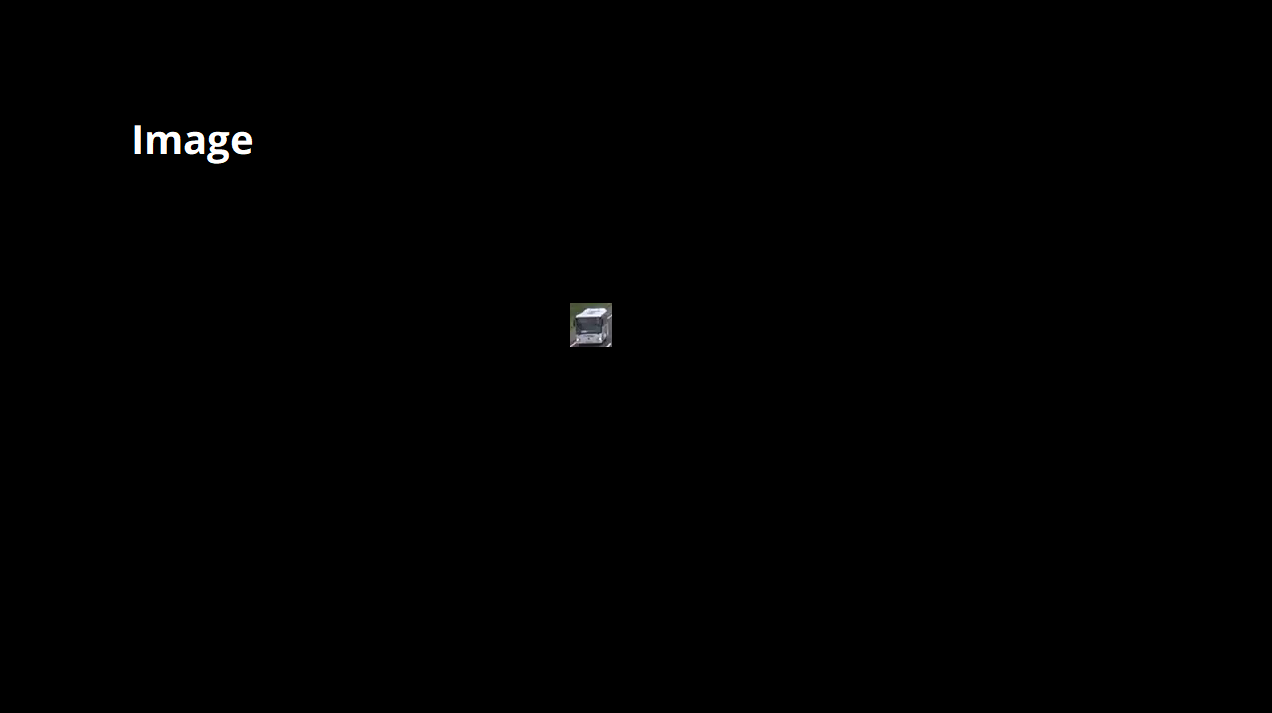
## 💬 **تابع do\_bgremove**

این تابع تصویر image که کراپ شده است را به عنوان ورودی می‌گیرد و بک گراند را از آن حذف می‌کند



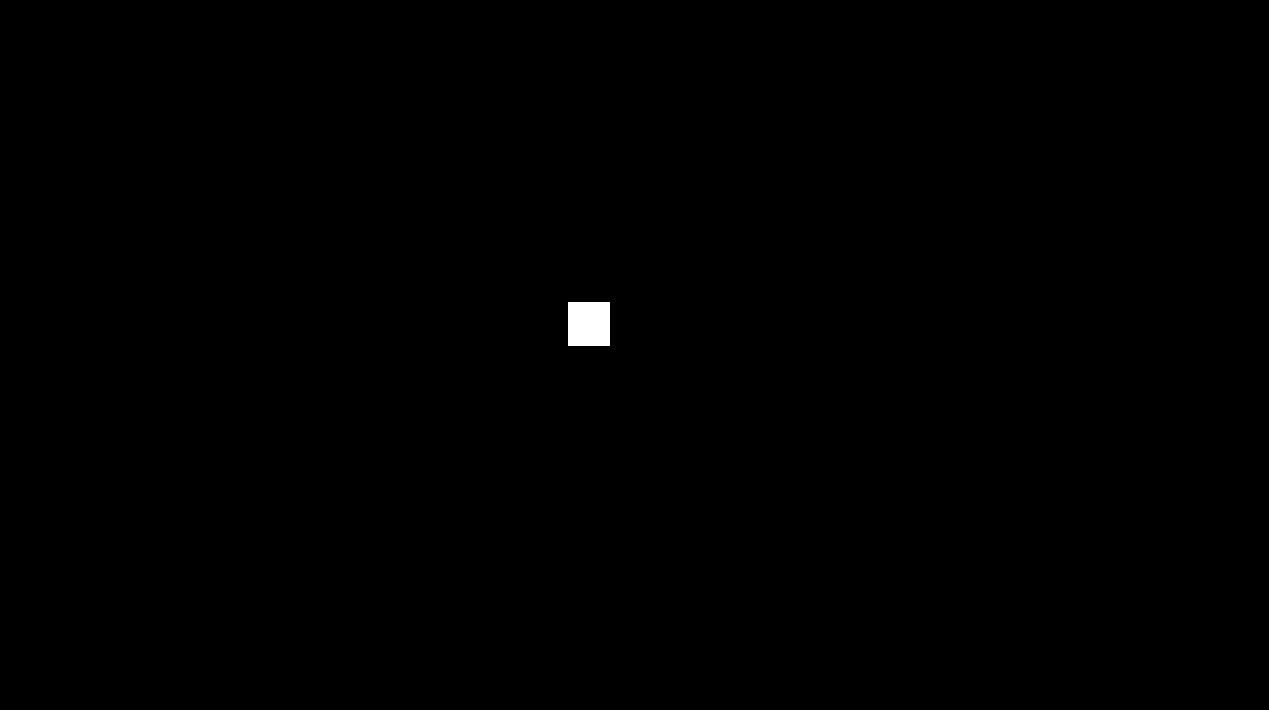
| background = cv2.imread('background.jpg') |
| --- |

این دستور فایل 'background را که قبلا استخراج شده است لود می‌کند.



| crop\_grayscale = cv2.cvtColor(  image, cv2.COLOR\_BGR2GRAY) # converting to grayscale  \_, crop\_mask\_grayscale = cv2.threshold(  crop\_grayscale, 1, 255, cv2.THRESH\_BINARY) # masking cropped image |
| --- |

این قسمت image را به grayscale تبدیل کرده و همچنین یک threshold روی آن می‌زند (مقدار ترشلد باید خیلی کم باشد که در اینجا ۱ قرار دادیم) تا یک ماسک از ناحیه‌ی کراپ شده به دست بیاید.



