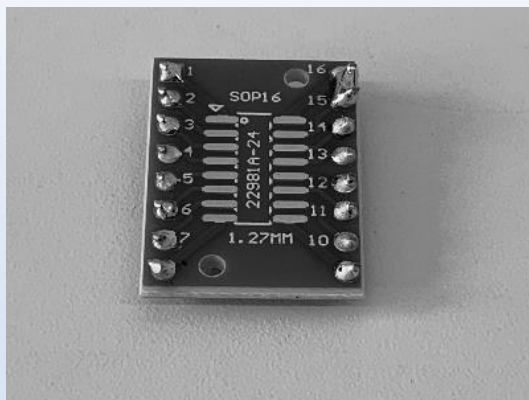
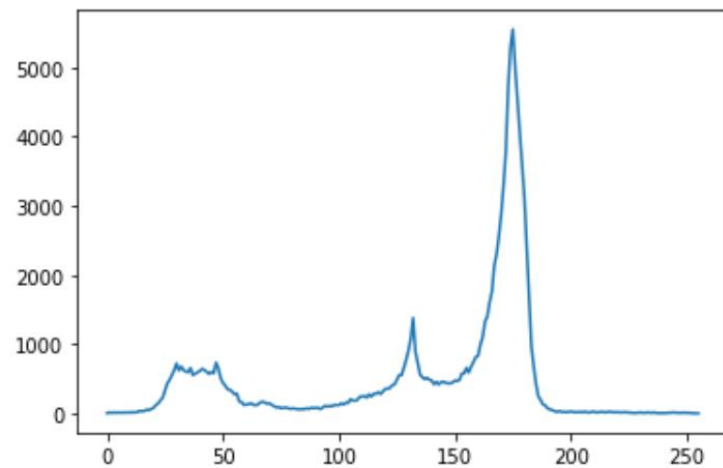
ヒストグラムは、その「点」の色の明るさをレベル別に分布したグラフ。  
ヒストグラムを見ればその画像の明るさの傾向をある程度把握することができる。



## 4. ヒストグラム均一化

```
▶ hist = cv2.calcHist([img_gray2], [0], None, [256], [0, 256])  
plt.plot(hist)
```

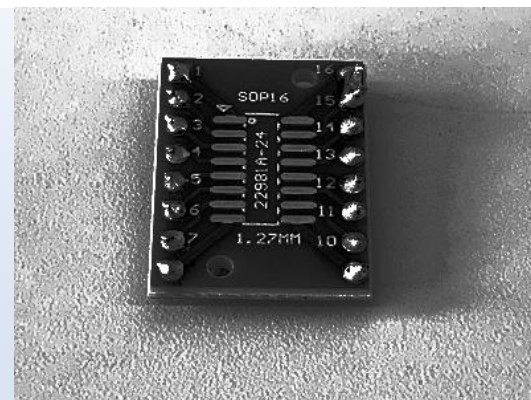
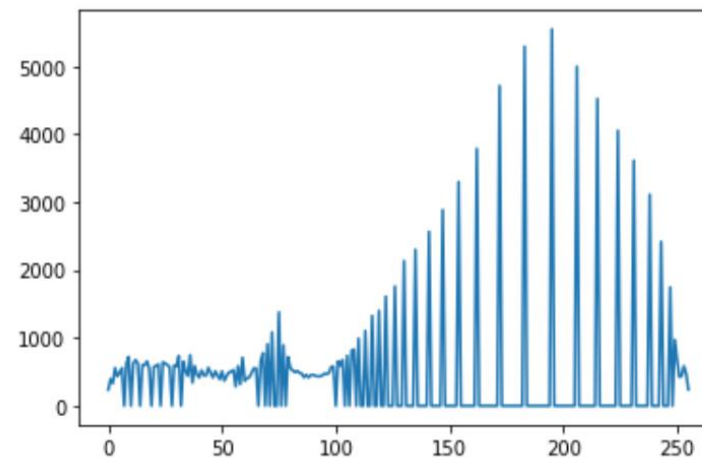
↳ [<matplotlib.lines.Line2D at 0x7f7568ddd610>]



明暗がはっきりする！

```
▶ img_eq = cv2.equalizeHist(img_gray2)  
hist_e = cv2.calcHist([img_eq], [0], None, [256], [0, 256])  
plt.plot(hist_e)
```

↳ [<matplotlib.lines.Line2D at 0x7f7568dcd050>]



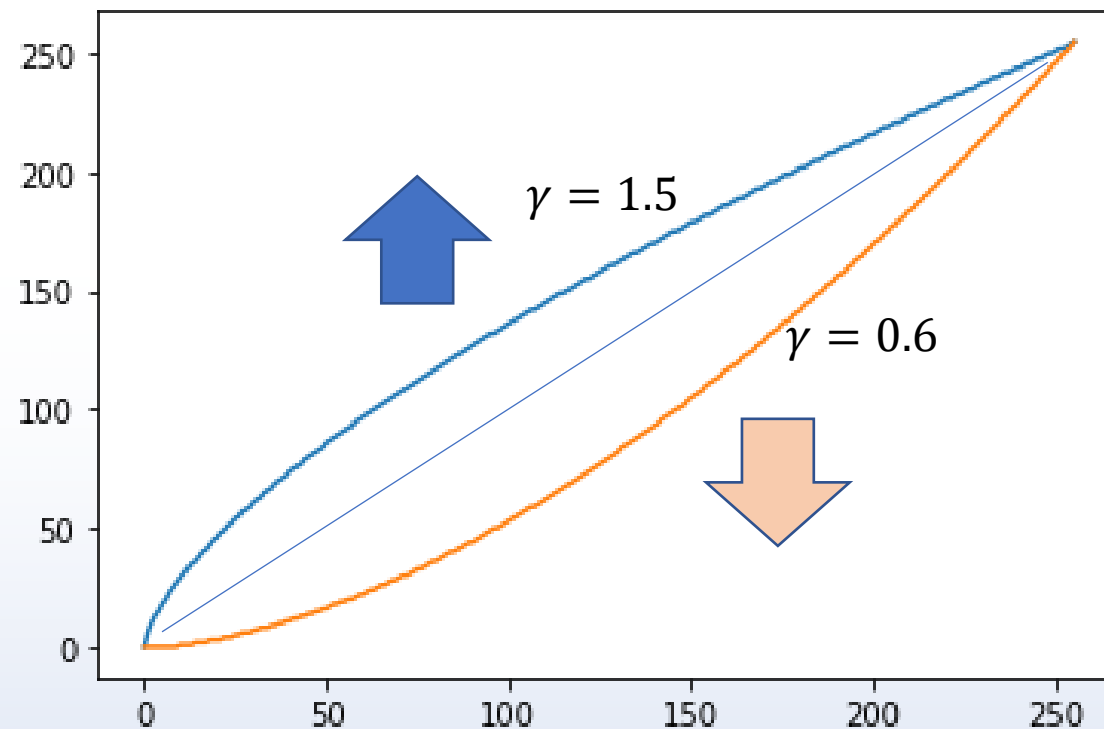
## 5. ガンマ変換

明るさの変換方法

HSVのV  
↓

$$y = 255 \left( \frac{x}{255} \right)^{\frac{1}{\gamma}}$$

$\gamma$ が1より小さい：画像が暗くなる  
 $\gamma$ が1より大きい：画像が明るくなる





## 5. ガンマ変換

```
[158] import numpy as np
import matplotlib.pyplot as plt

gamma = 1.5 # 明るくなる
gamma_cvt = np.zeros((256,1), dtype=np.uint8) # 256行1列の0ベクトル

for i in range(256):
    gamma_cvt[i][0] = 255 * (float(i)/255) ** (1.0 / gamma)
```

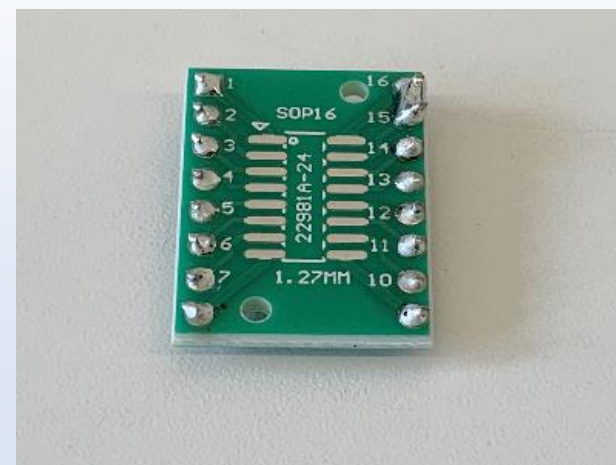
```
▶ img_gamma = cv2.LUT(img_resize, gamma_cvt)

cv2_imshow(img_resize)
cv2_imshow(img_gamma) # 明るくなった
```

