```
import warnings

import cv2
import numpy as np

import myfunc  #自作モジュール


warnings.filterwarnings("error")


''' 初期変数設定 '''

THRESHOLD = 25  # 二値化の際の閾値（大きいほど検出される粒子少）
PX = 115  # ミクロメーター1目盛り(10μm)当たりのピクセル数
x_list = []
y_list = []
ptlMileageSum  = 0
ptlCenterCoords = []  # 矩形領域(粒子)の中心座標リスト

VIDEO_PATH = "./video.mp4"  # 動画パス
''' '''


''' 動画の前処理～粒子検出 '''

# cv2で動画を読込み
video = cv2.VideoCapture(VIDEO_PATH)
isFrameLoaded, frame = video.read()
frame = cv2.resize(frame, (1200, 674))
previousFrame = frame.copy()

# 画像の前処理
binalizedFrame = myfunc.preprocess(frame, True, THRESHOLD)

# 外枠のみ輪郭検出
# 戻り値: contours = 輪郭座標, hierarchy = 輪郭の階層情報
detectedContours_f, hierarchy = cv2.findContours(binalizedFrame,
                                                 cv2.RETR_EXTERNAL,
                                                 cv2.CHAIN_APPROX_SIMPLE)

for i in range(len(detectedContours_f)):
    rectAreasCoords = cv2.boundingRect(detectedContours_f[i]) # 輪郭座標から矩形領域取得
    x, y = myfunc.calc_center(rectAreasCoords)          # 矩形領域の中心座標計算
    ptlCenterCoords.append([x, y])
    # 矩形領域の左上に番号印字
    cv2.putText(img = frame, text = f"{i}",
                org = (rectAreasCoords[0], rectAreasCoords[1]),
                fontFace = cv2.FONT_HERSHEY_PLAIN, fontScale = 0.7,
                color = (255,0,0), thickness = 1, lineType = cv2.LINE_AA)
    cv2.drawMarker(frame, position = (x, y),
                   color = (0,255,0), markerSize = 5) # 中心座標にマーク

''' '''


print("> キー入力を待機しています...")
```

```
while True:
    cv2.imshow('Particles Map', frame)
    cv2.waitKey(-1)
    selectedPtlNum = int(input("> 追跡する粒子の番号を入力してください: "))

    if selectedPtlNum > len(detectedContours_f) - 1:
        print(f"> 0～{len(detectedContours_f) - 1}の範囲で指定してください")

    else:
        break

# 選択した粒子の座標の代入
x_list.append(ptlCenterCoords[selectedPtlNum][0])
y_list.append(ptlCenterCoords[selectedPtlNum][1])


''' 稼働開始 '''
flameNum = 0
isNotPushed = True
isLessthanfive = True
count = 1

trackingTime = int(input("> 追跡する秒数を入力してください: "))
fps = int(video.get(cv2.CAP_PROP_FPS))
endFlameNum = fps * trackingTime # 追跡終了時のフレーム番号

# 粒子の位置情報を格納するためのリスト
avg_location = np.zeros((4), np.int32)
location1 = np.zeros((4), np.int32)
location2 = location1.copy()
location3 = location1.copy()
location4 = location1.copy()
location5 = location1.copy()

while video.isOpened():
    isFrameLoaded, frame = video.read() # フレームを読込み
    flameNum += 1

    ' フレームが読み込まれていない (動画再生終了時の処理) '
    if not isFrameLoaded:
        print("動画再生が終了しました。")
        print(f"現在の経過時間{myfunc.decimalRound(sum = flameNum / fps, sig =
len(str(flameNum)))}[秒]")
        print(f"粒子番号: {selectedPtlNum}")
        print(f"閾値: {THRESHOLD}")
        myfunc.measure(x_list = x_list, y_list = y_list,
                       ptlMillageSum = ptlMileageSum, px = PX) # 距離出力
        break

    ' 前処理～粒子全て検出 '
    frame = cv2.resize(frame, (1200, 674))    # リサイズ
    previousFrame = frame.copy()              # 画像出力時のため複製する
    binalizedFrame = myfunc.preprocess(frame, False,
                                       THRESHOLD, x_list, y_list) # 前処理
    cv2.imshow("binaly", binalizedFrame)

    # 輪郭検出
```

```
    detectedContours, hierarchy = cv2.findContours(binalizedFrame,
cv2.RETR_EXTERNAL,
                                                   cv2.CHAIN_APPROX_SIMPLE) # 外
枠のみ検出
    # 円検出
    # dp = 解像度(1程度), minDist = 円同士の最小距離, param1 = cannyの閾値,