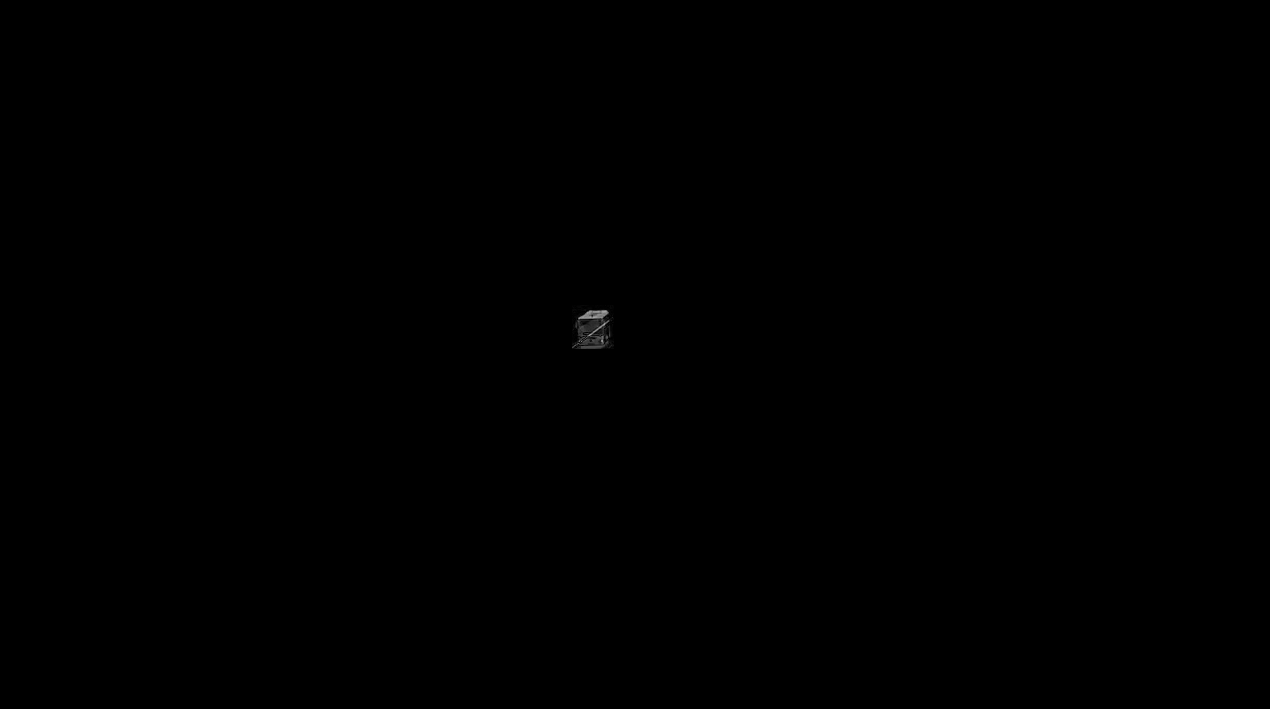
| image\_dist = do\_euclid\_dist(image, background) |
| --- |

این تابع فاصله‌ی اقلیدسی image از background را محاسبه می‌کند. خروجی آن به این شکل است



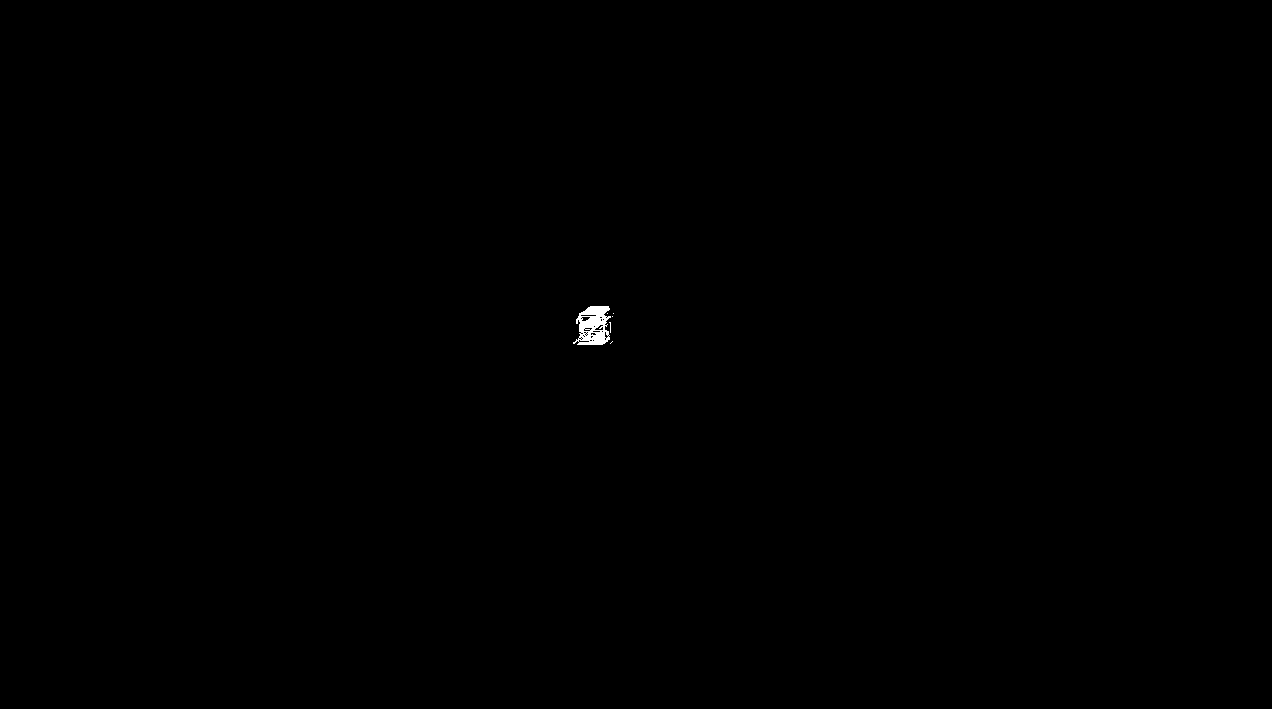
| image\_dist = cv2.bitwise\_and(  image\_dist, image\_dist, mask=crop\_mask\_grayscale)  image\_dist = image\_dist.astype('uint8') |
| --- |

سپس این خروجی با crop\_mask\_grayscale که پیش از این به دست آوردیم AND شده و به 'uint8' تبدیل خواهد شد تا قسمت‌های اضافی آن حذف شود