ORG



$\gamma = 1.5$



## 6. 2値化

ある閾値を基準に0か1の画像に切り替える。

```
[164] img_gray = cv2.imread("./train/bridge/bridge_000.jpeg", 0)
      img_gray = cv2.resize(img_gray, size, interpolation = cv2.INTER_LINEAR)

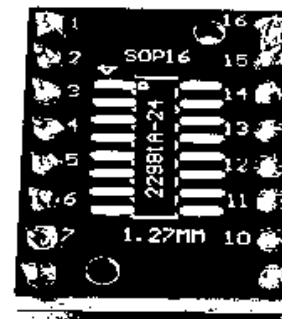
      threshold = 100

      # retにはthreshold値, img_thには画像が入る
      ret, img_th = cv2.threshold(img_gray, threshold, 255, cv2.THRESH_BINARY)
```

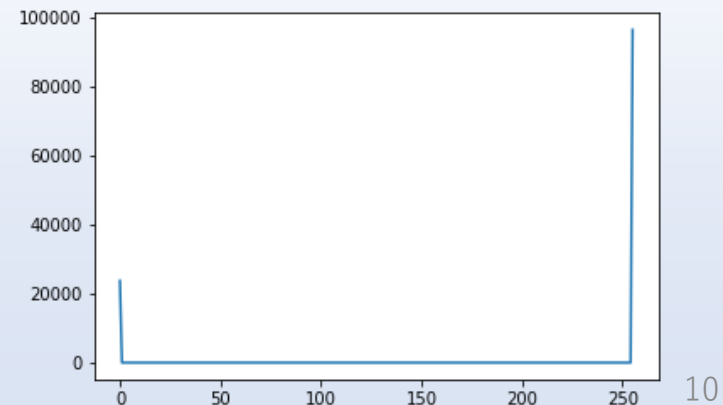
▶ ret

↗ 100.0

```
[166] cv2.imshow(img_th)
```



ヒストグラムはこんな




## 7. アフィン変換

画像の回転や、移動を行う。  
変換用の行列を定義して座標点を変換する。

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a & b \\ c & d \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} = \begin{pmatrix} a & b & t_x \\ c & d & t_y \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

これ



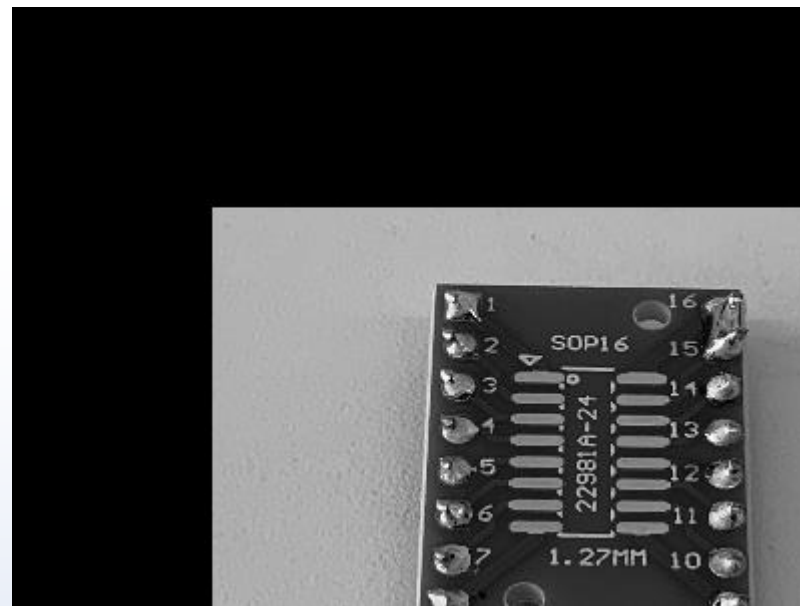
## 7. アフィン変換 (平行移動)

```
[174] # 画素を定義  
      h, w = img_gray.shape  
      # 平行移動の場合、tx, tyを定義  
      tx, ty = 100, 100
```

```
[175] # 変換行列の定義  
      # cv2上は、a, b, c, dとtx, tyの定義をすればよい  
      afn_mat = np.float32([[1, 0, tx], [0, 1, ty]])  
      img_afn = cv2.warpAffine(img_gray, afn_mat, (w, h))
```



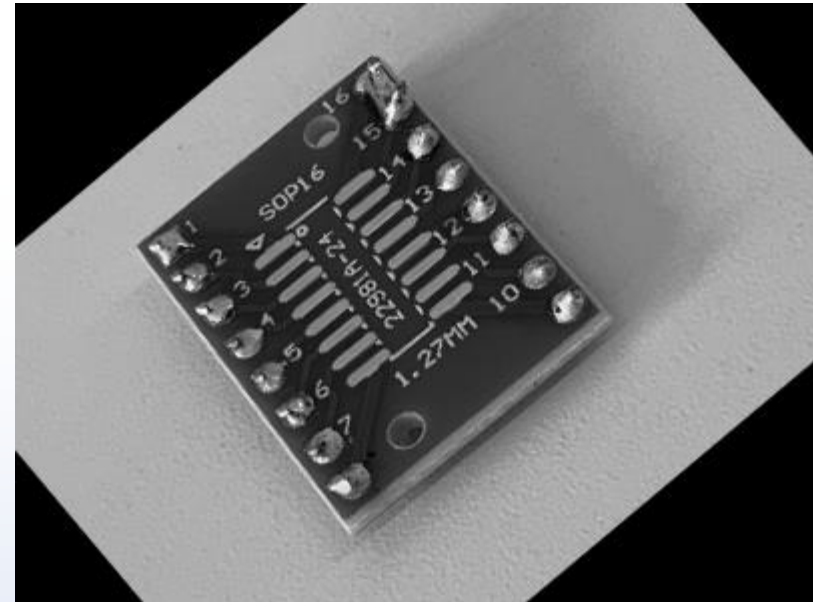
```
cv2.imshow(img_afn) # 右100下100に平行移動
```



## 7. アフィン変換 (回転)

```
▶ # 回転の場合、cv2.getRotationMatrix2Dを用いるのが楽  
# 引数1:回転中心, 引数2:回転角度, 引数3:スケール  
rot_mat = cv2.getRotationMatrix2D((w/2, h/2), 40, 1)  
img_afn2 = cv2.warpAffine(img_gray, rot_mat, (w, h))
```

```
[178] cv2.imshow('img_afn2')
```



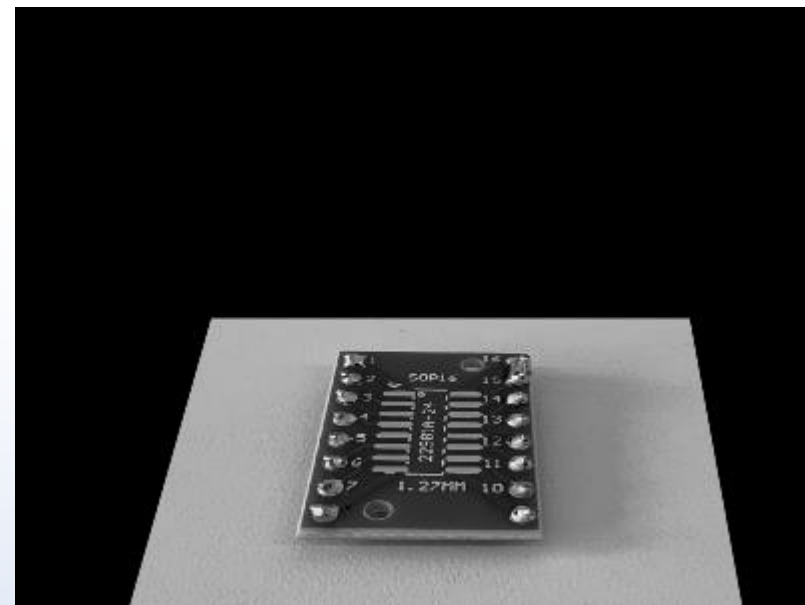
## 8. 透視変換

奥行きを出したりする変換。

```
▶ # 画像のshapeを格納しておく
h, w = img_gray.shape

# 変換する際の画像の対応関係を定義する（どの点がどこへ移動するのか）
# 元の点
per1 = np.float32([[100, 500], [300, 500], [300, 100], [100, 100]])
# 変換後の点
per2 = np.float32([[100, 500], [300, 500], [280, 200], [150, 200]])

psp_matrix = cv2.getPerspectiveTransform(per1, per2)
img_psp = cv2.warpPerspective(img_gray, psp_matrix, (w, h))
cv2.imshow(img_psp)
```