    # param2 = 低→誤検出多, minRadius = 最小半径, maxRadius = 最大半径
    detectedCircles = cv2.HoughCircles(binalizedFrame, cv2.HOUGH_GRADIENT,
                                       dp = 1.2, minDist = 5, param1 = 100,
                                       param2 = 8, minRadius = 2, maxRadius = 9)

    # 候補関係のリスト初期化
    candidates = []    # 粒子の候補
    ptlMileages = []   # 移動距離
    ptlAreas = []      # 粒子面積

    ' 検出した座標の処理 '
    # 輪郭座標(contours)が得られたとき
    if detectedContours is not None:

        for i in detectedContours:
            # 矩形領域の取得
            # 戻り値: 領域左上x, y, width, height
            rectAreasCoords = cv2.boundingRect(i)

            x, y = myfunc.calc_center(rectAreasCoords)       # 粒子の中心を求め
る
            currentCoord = np.array([x, y])                  # 現在の粒子の座標
            lastCoord = np.array([x_list[-1], y_list[-1]])   # 前回の粒子の座標

            currentMileage = myfunc.decimalRound(
                np.linalg.norm(currentCoord - lastCoord), 4)  # 前回との座標差分
より距離算出

            # frameをトリムして粒子を抜き出し (top:bottom, left:right)
            trimedPtlFrame = binalizedFrame[y - 8 : y + 8, x - 8 : x + 8]

            currentArea = myfunc.calc_area(trimedPtlFrame)  # 粒子面積を算出

            ptlAreas.append(currentArea)
            candidates.append([x, y, 0, 0, 0, 0]) # [粒子の中心x, y, 上, 下, 右,
左]
            ptlMileages.append(currentMileage)

    # 円(circle)が得られたとき
    if detectedCircles is not None:
        detectedCircles = np.uint16(np.around(detectedCircles)) # 円座標データの
処理

        # 検出した円全てに対して処理する
        for i in detectedCircles[0, :]:
            x = int(i[0]) # 円の中心x座標
            y = int(i[1]) # 円の中心y座標

            currentCoord = np.array([x, y])
            lastCoord = np.array([x_list[-1], y_list[-1]])

            currentMileage = myfunc.decimalRound(
```

```
                np.linalg.norm(currentCoord - lastCoord), 4)

            trimedPtlFrame = binalizedFrame[y - 8 : y + 8, x - 8 : x + 8]

            currentArea = myfunc.calc_area(trimedPtlFrame)

            isDuplicate = False

            # 粒子の重複判定 (contoursとの重複を確認。重複あれば候補に入れない)
            for j in range(len(candidates)):
                distBetweenPlts = myfunc.decimalRound(
                    np.linalg.norm(currentCoord - candidates[j][:2]), 4) # 粒子
同士の距離算出

                # 粒子同士の距離が2以下 or 面積の一致 で重複と見做す
                if float(distBetweenPlts) <= 2 or currentArea == ptlAreas[j]:
                    isDuplicate = True
                    break

            # 重複しない場合追加
            if isDuplicate == False:
                candidates.append([x, y, 0, 0, 0, 0])
                ptlMileages.append(currentMileage)
                ptlAreas.append(currentArea)

    # 候補がない場合、最新の座標を流用して終了
    if len(candidates) == 0:
        track_x = x_list[-1]
        track_y = y_list[-1]
        last_location = np.zeros((4), np.int32)

    else:

        ' 追跡する座標の決定 '
        if len(candidates) >= 2:

            # 最小距離と最大距離の差が12より大きいなら候補を削除
            if max(ptlMileages) - min(ptlMileages) > 12:
                del candidates[ptlMileages.index(max(ptlMileages))]
                del ptlMileages[ptlMileages.index(max(ptlMileages))]

        gaps = []

        # 候補(粒子)の数だけ実行
        for i in range(len(candidates)):

            for j in range(i + 1, len(candidates)):
                #candidates[i]の座標を原点に平行移動
                pos_x = candidates[j][0] - candidates[i][0]
                pos_y = candidates[j][1] - candidates[i][1]