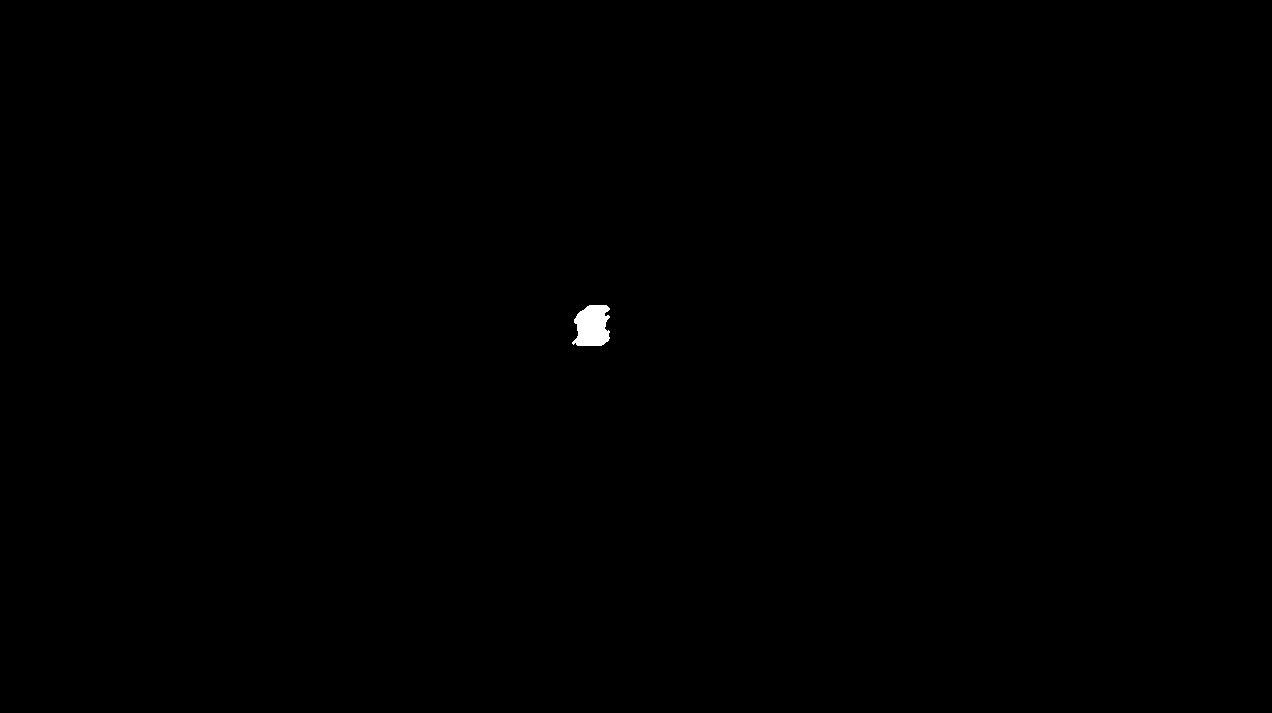


| \_, image\_mask = cv2.threshold(  image\_dist, 25, 255, cv2.THRESH\_BINARY) |
| --- |

حالا زوی این image\_dist یک ترشلد انجام می شود

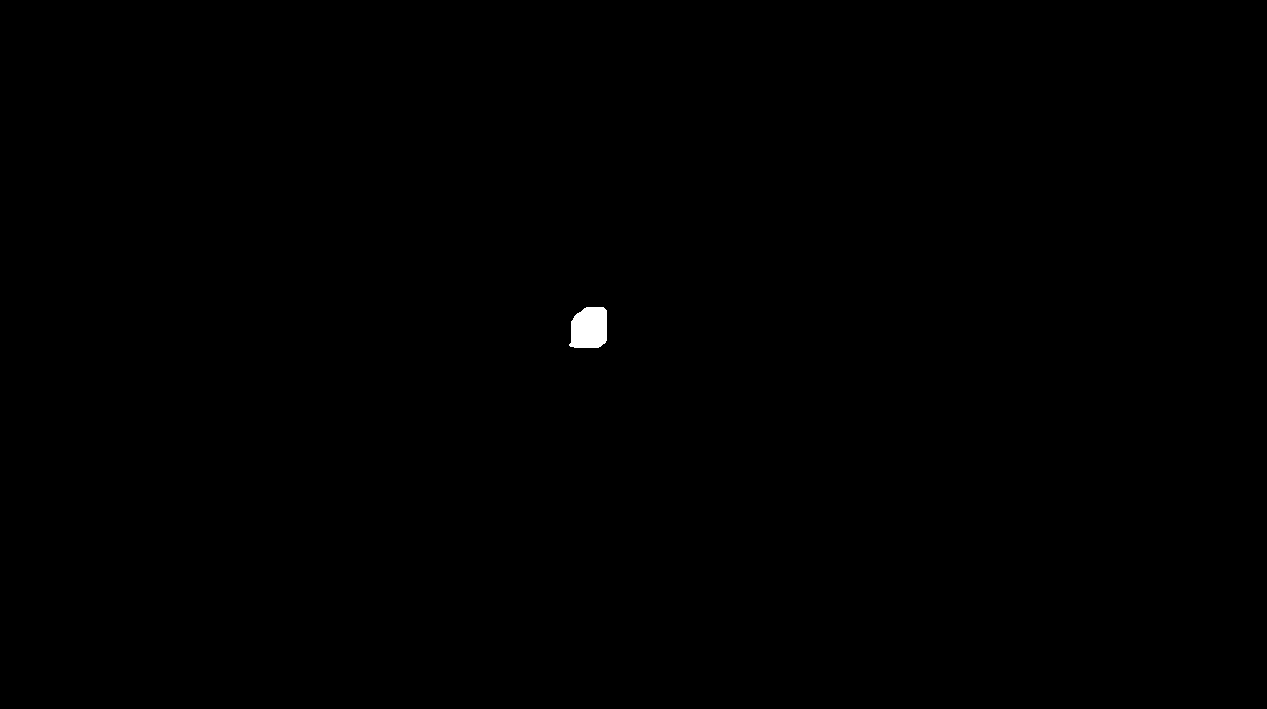


برای اینکه خروجی بهتر و یک تکه باشد این خروجی به تابع do\_findContours داده می‌شود که بزرگترین کانتور تصویر را استخراج می‌کند