## 9. 畳み込み

周囲の情報を使った、自分の画素値の更新

＜畳み込みの流れ＞

- ① フィルターを用意
- ② 着目画素の周囲で（画素値）＊（フィルター）を行い足していく
- ③ すべての画素について畳み込みを行う。

これにより、周囲の情報を取り込んで、値が更新される

（ノイズが取り除かれる場合＝平滑化フィルタ）  
（縦横のエッジを検出＝Sobelフィルタ）。

他にもいろんなフィルターがある？

平均化フィルター

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

3×3画素

ガウシアンフィルター

1/16	2/16	1/16
2/16	4/16	2/16
1/16	2/16	1/16

3×3画素

モーションフィルター

1/3	0	0
0	1/3	0
0	0	1/3

3×3画素

1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25
1/25	1/25	1/25	1/25	1/25

5×5画素

1/256	4/256	6/256	4/256	1/256
4/256	16/256	24/256	16/256	4/256
6/256	24/256	36/256	24/256	6/256
4/256	16/256	24/256	16/256	4/256
1/256	4/256	6/256	4/256	1/256

5×5画素

1/5	0	0	0	0
0	1/5	0	0	0
0	0	1/5	0	0
0	0	0	1/5	0
0	0	0	0	1/5

5×5画素

微分フィルター

0	0	0
-1	1	0
0	0	0

x方向

0	-1	0
0	1	0
0	0	0

y方向

Prewittフィルター

-1	0	1
-1	0	1
-1	0	1

x方向

-1	-1	-1
0	0	0
1	1	1

y方向

Sobelフィルター

-1	0	1
-2	0	2
-1	0	1

x方向

-1	-2	-1
0	0	0
1	2	1

y方向



## 9. 畳み込み (平滑化フィルタ)

```
[8] kernel = np.ones((3, 3)) / 9.0  
kernel
```

```
array([[0.11111111, 0.11111111, 0.11111111],  
       [0.11111111, 0.11111111, 0.11111111],  
       [0.11111111, 0.11111111, 0.11111111]])
```

平均化フィルター

1/9	1/9	1/9
1/9	1/9	1/9
1/9	1/9	1/9

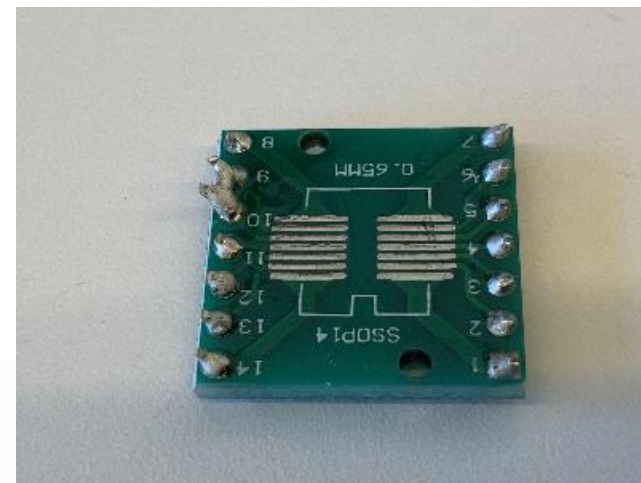
3×3 画素

```
[13] size = (400, 300)  
img = cv2.imread("./train/bridge/bridge_010.jpeg")  
img_resize = cv2.resize(img, size)
```

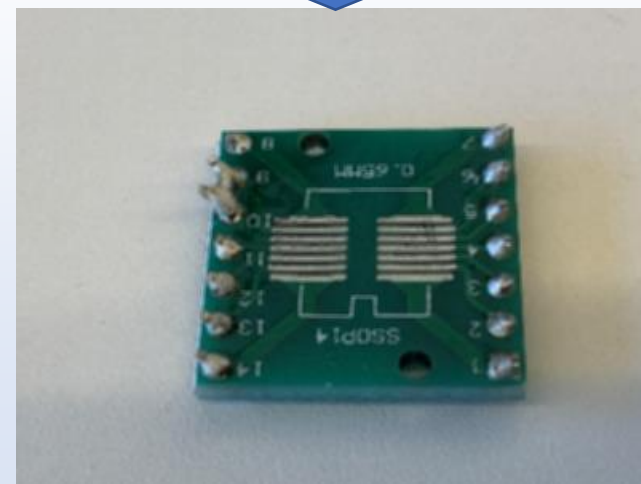
▶ # 第2引数: ビット深度? 負の値を入れておくと、  
# 元の画像を返すので-1を使うことが多いらしい  
img\_ke1 = cv2.filter2D(img\_resize, -1, kernel)  
cv2\_imshow(img\_resize)  
cv2\_imshow(img\_ke1) #ボケた

これも同じ処理。こちらの方が簡単?

▶ # kernel = np.ones((3, 3)) / 9.0と同じ機能⇒cv2.blur  
img\_bulr = cv2.blur(img\_resize, (3, 3))  
cv2\_imshow(img\_bulr)



平滑化



## 9. 畳み込み (エッジ検出)

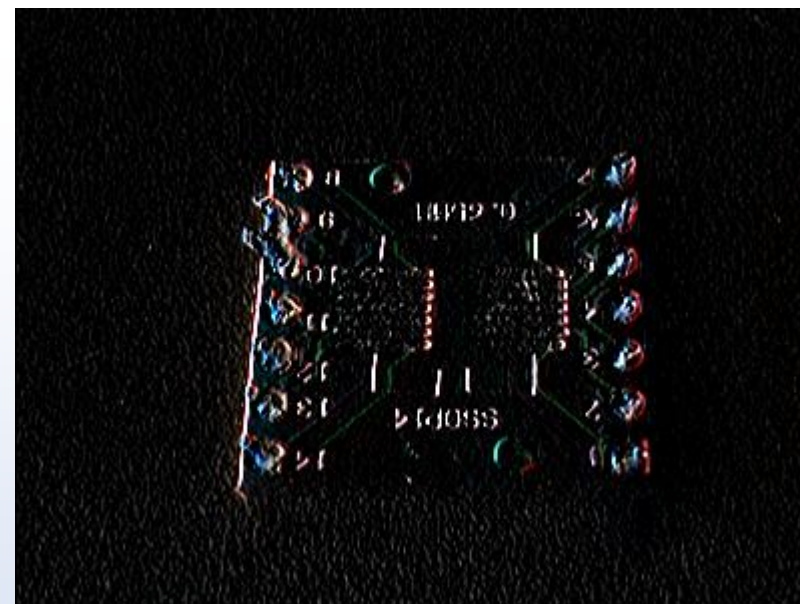
```
▶ # Sobelフィルタ
kernel2 = np.zeros((3, 3))
kernel2[0, 0] = 1
kernel2[1, 0] = 2
kernel2[2, 0] = 1
kernel2[0, 2] = -1
kernel2[1, 2] = -2
kernel2[2, 2] = -1
```

kernel2

```
➡ array([[ 1.,  0., -1.],
         [ 2.,  0., -2.],
         [ 1.,  0., -1.]])
```

```
[17] img_ke2 = cv2.filter2D(img_resize, -1, kernel2)
      cv2_imshow(img_ke2) #横エッジ検出
```

横エッジ検出



## 9. 畳み込み (ガウシアンフィルタ)

ガウス分布を利用して「注目画素からの距離に応じて近傍の画素値に重みをかける」という処理を行い、自然な平滑化を実現



# ガウシアンフィルタ

# 第2引数: フィルタの大きさ、奇数じゃないとエラー, 第3引数: ガウス関数の標準偏差

```
img_ga = cv2.GaussianBlur(img_resize, (9, 9), 2)
```

```
cv2_imshow(img_ga)
```

