

カラーコーン検出アルゴリズムについて

kuma003 (<https://github.com/kuma003>, F.T.E. 14th.)

2025年11月9日

目次

1	はじめに	2
2	画像処理について	2
2.1	色空間	2
2.2	色の演算	4
2.3	モルフォロジー処理	4
3	色相を用いる手法	6

1 はじめに

CanSat では、最終的にゴールへ至るためにゴールコーンを認識する必要がある。そのため、カメラの入力映像からカラーコーンを何らかの手法で検出することが求められる。

そこで本稿ではカラーコーン検出アルゴリズムについて、古典的な色相を用いる場合、逆投影法を用いる場合、そして機械学習を用いる場合の 3 つの手法を紹介し、それぞれの手法の特徴や精度についてまとめる。主にアルゴリズムについて焦点をあて、プログラムの詳細には立ち入らないこともあるため、適宜リファレンスを参照されたい。

テストデータとして用いるデータについては筆者が撮影したものを用いるが、オープンにしづらい画像もあるため、比較的オープンにしやすい画像について、public_dataset フォルダにまとめている。以下に用いるサンプルで用いる画像を示す。



図 1: サンプルで用いる画像。

2 画像処理について

2.1 色空間

カラーコーン検出において、まず重要なのが色の取り扱いである。一般的に、画像は RGB で表現されるのが一般的であるのは周知の事実である。色を RGB で指定することを **RGB 空間**と呼ぶ。

各色の強度を 0 から 255 までの 256 段階 (8 ビット) で表現することが多いが、この場合、 $(0, 0, 0)$ が黒、 $(255, 255, 255)$ が白、 $(255, 0, 0)$ が赤、 $(0, 255, 0)$ が緑、 $(0, 0, 255)$ が青を表す。もしくは、16 進数を用いて `0x000000`、`0x00FF00` のよ

うに表現することも多い。また、さらにアルファチャンネルという透明度を表す成分を加えた RGBA で表現されることもある。この場合、0x00000000 が完全に透明な黒、0xFFFFFFFF が不透明な白を表す。

画像の処理で頻繁に用いられる OpenCV というライブラリにおいても、画像は基本この三原色で表現される。ただし、OpenCV では画像は BGR の順番で表現されるので注意が必要である^{*1}。そのため、例えば OpenCV で処理した画像を matplotlib で表示すると色が変わって見えることがあるが、これは BGR 表記に起因するものであり、そのときは RGB に変換する必要がある。

RGB 表記は色の三原色に基づくものであるが、あまり直観的ではない。画像処理をする上では、色相 (Hue), 彩度 (Saturation), 明度 (Value) の 3 つの成分で色を表現する HSV 空間がよく用いられる。ペイントエディタなどでも直観的に色を選択できるようにな、図 2 のような HSV の色相環が用いられることが多い。

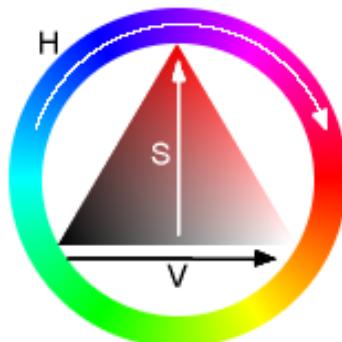


図 2: HSV 色空間のイメージ図。(出典: https://commons.wikimedia.org/wiki/File:Hsv_sample.png)

実用上でも、その場の明るさによって明度が変化することはあるが、色相は比較的变化しづらいため、色相を用いた画像認識が有用である。なお、OpenCV では色相のみ範囲が 0 から 179 までの 180 段階で表現されることに注意が必要である^{*2}。

^{*1} OpenCV が開発された初期のころ、BGR 形式が一般的だったためらしい。

The reason the early developers at OpenCV chose BGR color format is that back then BGR color format was popular among camera manufacturers and software providers. E.g. in Windows, when specifying color value using COLORREF they use the BGR format 0x00bbggrr. (Why does OpenCV use BGR color format ?より)

^{*2} 参考:https://docs.opencv.org/4.x/d9d/tutorial_py_colorspaces.html。ソフトウェアによって異なり、0 から 359 までの範囲で表現されることもある。