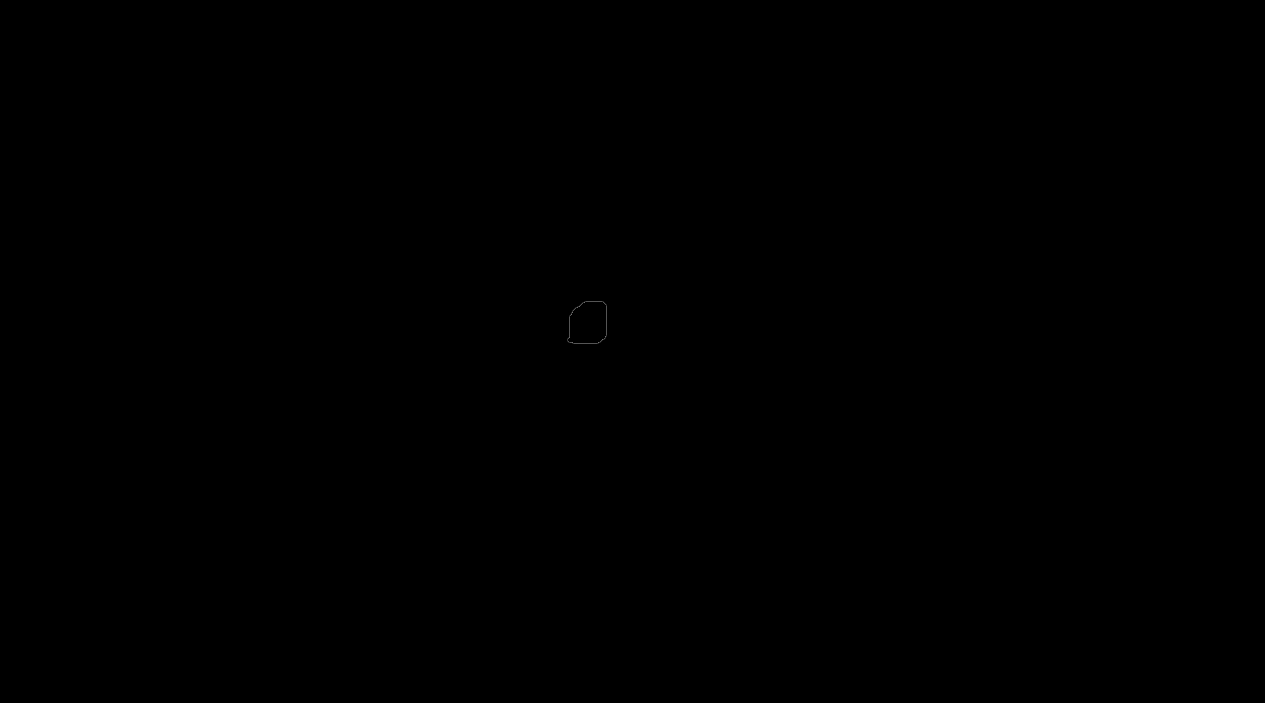
همچنین برای بهبود خروجی و پر کردن حفره‌های احتمالی از توابع مورفولوژی closing استفاده می‌کنیم

| kernel\_closing = np.ones((100, 100), np.uint8)  image\_mask\_closing = cv2.morphologyEx(  image\_mask, cv2.MORPH\_CLOSE, kernel\_closing) |
| --- |

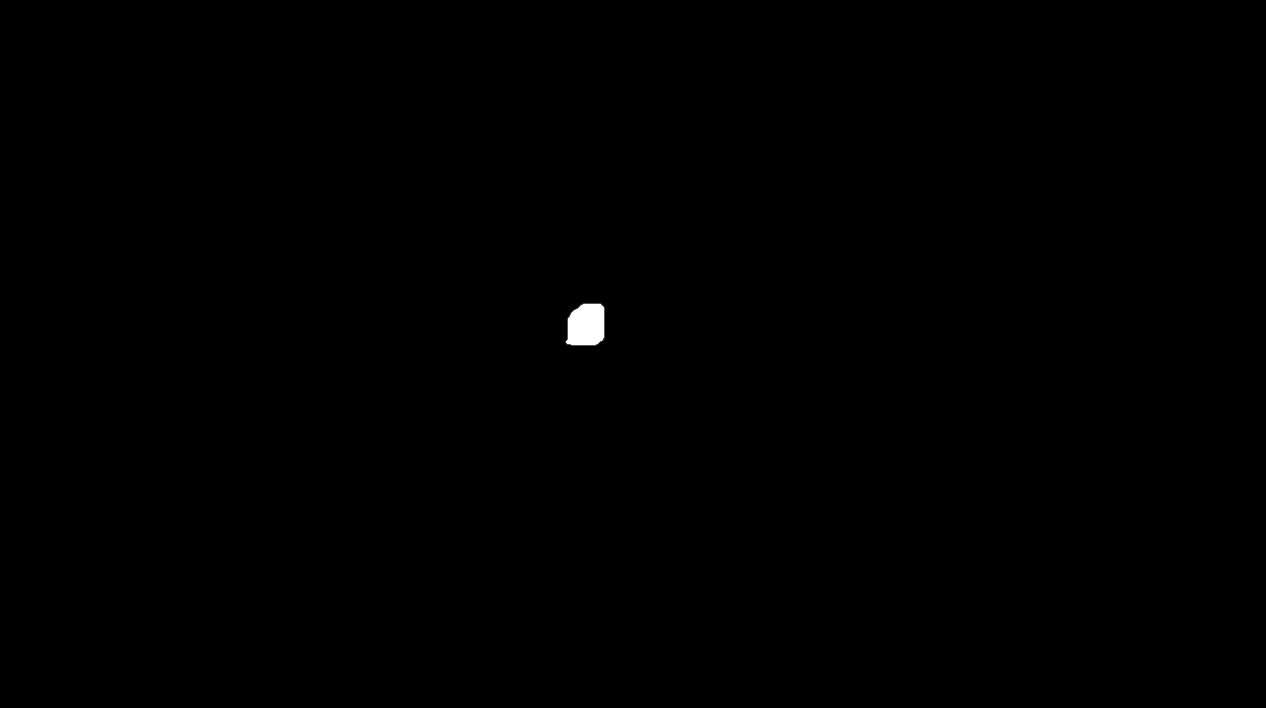


سپس برای اینکه خروجی هموار‌تر شود لبه‌های آن را محو می کنیم. در اینجا از یک blur با کرنل ۳ در ۳ استفاده است که هر قدر این کرنل بزرگ تر شود لبه‌ها smooth تر خواهد شد.

| image\_mask\_closing\_blur = cv2.blur(image\_mask\_closing, (3, 3))  image\_mask\_stroke = image\_mask\_closing\_blur.copy()  image\_mask\_stroke[image\_mask\_closing != 0] = 0  final\_mask = cv2.add(image\_mask\_stroke, image\_mask\_closing) |
| --- |