### ガウシアンフィルタ

1/16	2/16	1/16
2/16	4/16	2/16
1/16	2/16	1/16

3×3 画素

1/256	4/256	6/256	4/256	1/256
4/256	16/256	24/256	16/256	4/256
6/256	24/256	36/256	24/256	6/256
4/256	16/256	24/256	16/256	4/256
1/256	4/256	6/256	4/256	1/256

5×5 画素



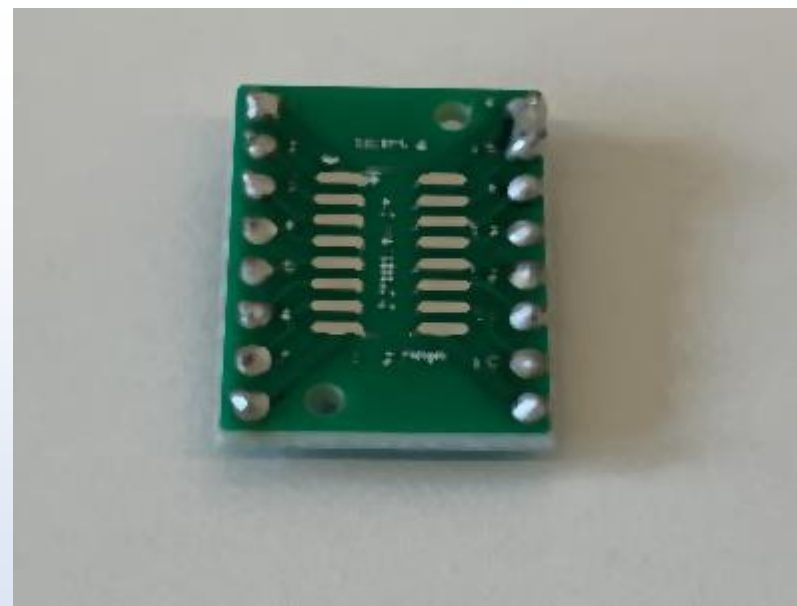
## 9. 畳み込み（メディアンフィルタ）

ある画素を、周りの画素の濃度の中央値に変換する。メディアンフィルタは、画像を平滑化することなく、画像のエッジ部分をそのまま残してノイズが除去できるという特徴



```
# medianフィルタ  
# 中央値で塗りつぶす  
img_me = cv2.medianBlur(img_resize, 5)  
cv2_imshow(img_me)
```

2	1	3		2	1	3
1	8	4	⇒	1	2	4
3	1	2		3	1	2

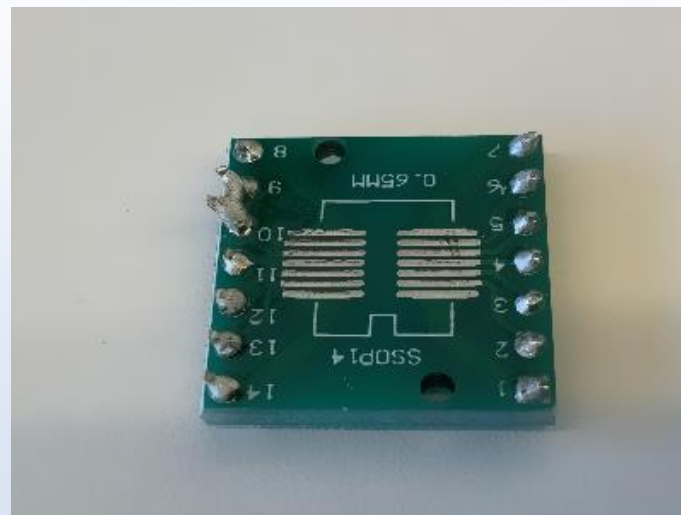
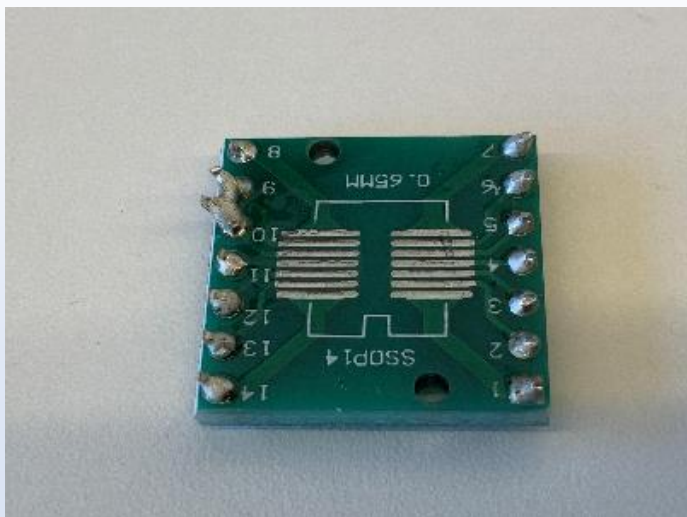


## 9. 畳み込み（バイラテラルフィルタ）

非線形のフィルタ。ガウシアンフィルタなどのフィルタでは、ノイズをできるだけ除去しようとする、輪郭もボケてしまうという欠点があるが、輝度変化が激しいところは残し、それ以外のところ変化がなだらかなところは平滑化を行うようなフィルタ。



```
# 第2引数：窓の大きさ, 第3引数：エッジを保存する $\sigma$ , 第4引数：ガウシアンフィルタの $\sigma$   
img_bi = cv2.bilateralFilter(img_resize, 20, 30, 30)  
cv2_imshow(img_resize)  
cv2_imshow(img_bi)
```





## 10.画像の微分

傾き=隣の画素値-自分の画素値

微分すると、隣の画素値との差分だけが残るので画像のエッジを含んだ部分が残る。

(Sobelフィルタ)

```
▶ # 第2引数: ビット深度、第3-4引数: 1,0でx方向の微分, 第5引数: カーネルサイズ  
img_sobelx = cv2.Sobel(img_resize, cv2.CV_32F, 1, 0, ksize=3)  
img_sobely = cv2.Sobel(img_resize, cv2.CV_32F, 0, 1, ksize=3)
```

# 0-255の値に変換

```
img_sobelx = cv2.convertScaleAbs(img_sobelx)  
img_sobely = cv2.convertScaleAbs(img_sobely)
```

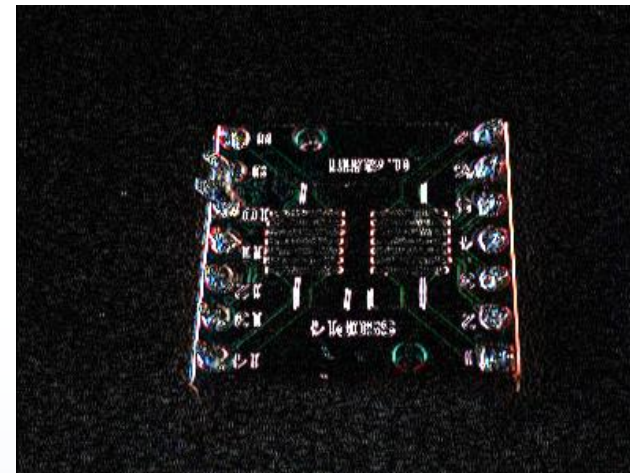
# x方向の微分

```
cv2.imshow('img_sobelx')
```

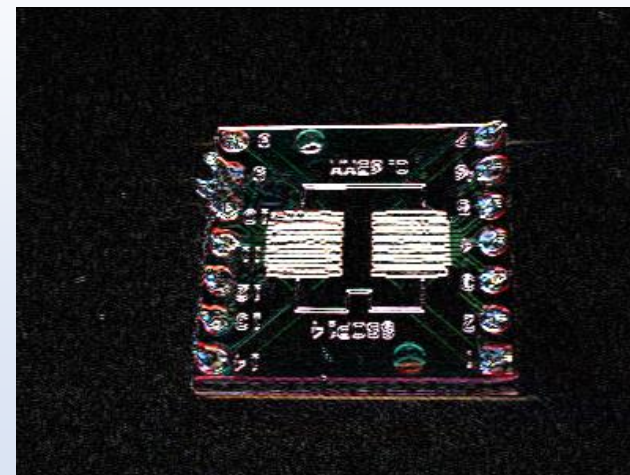
# y方向の微分

```
cv2.imshow('img_sobely')
```