2.2 色の演算

ここについては、本番には不要であるが、画像処理の解析を行う上で知っておくと便利な演算について説明する。読み飛ばしても差し支えない。

簡単のために、各ピクセルごとに黒(0)か白(1)の2値で表現される画像を考える。このような画像を**二値画像**と呼び、このような画像を作成することを**二值化**と呼ぶ。このような二値画像に対しては、AND, OR, NOTなどの論理演算を行うことができる。

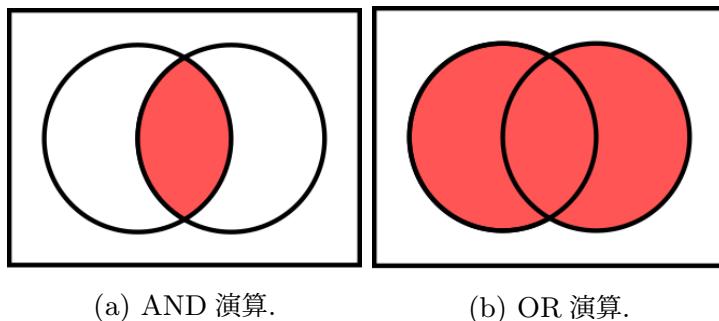


図 3: AND 演算と OR 演算 (出典: <https://commons.wikimedia.org/wiki/File:Venn0001.svg>, <https://commons.wikimedia.org/wiki/File:Venn0111.svg>).

例えば、必要な領域を1、それ以外を0とした二値画像を用意する（このような画像を**マスク画像**と呼ぶ）。このマスク画像と元の画像に対して AND 演算を行うことで、必要な領域だけを抽出することができる。また、マスク画像を複数用意し、OR 演算を行うことで、複数の領域をまとめて抽出することもできる。本番で用いることは少ないが、処理の結果どこを検出したのかを可視化して解析する際には非常に有用である。

2.3 モルフォロジー処理

画像処理において、ノイズ除去や穴埋めなどの目的で**モルフォロジー処理**が用いられることがある。例えば先の述べたようなマスク画像を作成する際、画像自体のノイズや影などの影響で、検出したい領域が途切れ途切れになってしまうことがある。このような場合に、モルフォロジー処理を用いることで、検出したい領域を滑らかにしたり、穴埋めしたりすることができる。

モルフォロジー処理は、膨張 (Dilation) と収縮 (Erosion) の2つの基本的な操作から構成される。膨張は、画像中の白い領域を拡大し、収縮は白い領域を縮小する操作であ

る。膨張によって白い領域に生じた穴が埋まり、収縮によって膨張によって拡大した領域が元の大きさに戻る。これらの操作を組み合わせることで、ノイズ除去や穴埋めなどの効果を得ることができる。ただし、カラーコーン検出においては膨張処理のみを行うことで不確実性を減らしている。

OpenCVでは、`cv2.morphologyEx`を用いると、様々なモルフォロジー処理を簡単に実装できる。以下にその引数についてまとめる。

cv2.morphologyEx の引数 (必須部分のみ)³

```
cv2.morphologyEx(src, op, kernel)
```

1. `src`: 入力画像 (通常は二値画像)
2. `op`: オプション
 - (a) `cv2.MORPH_DILATE`: 膨張処理 (Dilation)
 - (b) `cv2.MORPH_ERODE`: 収縮処理 (Erosion)
 - (c) `cv2.MORPH_OPEN`: 収縮処理 → 膨張処理 (Opening)
 - (d) `cv2.MORPH_CLOSE`: 膨張処理 → 収縮処理 (Closing)
 - (e) `cv2.MORPH_GRADIENT`: 勾配処理 (Gradient)
3. `kernel`: カーネル (後述)

さて、モルフォロジー処理において重要なのがカーネルである。カーネルとは、モルフォロジー処理において用いられる小さな領域であり、画像の各ピクセルに対して適用される。

例えば 5×5 の大きさの正方形のカーネルを考えよう。このカーネルを画像の各ピクセルに対して適用することで、そのピクセルの周囲 5×5 の領域に対して処理が行なわれる。例えば膨張処理の場合は、周囲 5×5 の領域に白いピクセルが1つでもあれば、その中心のピクセルを白にする。逆に収縮処理の場合は、周囲 5×5 の領域が全て白でなければ、その中心のピクセルを黒にする⁴。

カーネルの形状は矩形以外にもとることができ、OpenCVでは矩形の他に十字型や橙円、菱形などを指定することもできる。OpenCVではカーネルは以下に紹介する

³ 参考: https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html.