                #周り4方位に粒子がいくつあるかを格納していく
                #上
                if pos_y <= 0:
                    candidates[i][2] += 1
                    candidates[j][3] += 1

                #下
```

```
                else:
                    candidates[i][3] += 1
                    candidates[j][2] += 1

                #右
                if pos_x >= 0:
                    candidates[i][4] += 1
                    candidates[j][5] += 1

                #左
                else:
                    candidates[i][5] += 1
                    candidates[j][4] += 1

            current_location = np.array(candidates[i][2:])

            try:
                #相関係数計算
                coef = np.corrcoef(current_location, avg_location)

            except RuntimeWarning:
                correlation = 0

            else:
                correlation = round(coef[0, 1], 2)

                #相関係数がNaNの場合
                if np.isnan(correlation):
                    correlation = 0

            gap = myfunc.decimalRound(
                ptlMileages[i] * 0.7 + (1 - correlation) * 4, 4)
            gaps.append(gap)

        track_num = gaps.index(min(gaps))  # gapが最小になる座標を追跡する
        track_x = candidates[track_num][0]
        track_y = candidates[track_num][1]
        last_location = np.array(candidates[track_num][2:])

    ' 現在の粒子の座標と過去の座標を使用した処理 '
    # 移動距離をptlMileageSumに加算
    ptlMileageSquare = ((x_list[-1] - track_x) ** 2
                        + (y_list[-1] - track_y) ** 2)
    ptlMileageSum = (ptlMileageSum
                     + myfunc.decimalRound(ptlMileageSquare ** 0.5, 5))

    # 粒子の現在地をリストに追加
    x_list.append(track_x)
    y_list.append(track_y)

    # 粒子の位置にマークする
    cv2.drawMarker(frame, (track_x, track_y),(255, 0, 255), markerSize = 5)

    # 軌跡の点を直線で結ぶ
    for i in range(1, len(x_list)):
        cv2.line(frame, (x_list[i-1], y_list[i-1]), (x_list[i], y_list[i]),
(255, 0, 0))
```

```
    #location1~5のcount番目を更新
    exec(f"location{count} = last_location")

    avg_location = np.zeros((4), np.float32)

    if isLessthanfive: #繰り返し回数が5回以下の場合
        divisor = count #割る数をcountにする

    else:
        divisor = 5

    for i in range(1, divisor + 1):
        exec(f"avg_location += location{i}") #位置情報を足していく

    avg_location = np.round(avg_location / divisor, 2) #過去5回分の位置情報の平
均をとる

    count += 1 #1~5まで数える

    if count == 6:

        if isLessthanfive:
            isLessthanfive = False

        count = 1

    cv2.imshow('Tracking', frame)

    ' システム '
    # Escキーの押下で処理を中止
    key = cv2.waitKey(1) & 0xFF
    if key == 27:
        print(f"現在の経過時間{myfunc.decimalRound(num = flameNum / fps, sig =
len(str(flameNum)))}[秒]")
        print(f"粒子番号: {selectedPtlNum}")
        print(f"閾値: {THRESHOLD}")
        myfunc.measure(x_list = x_list, y_list = y_list,
                       ptlMileageSum = ptlMileageSum, px = PX) # 移動距離計算
        break

    if flameNum == endFlameNum:
        print(f"{trackingTime}秒間追跡が終了しました")
        print(f"粒子番号: {selectedPtlNum}")
        print(f"閾値: {THRESHOLD}")
        myfunc.measure(x_list = x_list, y_list = y_list,
                       ptlMillageSum = ptlMileageSum, px = PX) # 移動距離計算
        break

    # 押されるまで待機
    if isNotPushed:
        print("> キー入力を待機しています...")
        cv2.waitKey(-1)
        isNotPushed = False
''' '''



#動画の解放とウィンドウ終了
video.release()
```

```
cv2.destroyAllWindows()
print("> 操作を終了しました。")
```