X方向



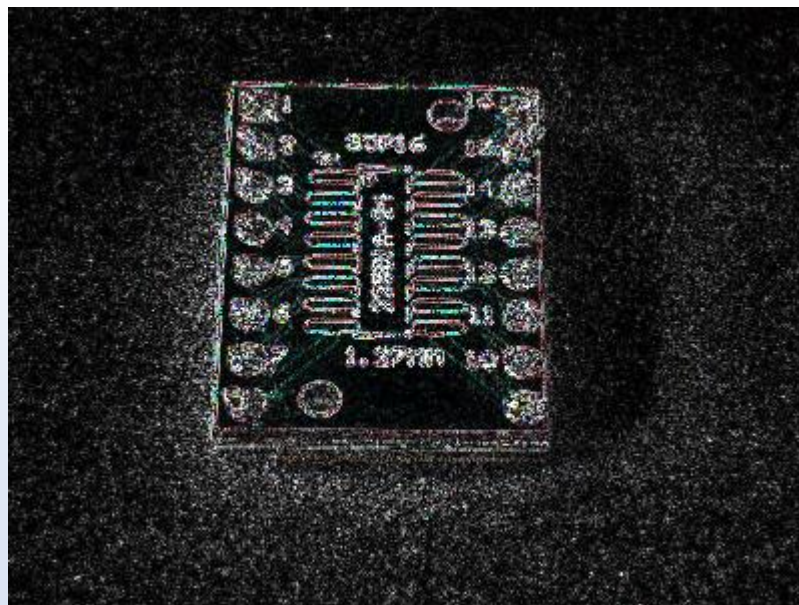
Y方向



## 10.画像の微分（ラプラシアンフィルタ）

二次微分を利用して画像から輪郭を抽出する。全方向のエッジ検出。

```
img_lap = cv2.Laplacian(img_resize, cv2.CV_32F)  
img_lap = cv2.convertScaleAbs(img_lap)  
img_lap *= 2 # 2次の微分なので値そのものが小さくなる。強くしたいなら2倍とかにする  
cv2.imshow('img_lap')
```



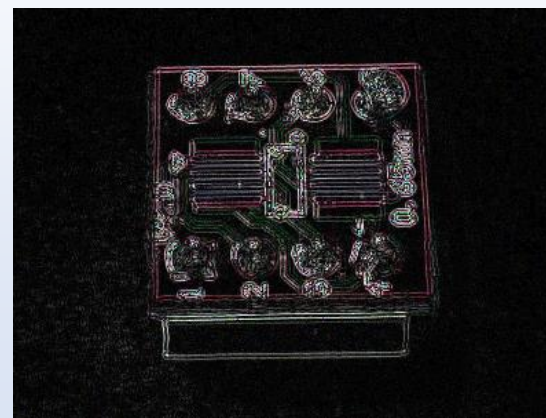
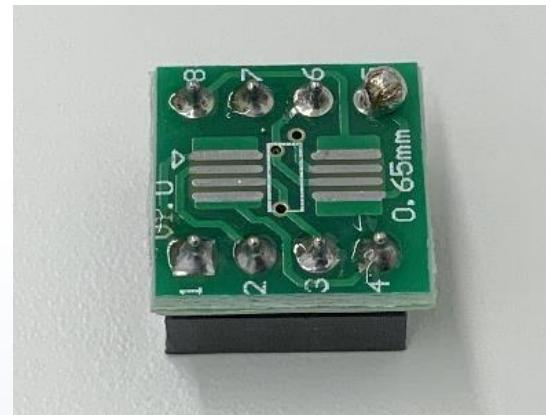


## 10.画像の微分（ラプラシアンガウシアンフィルタ）

ラプラシアンだけだと、ノイズが残っていることがある。ガウシアンフィルタを乗せることで解消

```
▶ img = cv2.imread("./train/potato/potato_012.jpeg")  
img_resize = cv2.resize(img, size)  
  
img_blur = cv2.GaussianBlur(img_resize, (3, 3), 2)  
img_lap2 = cv2.Laplacian(img_blur, cv2.CV_32F)  
img_lap2 = cv2.convertScaleAbs(img_lap2)  
img_lap2 *= 4  
cv2_imshow(img_resize)  
cv2_imshow(img_lap2)
```

たしかにラプラシアンのみよりもノイズが少ない



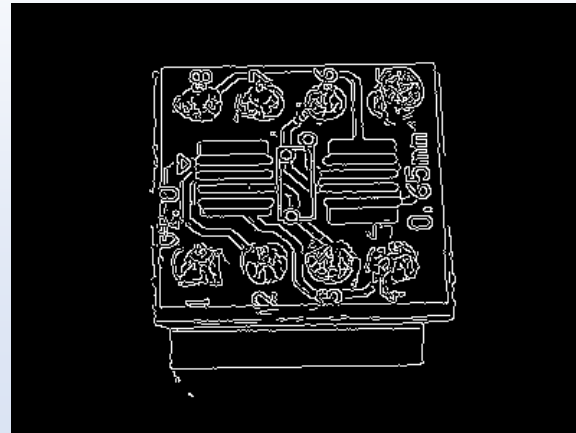
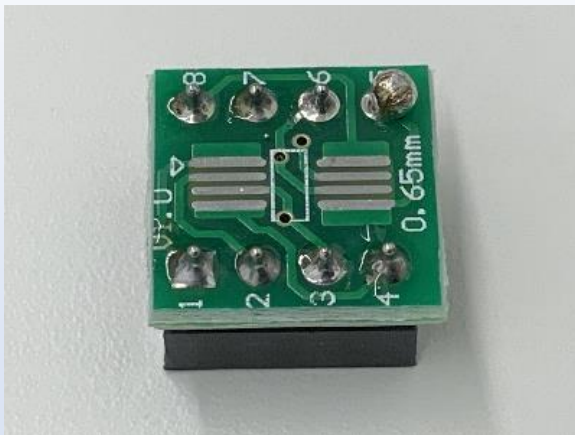
## 10.エッジ検出 (Canny)

Cannyのアルゴリズムは4段階

- ① ガウシアンフィルタでぼかす
- ② Sobelフィルタで微分
- ③ 極大点を探す
- ④ 2段階の閾値処理でエッジを残す



```
img_canny = cv2.Canny(img_resize, 100, 200)  
cv2.imshow('img_canny')
```



## 11.直線の検出

原点から直線までの距離  $\rho$ 、 $x$  軸から直線までの角度  $\theta$ を使って、全ての点の  $\rho$  と  $\theta$  を集め、一番多かったパラメータを直線とする。

$$y = -\frac{\cos\theta}{\sin\theta}x + \frac{\rho}{\sin\theta}$$

- ① 画素のある点を探す
- ②  $\theta$  をさまざまな値に設定し、 $\rho$  を計算
- ③ すべての点で①②を繰り返す
- ④  $\rho$  と  $\theta$  の出現頻度を集計
- ⑤ 出現頻度の最も多いものが直線

このページが詳しい

<http://www.allisone.co.jp/html/Notes/image/Hough/index.html>

## 11.直線の検出

```
[22] import cv2
import numpy as np
from google.colab.patches import cv2_imshow

img = cv2.imread("./train/potato/potato_000.jpeg")
img = cv2.resize(img, size)
img_g = cv2.imread("./train/potato/potato_000.jpeg", 0)
img_g = cv2.resize(img_g, size)
```