⁴ 一般にこのようなカーネルを用いた演算は畳み込み演算と呼ばれる。カーネル内での最小値が0であれば収縮、最大値が1であれば膨張となる。また、畳み込み演算では周辺部分でカーネルを用いてどのように畳み込むかが重要となり、省略したが `cv2.morphologyEx` の引数でも指定できる。しかしながら、カラーコーン検出ではそこまで重要でないため省略する。

`cv2.getStructuringElement` で生成できる。

cv2.getStructuringElement の引数

```
cv2.getStructuringElement(shape, ksize)
1. shape: カーネルの形状
   (a) cv2.MORPH_RECT: 矩形
   (b) cv2.MORPH_ELLIPSE: 楕円
   (c) cv2.MORPH_CROSS: 十字型
   (d) cv2.MORPH_DIAMOND: 菱形
2. ksize: カーネルの大きさ (タプルで指定. e.g., (5, 5))
```

3 色相を用いる手法

色相を用いる手法は、カラーコーンの色相の範囲をあらかじめ決めておき、その範囲内にあるピクセルをカラーコーンとして検出することで実現される。非常に実装が容易であり、計算の負荷がとても低いため、堅実に動作する手法である。

ソースコードは以下の listing 1 のようになる。

Listing 1: 色相を用いる手法のサンプルコード (codes/hue.py)

```
1 import cv2
2 import numpy as np
3 import graph
4
5 if __name__ == "__main__":
6     # Load an image from file
7     image = cv2.imread("../public_dataset/testdata.png")
8
9     # Convert the image from BGR to HSV color space
10    hsv_image = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
11
12    # Define the range for a specific hue (e.g., red color)
13    lower_hue = 150
14    upper_hue = 180
15
```

```

16     # Create a mask using the defined hue range
17     mask = cv2.inRange(hsv_image, (lower_hue, 100, 0), (upper_hue,
18                               255, 255))
19
20     # Apply the mask to the original image
21     result = cv2.bitwise_and(image, image, mask=mask)
22
23     # Display the original and resulting images
24     graph.plot_result(
25         image, mask, result, title1="Original", title2="Mask", title3
26             ="Result"
27     )

```

コードの概略は非常にシンプルであり、特定の色相（ここでは 150 から 180）の範囲を定義し、その範囲にあるピクセルをマスクとして抽出している。また、空などを誤検知するのを防ぐために、彩度についても制限を加えている。

画像処理による結果を以下の図 4 に示す。

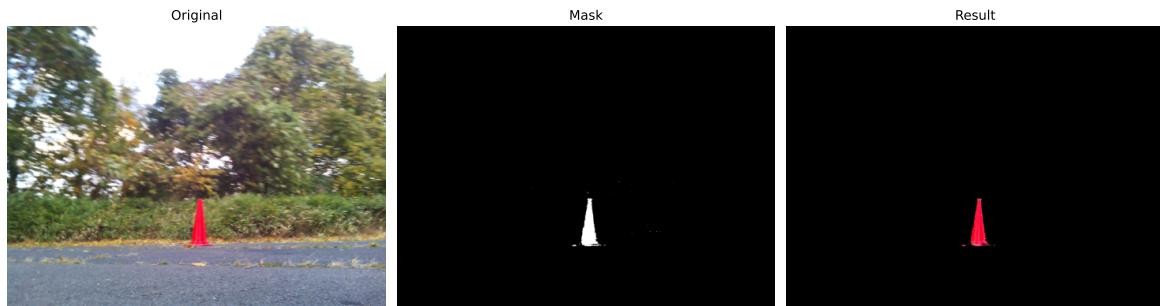


図 4: 色相を用いる手法の結果。左から元画像、マスク画像、検出結果。