```
from math import log10, floor
from decimal import Decimal, ROUND_HALF_UP

import cv2
import numpy as np


#背景差分モデル設定
bgsegmModel = cv2.bgsegm.createBackgroundSubtractorMOG()


def calc_dig(num):
    '''
    数字の桁数を計算する。

    Parameters
    ----------
    num : int or float
        桁数を算出する数値

    Returns
    -------
    digits : int
    '''    算出した桁数


    if num == 0:
        digits = 1
    else:
        digits = int(floor(log10(abs(num)))) + 1  #桁数算出
    return digits


def decimalRound(num, sig):
    '''
    decimalを用いて四捨五入する。

    Parameters
    ----------
    num : int or float
        四捨五入する数値

    Returns
    -------
    roundedNum : int or float
    '''    四捨五入された数値


    num = Decimal(str(num))
    dig = "1E" + str(calc_dig(num) - sig)  # 四捨五入する位を求める(1E1なら1の位
で四捨五入)
    roundedNum = num.quantize(Decimal(dig), rounding = ROUND_HALF_UP)

    if "E" in str(roundedNum):
        return int(roundedNum)  # 整数の場合にEで表示されることを回避

    else:
        return float(roundedNum)
```

```
def calc_center(coodinates):
    '''
    粒子の中心座標(x, y)を求める。
    引数は整数値にする必要あり。

    Parameters
    ----------
    coordinates : list of int
        それぞれx左端(0), y上端(1), width(2), height(3)の値を格納したリスト。

    Returns
    -------
    x_center : int
        粒子中心のx座標
    y_center : int
    ,,,    粒子中心のy座標
    '''


    x_center = coodinates[0] + (coodinates[2] - 1) / 2
    x_center = int(decimalRound(x_center, calc_dig(x_center)))
    y_center = coodinates[1] + (coodinates[3] - 1) / 2
    y_center = int(decimalRound(y_center, calc_dig(y_center)))
    return x_center, y_center


# 移動距離の測定
def measure(x_list, y_list, ptlMileageSum, px):
    '''
    移動距離の測定

    Parameters
    ----------
    x_list : list of int
        各フレームにおける粒子位置のx座標
    y_list : list of int
        各フレームにおける粒子位置のy座標
    ptlMillageSum : int or float
        粒子の総移動距離(単位: ピクセル)
    px : int
    ,,,    ミクロメーター1目盛り(10μm)あたりのピクセル数
    '''


    # 1ピクセルが何μmに当たるか定義
    scale_per_pxl = decimalRound(10 / px, len(str(px)))

    x = max(x_list) - min(x_list)
    y = max(y_list) - min(y_list)
    width = decimalRound(x * scale_per_pxl, min(calc_dig(x), calc_dig(px)))
    height = decimalRound(y * scale_per_pxl, min(calc_dig(y), calc_dig(px)))
    ptlMileageSum = decimalRound(ptlMileageSum * scale_per_pxl,
                                 min(calc_dig(ptlMileageSum), calc_dig(px)))

    print(f"移動距離: {ptlMileageSum}")
    print(f"横幅: {width}")
    print(f"縦幅: {height}")
```

```
def preprocess(src, first, threshold, *plist):
    '''
    前処理

    Parameters
    ----------
    src : numpy.ndarray
        前処理を施すフレーム
    first : boolean
        追跡する粒子を選ぶ段階ならTrue
        追跡中はFalse
    *plist : list of int
        x_listとy_lsitが渡される想定

    Returns
    -------
    frame_return : numpy.ndarray
        前処理が施されたフレーム
    '''


    ' Step 1 (グレースケール変換)'
    src = cv2.cvtColor(src, cv2.COLOR_BGR2GRAY)

    ' Step 2 (コントラスト強調)'
    # 引数: clipLimit = 大きくするほどコントラスト強く荒く,
    #       tileGridSize = 大きいと大局的、小さいと部分的に平坦化
    clahe = cv2.createCLAHE(clipLimit = 4, tileGridSize = (50, 50)) # cliplimit
は3~4
    frame_clahe = clahe.apply(src)

    ' Step 3 (ノイズを除去して二値化画像作成)'
    # clahe済みframeから画像のノイズ除去
    # 引数: frame, ksize=カーネルサイズ(奇数) カーネル(ぼかし処理の領域単位・大
きいほど強い)
    frame_blur1 = cv2.medianBlur(frame_clahe, 7)

    # 適応的な閾値で画像の二値化 (白黒変換)
    # 引数: frame, maxValue = 二値化後の輝度値, adaptiveMethod = 適応的閾値処理
のアルゴリズム
    #       thresholdType = 二値化の種類, blockSizr = 局所領域の大きさ(奇数), C
= 閾値から引く数
    frame_thresh = cv2.adaptiveThreshold(
        frame_blur1, 255, cv2.ADAPTIVE_THRESH_GAUSSIAN_C,
        cv2.THRESH_BINARY_INV, blockSize = 55, C = threshold)

    ' Step 4 (二値化画像から更にノイズ除去)'
    frame_thresh = cv2.medianBlur(frame_thresh, 7)

    ' Step 4.5 (背景差分実装、粒子にマスク合成する)'
    if first == False:
        frame_bgsegm = bgsegmModel.apply(frame_thresh)  #背景差分実装

        # グレースケールの黒画像 (674x1200のndarrayオブジェクトを生成)
        black = np.zeros((674, 1200), dtype = np.uint8)
        # 現在の粒子の座標を中心に、半径16の白い円を塗りつぶし描画
        cv2.circle(img = black, center = (plist[0][-1], plist[1][-1]), radius =
16,
```

```
                    color = (255, 255, 255), thickness = -1)
        frame_bgsegm = cv2.bitwise_and(frame_bgsegm, black)  # マスク合成

        #モルフォロジー演算（クロージング、穴埋め）
        kernel = cv2.getStructuringElement(cv2.MORPH_ELLIPSE, (5, 5))
        frame_closed = cv2.morphologyEx(frame_bgsegm, cv2.MORPH_CLOSE,
                                        kernel, iterations = 2)

        frame_return = frame_closed

    else:
        frame_return = frame_thresh

    return frame_return


def calc_area(src):
    '''
    粒子面積の算出。

    Parameters
    ----------
    src : numpy.ndarray
        粒子のみをトリミングしたフレーム

    Returns
    -------
    area_ptl : float
        粒子の面積
    '''

    contours_ptl, hierarchy = cv2.findContours(src, cv2.RETR_EXTERNAL,
                                               cv2.CHAIN_APPROX_SIMPLE) # 外枠の
み輪郭検出
    areas = []

    # 輪郭が複数検出されることを考慮
    for i in contours_ptl:
        area = cv2.contourArea(i) # 輪郭の面積計算
        areas.append(area)

    # 最大のものを粒子の面積とする
    if areas == []:
        area_ptl = 0
    else:
        area_ptl = max(areas)

    return area_ptl
```