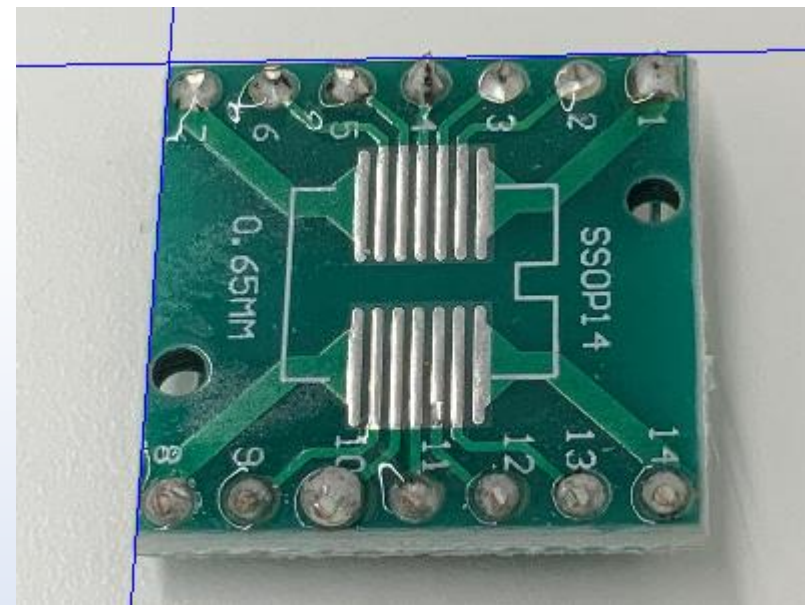
```
▶ img_canny = cv2.Canny(img_g, 300, 450)
cv2_imshow(img_canny)
```

```
[24] # 第2引数: rhoの刻み幅, 第3引数: thetaの刻み幅, 第4引数: rhoとthetaの組み合わせの閾値
lines = cv2.HoughLines(img_canny, 1, np.pi/180, 100)
```

```
▶ for i in lines[:]:
    rho = i[0][0]
    theta = i[0][1]
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = rho * a
    y0 = rho * b
    x1 = int(x0 + 1000 * (-b))
    y1 = int(y0 + 1000 * (a))
    x2 = int(x0 - 1000 * (-b))
    y2 = int(y0 - 1000 * (a))
    cv2.line(img, (x1, y1), (x2, y2), (255, 0, 0), 1)

cv2_imshow(img)
```

この画像では上手く検出できず



## 11.円の検出

```
[59] img = cv2.imread("./train/horn/horn_000.jpeg")  
img = cv2.resize(img, size)  
img_g = cv2.imread("./train/horn/horn_000.jpeg", 0)  
img_g = cv2.resize(img_g, size)
```

```
[60] # ?を付けると関数helpが出る  
circles = cv2.HoughCircles?
```

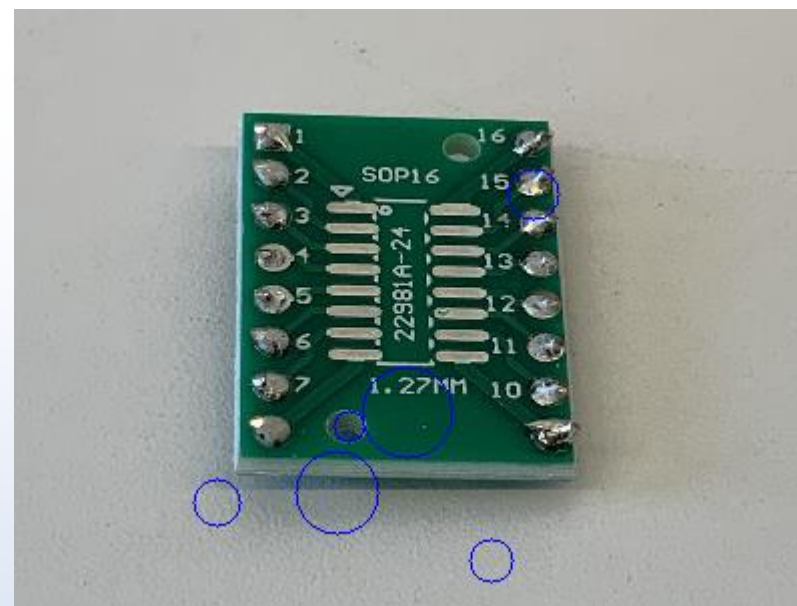
```
[61] circles = cv2.HoughCircles(img_g, cv2.HOUGH_GRADIENT,  
                                dp=1, minDist=1, param1=4,  
                                param2=35, minRadius=1, maxRadius=30)
```

```
[62] circles
```

```
array([[ [196.5, 201.5, 23.9],  
        [161.5, 241.5, 21.7],  
        [167.5, 208.5, 8.2],  
        [259.5, 92.5, 13.8],  
        [101.5, 246.5, 12.7],  
        [238.5, 275.5, 11.6]]], dtype=float32)
```

```
[63] for i in circles[0]:  
    #中心i[0]i[1], 半径i[2],  
    cv2.circle(img, (i[0], i[1]), i[2], (255, 0, 0), 1)  
cv2_imshow(img) # 調整難しい、、、
```

この画像では上手く検出できず  
調整難しい？



## 11.モルフォジー演算

膨張と、収縮からなる。ピクセルを収縮させたり、膨張させることによって、図形を分離したり、協調させたりできる。二つを組み合わせることにより、オープニング、クロージングという手法が成り立つ

- ✓ オープニング：収縮⇒膨張 ノイズが消える
- ✓ クロージング：膨張⇒収縮 大きさはそのまま。形状保ちノイズを消す

このページが詳しい

<https://pystyle.info/opencv-morphology-operation/>

## 11.モルフォジー演算（クロージング）

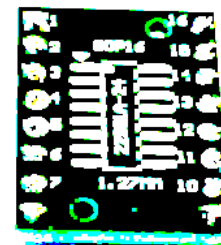
```
▶ img = cv2.imread("./train/horn/horn_000.jpeg")  
img = cv2.resize(img, size)  
img_g = cv2.imread("./train/horn/horn_000.jpeg", 0)  
img_g = cv2.resize(img_g, size)  
  
ret, img_th = cv2.threshold(img, 110, 255, cv2.THRESH_BINARY)
```

```
▶ kernel = np.ones((3,3), dtype=np.uint8) # 自分の画素から探す領域  
img_d = cv2.dilate(img_th, kernel) # 膨張  
img_e = cv2.erode(img_th, kernel) # 収縮
```

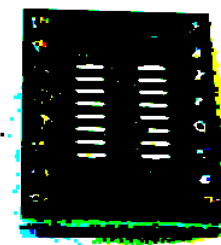
```
cv2.imshow(img_d)  
cv2.imshow(img_e)
```

```
▶ # クロージング  
img_c = cv2.morphologyEx(img_th, cv2.MORPH_CLOSE, kernel)  
cv2.imshow(img_c) # 形状を保ちつつノイズを消している
```

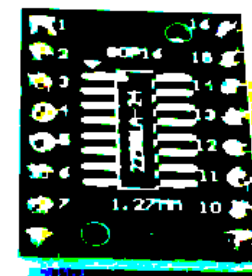
膨張



収縮



クロージング  
(膨張⇒収縮)





## 12.特徴抽出

特徴とは、パターン認識に役立つ情報量の多い部分の事。

画像の情報量は一般的に コーナー > エッジ > フラット の順で情報量が多い

フラットな領域は固有値の大きいものがない。

エッジがあると特定の固有ベクトルの固有値だけ大きくなる。

コーナーはエッジが2つあるので、2つの大きな固有値ベクトルがある。

特徴抽出、記述器には様々な種類のものがある。

- ORB
- KAZE
- AKAZE

などは特許問題もなく使いやすい。らしい

## 12.特徴抽出 (Harrisコーナー検出)

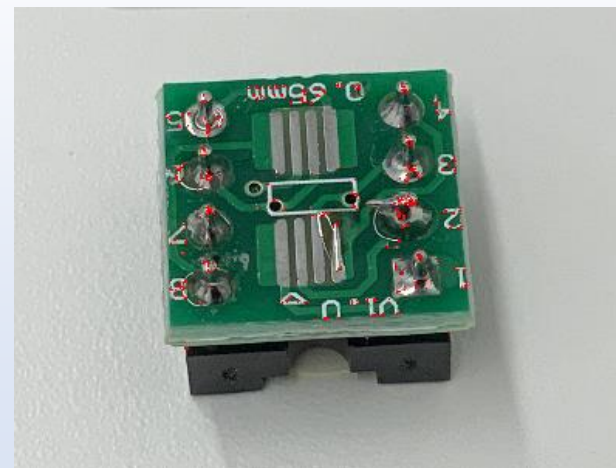
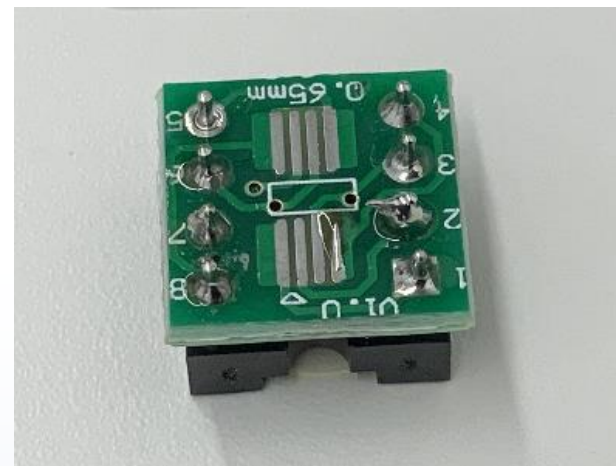
```
[69] import copy

size = (400, 300)
img = cv2.imread("./train/horn/horn_011.jpeg")
img = cv2.resize(img, size)
img_g = cv2.imread("./train/horn/horn_011.jpeg", 0)
img_g = cv2.resize(img_g, size)
```

```
[70] img_harris = copy.deepcopy(img) # 特徴点を上書きしないようコピーしておく
# 2:blocksizes, 3:sobelフィルタの大きさ
img_dst = cv2.cornerHarris(img_g, 2, 3, 0.08)
```

```
▶ # maxの3%を超えていれば特徴点だとして抜き出す。
img_harris[img_dst > 0.03 * img_dst.max()] = [0, 0, 255]
```

```
cv2_imshow(img)
cv2_imshow(img_harris)
```