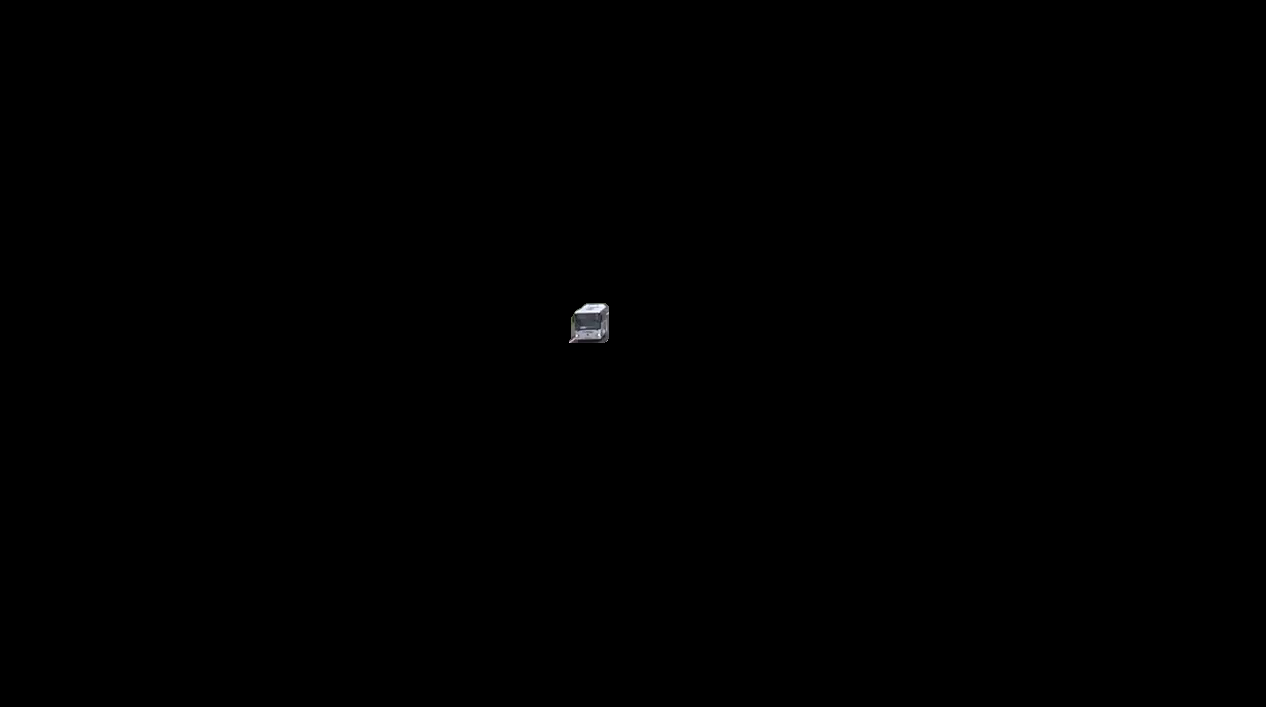


بنابراین در نهایت با جمع کردن این لبه‌های محو شده و ماسک اولیه این ماسک نهایی به دست می‌آید.



در آخر این ماسک را با تصویر کراپ شده اولیه And می‌کنیم

| masked\_image = cv2.bitwise\_and(  image, image, mask=final\_mask) |
| --- |



## 💬 **تابع do\_euclid\_dist**

ابن تابع که درون تابع do\_bgremove فراخوانی می‌شود یک تصویر background و یک تصویر image را دریافت می‌کند و فاصله‌ی اقلیدسی این دو از هم را محاسبه می‌کند.

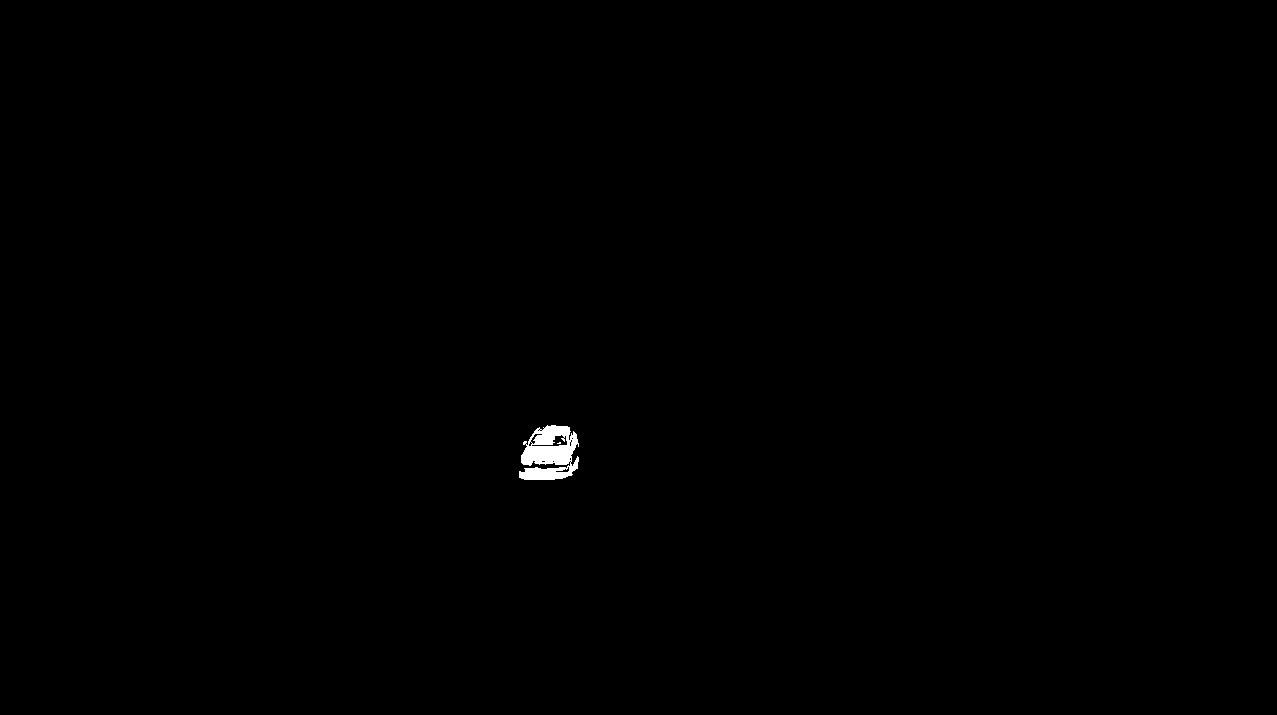
| background = cv2.cvtColor(background, cv2.COLOR\_BGR2GRAY)  image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY) # converting to float64  image\_dist = cv2.absdiff(background, image) # algebraic dist  return image\_dist |
| --- |

## 💬 **تابع do\_findContours**

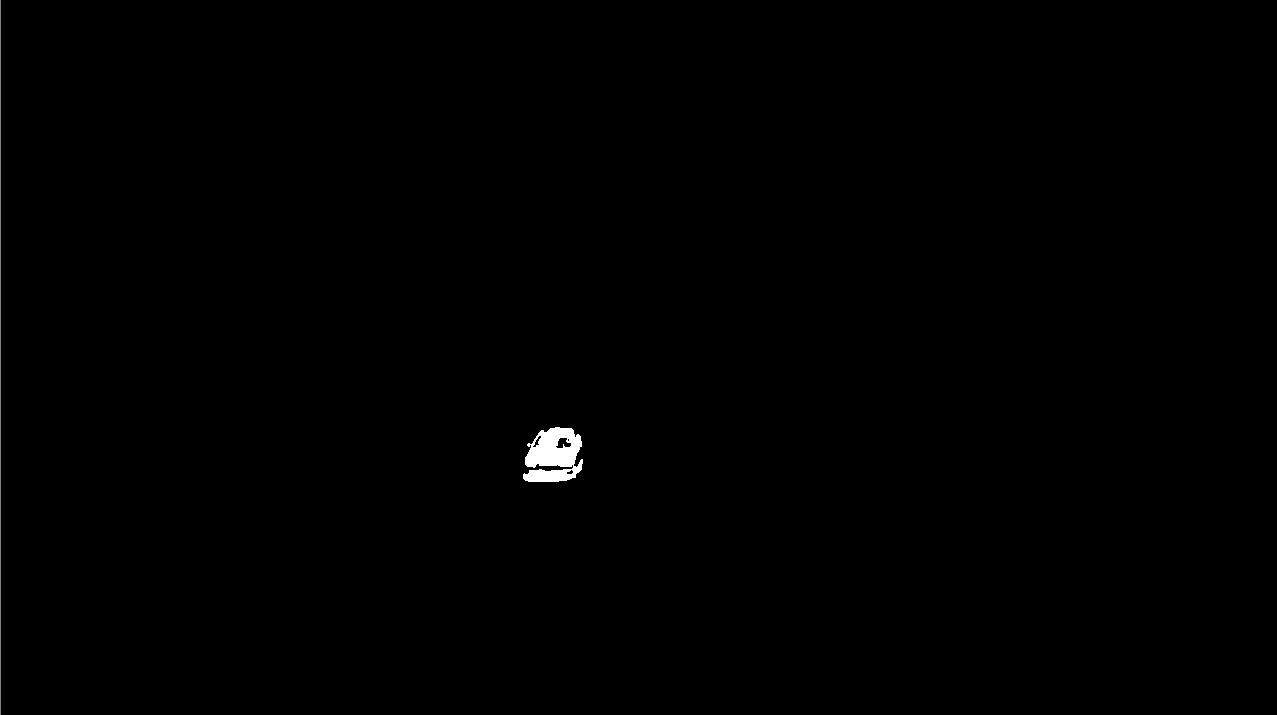
این تابع یک تصویر را به عنوان ورودی دریافت می‌کند و یک ماسک بر می‌گرداند که شامل بزرگ‌ترین کانتور آن است. ابتدا یک سری پردازش روی ورودی انجام می‌شود و سپس تصاویر پردازش شده تحویل تابع findContours پایتون می‌شود تا کانتور‌های آن استخراج شوند

| image\_median = cv2.medianBlur(image, 3)  image\_median\_pad = cv2.copyMakeBorder(  image\_median, 10, 10, 10, 10, cv2.BORDER\_CONSTANT)  image\_edge = cv2.Laplacian(  image\_median\_pad, cv2.CV\_8U) # laplacian edge detection  contours, \_ = cv2.findContours(  image\_edge, cv2.RETR\_TREE, cv2.CHAIN\_APPROX\_SIMPLE)  if len(contours) != 0:  image\_mask = np.zeros(image\_edge.shape, dtype=np.uint8)  c = max(contours, key=cv2.contourArea) # biggest contour  cv2.drawContours(image\_mask, [c], -1, (255, 255, 255), cv2.FILLED)  image\_mask = image\_mask[10:-10, 10:-10] # removing padding  else:  image\_mask = image  return image\_mask |
| --- |