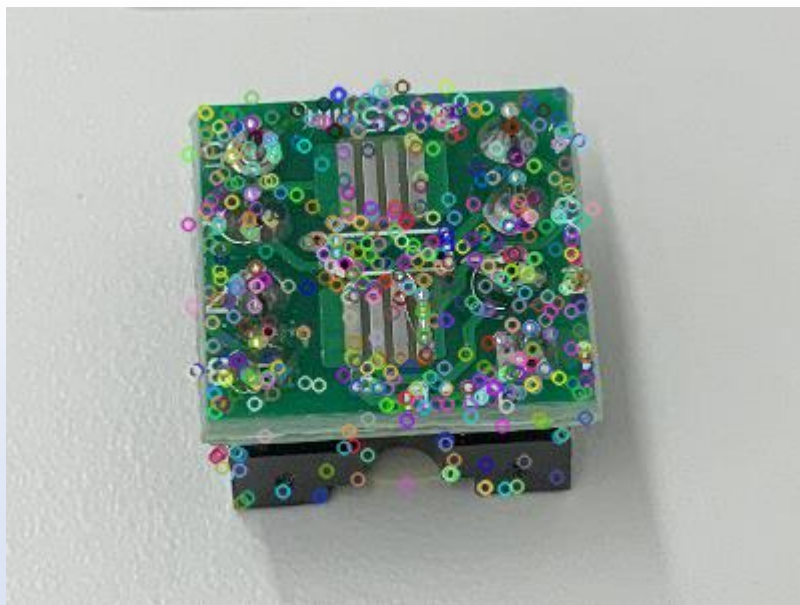


## 12.特徴抽出 (KAZE)



```
img_kaze = copy.deepcopy(img)
kaze = cv2.KAZE_create()

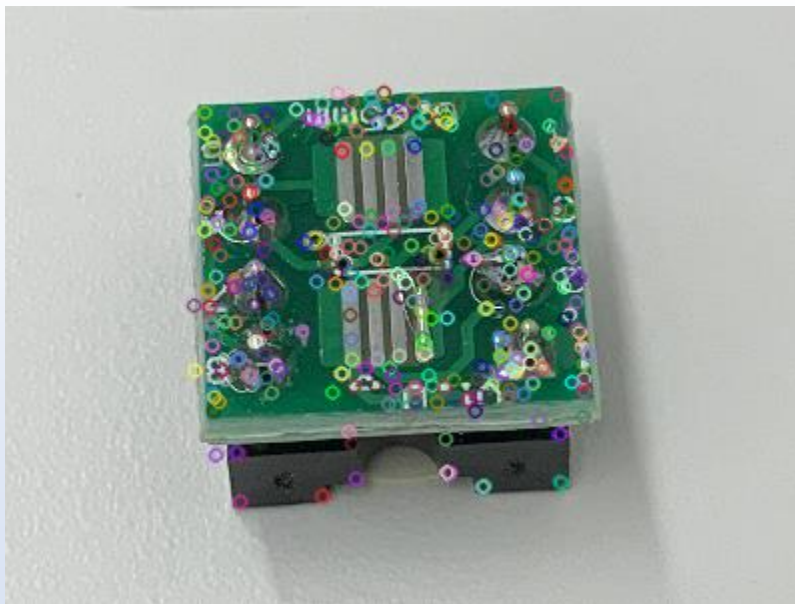
kp1 = kaze.detect(img, None) # 特徴点を抽出
img_kaze = cv2.drawKeypoints(img_kaze, kp1, None)
cv2.imshow('img_kaze')
```



## 12.特徴抽出 (AKAZE)



```
img_kaze = copy.deepcopy(img)
kaze = cv2.AKAZE_create() # akazeに替えるだけ
kp1 = kaze.detect(img, None) # 特徴点を抽出
img_kaze = cv2.drawKeypoints(img_kaze, kp1, None)
cv2.imshow('img_kaze')
```



## 12.特徴抽出 (ORB)



```
img_orb = copy.deepcopy(img)
orb = cv2.ORB_create()
kp2 = orb.detect(img, None) # 特徴点を抽出
img_orb = cv2.drawKeypoints(img_orb, kp2, None)

cv2.imshow(img_orb)
```



点拾いすぎて欠陥を目立たせる方法にはできないか・・・

## 13.色検出

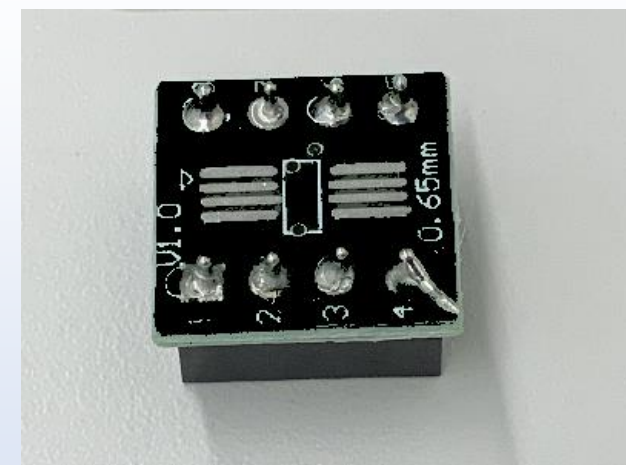
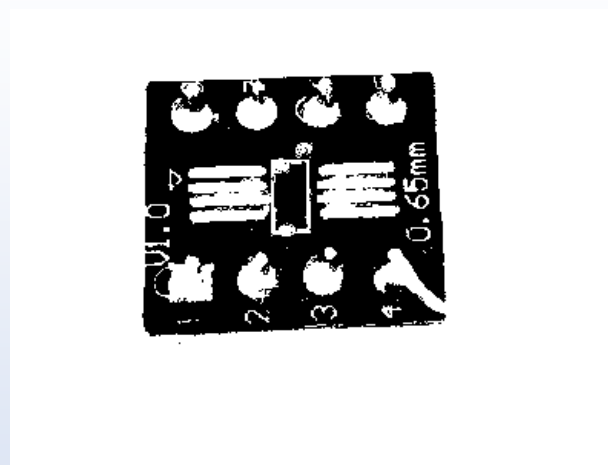
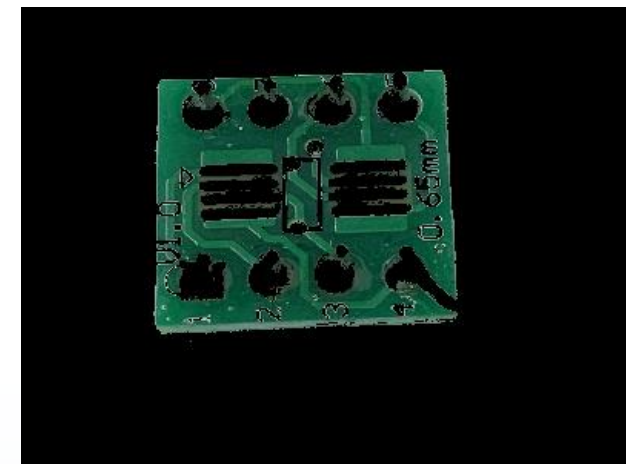
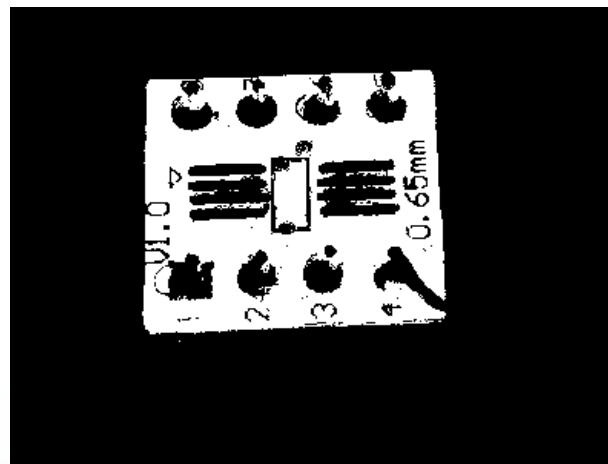
```
import numpy as np

size = (400, 300)
img = cv2.imread("../train/horn/horn_001.jpeg")
img = cv2.resize(img, size)

hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
# 緑色を抜き出す, 下限値、上限値を指定する
# hsv空間は、hは通常のhueの半分の値 (openCV上では)
# 下の例だと、hue20~50が抽出される (svは100/255%)
lower = np.array([30, 50, 50])
upper = np.array([90, 255, 255])
# マスク画像作成(緑色っぽい色の部分が2値化画像で保存される)
img_mask = cv2.inRange(hsv, lower, upper)
# 2値画像の論理積を出して、マスク部分だけを抽出
dst = cv2.bitwise_and(img, img, mask = img_mask)

# 逆マスクも作ってみる (白黒反転)
img_mask_rev = cv2.bitwise_not(img_mask)
dst_rev = cv2.bitwise_and(img, img, mask = img_mask_rev)

cv2.imshow('img_mask')
cv2.imshow('dst')
cv2.imshow('img_mask_rev')
cv2.imshow('dst_rev')
```



指定の範囲の色等を指定して画像を抽出できる

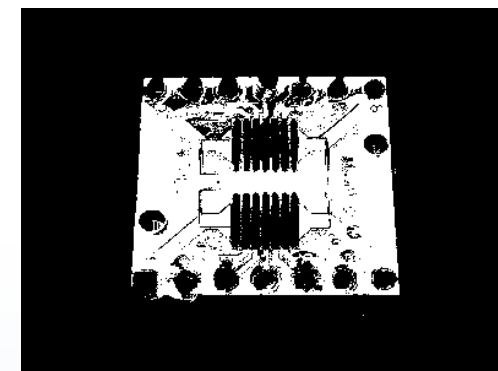
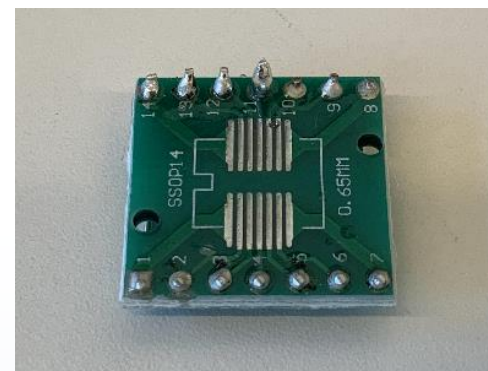


## 14.インペイント

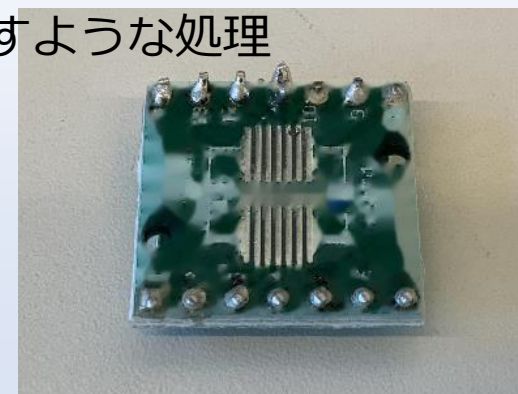
```
▶ img = cv2.imread("./train/horn/horn_009.jpeg")
img = cv2.resize(img, size)

hsv = cv2.cvtColor(img, cv2.COLOR_BGR2HSV)
# 緑色を抜き出す, 下限値、上限値を指定する
# hsv空間は、hは通常のhueの半分の値 (openCV上では)
# 下の例だと、hue20-50が抽出される (svは100/255%)
lower = np.array([30, 50, 50])
upper = np.array([90, 255, 255])
# マスク画像作成(緑色っぽい色の部分が2値化画像で保存される)
img_mask = cv2.inRange(hsv, lower, upper)
|
cv2.imshow(img)
cv2.imshow(img_mask)
```

```
▶ # 修復, 3は取り込む情報量
# うっすらマスク部分が塗りつぶされる
img_dst = cv2.inpaint(img, img_mask, 10, cv2.INPAINT_NS)
cv2.imshow(img_dst)|
```



落書きを消すような処理





## 15.画像切り抜き

[https://water2litter.net/rum/post/python\\_cv2\\_outerrectangle/](https://water2litter.net/rum/post/python_cv2_outerrectangle/)

基板の外形白黒画像の座標から、外形矩形を作成

## 15.画像切り抜き

```
[57] # img_binは輪郭を表す座標の集まり。グレースケールの画像OK
# 以下では外接矩形の座標を示すタプルが返される。(x, y, w, h)
retval = cv2.boundingRect(img_mask)
```

```
[58] retval
```

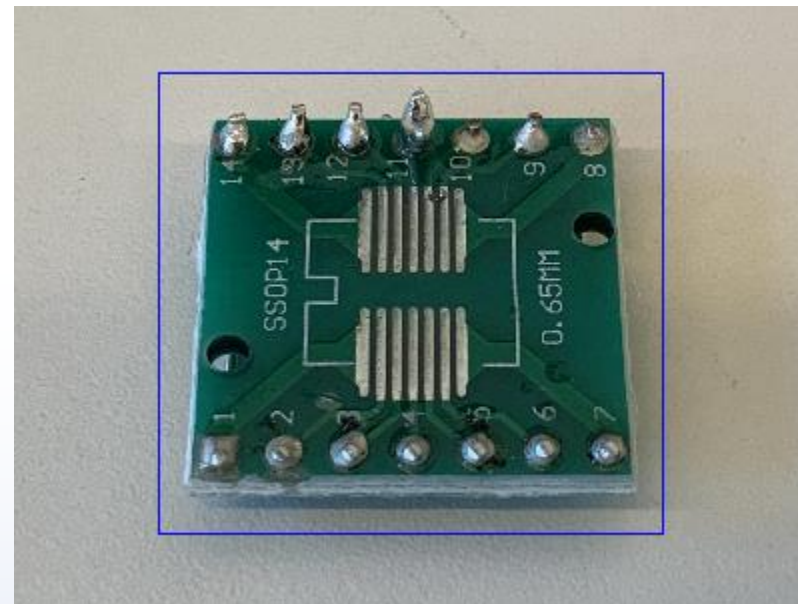
```
(90, 56, 219, 195)
```

```
[ ] # 矩形の座標が計算できたら、その座標の情報を使って矩形を描き
# cv2.rectangle(img, pt1, pt2, color, [thickness], [lineType], [shift])
```

変数	型	内容
img	ndarray	矩形を書き込む画像データ
pt1	tuple	矩形の右上隅の座標。(x, y)
pt2	tuple	矩形の左下隅の座標。(x, y)
color	tuple	矩形の線色。(r, g, b)
thickness	int	省略可。既定値は1。線幅
linetype		省略可。既定値はLINE_8。線種
shift	int	省略可。既定値は0。

```
▶ pt1 = (int((retval[0] + retval[2]) * 1.05), int(retval[1] * 0.6))
pt2 = (int(retval[0] * 0.8), int((retval[1] + retval[3]) * 1.05))
color = (255, 0, 0)
```

```
img_rect = cv2.rectangle(img, pt1, pt2, color)
cv2.imshow(img_rect)
```



上手く囲えているかを確認

## 15.画像切り抜き

```
[84] # 上手く囲えている事を確認。この枠で切り出すには座標をまんま切り取ればよい  
# img[top : bottom , left : right]  
top = int(retval[1] * 0.6)  
bottom = int((retval[1] + retval[3]) * 1.05)  
left = int(retval[0] * 0.8)  
right = int((retval[0] + retval[2]) * 1.05)  
  
img_cut = img[top : bottom , left : right]
```

```
▶ cv2_imshow(img_cut)
```



座標で切り抜けばよい（座標点定数倍だと画像によってはハンダがはみ出す。上手くい感じで自動で切り抜くには・・・？）