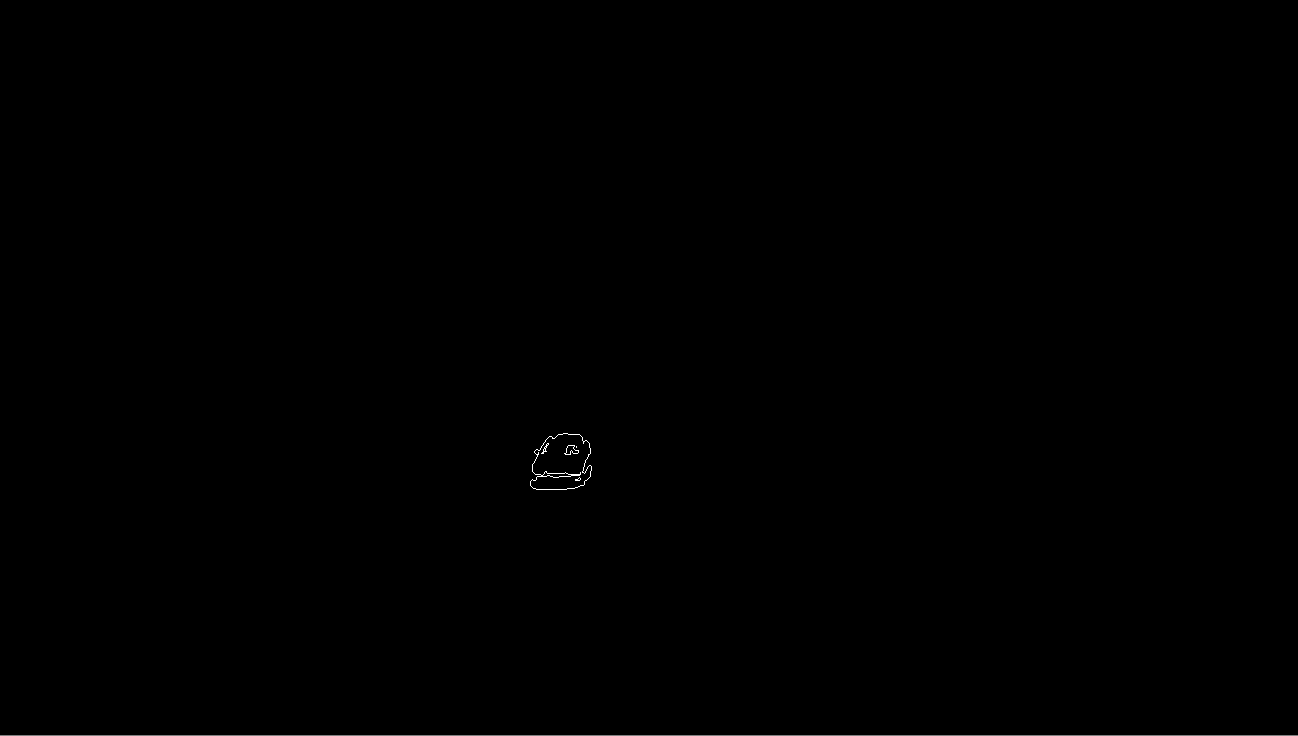
تصویر image ورودی به این شکل است



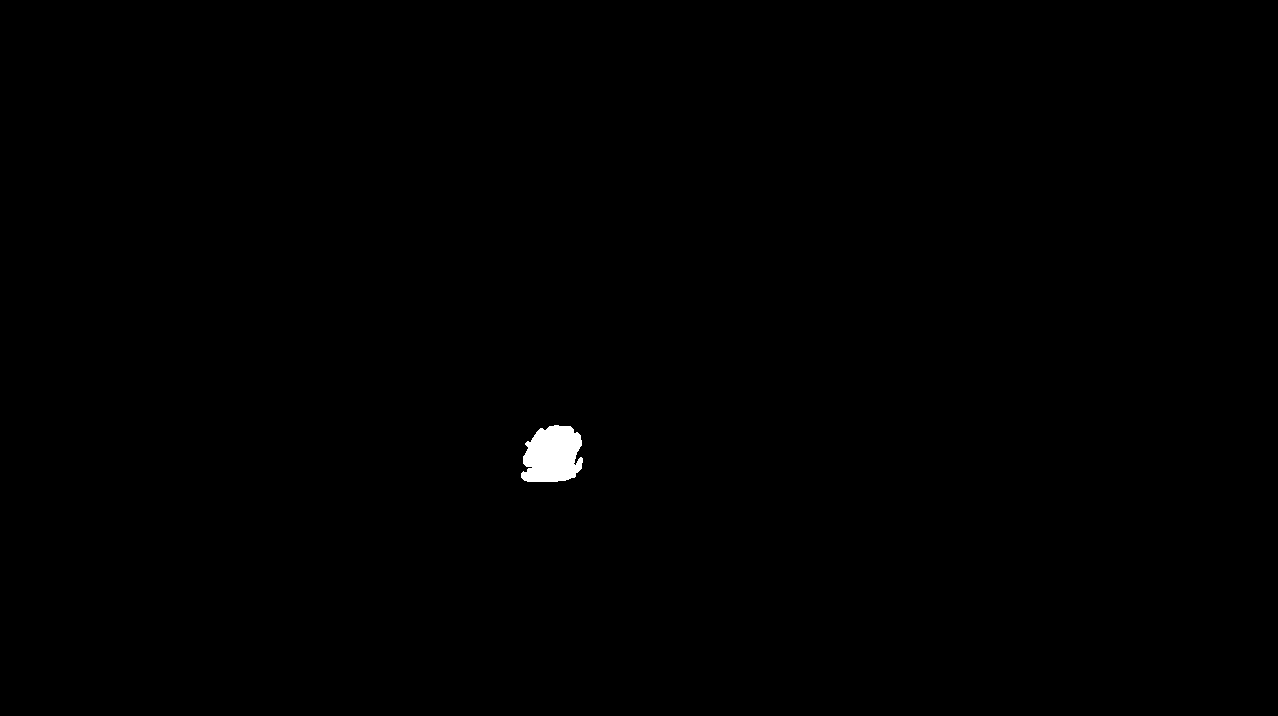
ابتدا فیلتر medianBlur روی این تصویر اعمال می‌شود تا نقاط توخالی آن تا حد ممکن پر شود



سپس یک zero padding انجام می‌شود و فیلتر لاپلاسین اعمال می‌شود تا لبه‌های تصویر استخراج شوند

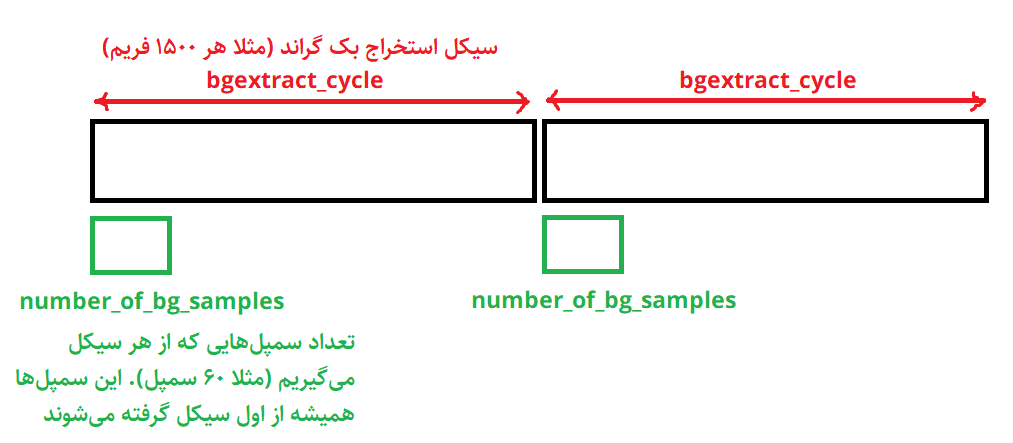


سپس این تصویر به تابع findContours داده می‌شود، کانتور‌های آن استخراج شده و کانتوری که بیشتری مساحت را دارد استخراج می‌شود



## 💬 **تابع do\_saveframes**

این تابع با یک سیکل مشخص (bgextract\_cycle) در main فراخوانی می‌شود و هدف آن این است که تعداد مشخصی سمپل از ویدیوی ورودی را ذخیره کند تا بعدا به کمک آن‌ها بک‌گراند را استخراج کنیم.



ورودی این تابع یک عکس image است که می‌خواهیم آن را ذخیره کنیم، یک عدد number است که نشان می‌دهد این نمونه چندمین نمونه در سیکل استخراج بک‌گراند فعلی است و یک عدد number\_of\_bg\_samples است که نشان می‌دهد کلا چند‌تا سمپل لازم داریم

| if number == 0:  if not os.path.exists('background'):  os.mkdir('background')  cv2.imwrite(os.path.join('background', '0' +  str(number) + '.jpg'), image)  elif number < 10:  cv2.imwrite(os.path.join('background', '0' +  str(number) + '.jpg'), image)  elif number < number\_of\_bg\_samples:  cv2.imwrite(os.path.join('background', str(number) + '.jpg'), image)  else:  pass |
| --- |

سپس اگر اولین نمونه بودیم یعنی number == 0 یک فولدر به اسم background در روت ساخته می‌شود و تمام سمپل‌ها در این فولدر ذخیره خواهند شد. سپس به ازای سایر number ها ، هر سمپل با یک شماره ذخیره می‌شود (این شماره با 0 شروع می‌شود زیرا فرض شده است که تعداد سمپل‌های مورد نیاز دو رقمی است اما اگر به تعداد سه‌رقمی سمپل نیاز بود باید با 00‌ شروع شود).

## 💬 **تابع do\_bgextract**

این تابع لیست آخرین تیوب‌های استخراج شده (تیوب‌هایی که خام هستند و هنوز سیناپس نشده‌اند) و همچنین f\_start\_buffer یعنی شماره‌ی فریم اولین سمپل را دریافت می‌کند. هدف این تابع این است که ابتدا سمپل‌های بک‌گراند را آماده‌سازی کند و سپس با تحویل این سمپل‌ها به تابع do\_bgmodel ، بک گراند را استخراج کند.

| if not os.path.exists('background'):  return  images = [cv2.imread(file) for file in sorted(glob.glob(  'background/\*.jpg'))] # Loading Saved Images  with open("aspects.txt", "rb") as fp:  (H, W) = pickle.load(fp) # loading Aspects |
| --- |

در ابتدا چک می‌شود که آیا فولدری به نام 'background' در دایرکتوری روت وجود دارد یا نه. اگر چنین دایرکتوری وجود نداشت یعنی تابع نمونه‌گیری از ویدیو قبلا اجرا نشده است و بنابراین نمونه‌ای وجود ندارد که بتوان از بک‌گراند را استخراج کرد و بنابراین تابع متوقف می شود. در غیر این صورت تمام فایل‌های jpg موجود در فولدر 'background' خوانده شده و در لیست images ریخته می شود . همچنین "aspects.txt" که در فایل main در روت برنامه ذخیره شده بود و حاوی طول و عرض ویدیو است باز می شود و در H و W ریخته می‌شود.

----

در ادامه سمپل ها را آمده می‌کنیم، یعنی هر سمپل را در نظر می گیریم و محل آبجکت‌هایی را که قبلا شناسایی شده‌اند سیاه می‌کنیم تا در سمپل وجود نداشته باشند.از لیست image که شامل همه نمونه‌ها است تصاویر را یکی یکی می‌خوانیم . به خاطر داشته باشید که این سمپل‌ها از فریم‌های متوالی گرفته شده‌اند بنابراین f\_start\_buffer شماره‌ی فریم اولین سمپل را نشان می‌دهد که یک شماره‌ی مطلق است و idx شماره‌ی هر سمپل در لیست سمپل‌ها را نشان می‌دهد که نسبی است.

| for idx in range(len(images)):  img = images[idx] # Image under Investigation  for tube in tube\_list:  idx2 = f\_start\_buffer + idx - tube.f\_start  cond1 = (f\_start\_buffer + idx >= tube.f\_start)  cond2 = (f\_start\_buffer + idx <= tube.f\_end)  cond3 = (idx2 <= len(tube.boxes) - 1) # enough frames  if cond1 and cond2 and cond3:  (x, y, w, h) = tube.boxes[idx2]  img[max(y, 0): min(y + h, H), max(x, 0): min(x + w, W), :] = np.zeros(  [min(y + h, H) - max(y, 0), min(x + w, W) - max(x, 0), 3], np.uint8)  images[idx] = img # Updating Images list |
| --- |

به ازای هر سمپل تمام تیوب‌هایی که در tube\_list است را بررسی می‌کنیم تا اگر در این تیوب‌ها آبجکتی پیدا شد که در این سمپل وجود دارد، آن آبجکت را سیاه کنیم.همان‌طور که می‌دانیم هر تیوب دو پارامتر tube.f\_start و tube.f\_end دارد که به ترتیب شماره‌ی مطلق اولین و آخرین فریم آن تیوب را مشخص می‌کند. برای این که مکان یک آبجکت را از تیوب متناظر استخراج کنیم و آن مکان را در سمپل سیاه کنیم، این شرط‌ها باید چک شوند تا مطمئن شویم شماره فریم تیوب تحت بررسی و سمپل یکی است.

| cond1 = (f\_start\_buffer + idx >= tube.f\_start) |
| --- |

شرط اول می‌گوید شماره مطلق سمپل یعنی f\_start\_buffer + idx باید از شماره مطلق اولین فریم تیوب تحت بررسی بیشتر باشد

| cond2 = (f\_start\_buffer + idx <= tube.f\_end) |
| --- |

شرط دوم می‌گوید شماره مطلق سمپل یعنی f\_start\_buffer + idx باید از شماره مطلق آخرین فریم تیوب تحت بررسی کمتر باشد

| idx2 = f\_start\_buffer + idx - tube.f\_start  cond3 = (idx2 <= len(tube.boxes) - 1) |
| --- |

یک اندیس idx2 معرفی شده که شماره‌ی نسبی سمپل تحت بررسی نسبت به ابتدای فریم تحت بررسی را نشان می‌دهد. برای درک بهتر یک مثال می‌زنیم. فرض کنیم سیکل استخراج بک‌گراند هر ۱۵۰۰ فریم، تعداد سمپل‌های مورد استفاده برای استخراج بک‌گراند ۶۰ سمپل و سیکل استفاده از YOLO هم هر ۳۰ فریک یک‌بار است. حالا فرض کنید در دومین سیکل استخراج بک‌گراند هستیم و تابع do\_bgextract را اجرا شده است. بنابراین f\_start\_buffer=1500 شماره مطلق اولین سمپل است و 1559 شماره مطلق ۶۰ امین سمپل است. اگر مثلا ۳۵-امین سمپل را در نظر بگیریم idx = 34 هست (شماره نسبی سمپل) و اگر مشغول بررسی یک تیوبی باشیم که tube.f\_start = 1530 و tube.f\_end=1559 است، آنگاه idx2 = 4‌ ، یعنی شماره نسبی سمپل در تیوب تحت بررسی، است ( f\_start\_buffer + idx - tube.f\_start = 1500 + 34 - 1530). حالا شرط سوم می‌گوید این اندیس idx2 باید کوچک‌تر از تعداد کل باکس‌های درون تیوب تحت بررسی باشد وگرنه این تیوب حاوی اطلاعات سمپل ما نیست.

حالا اگر هر سه شرط درست بود یعنی باکس idx2 در تیوب تحت بررسی حاوی اطلاعات سمپل است و (x, y, w, h) را از آن باکس استخراج کرده و مقدار سمپل به ازای این باکس را برابر ۰ قرار می‌دهیم. با اجرای کامل این حلقه تمام سمپل‌ها یک دور با تمام تیوب‌ها چک می‌شوند و مطمئن می‌شویم درتمام آبجکت‌هایی که در این سمپل‌ها وجود دارند سیاه شده ‌اند.



| background = do\_bgmodel(images)  if os.path.exists('background'):  shutil.rmtree('background')  cv2.imwrite('background.jpg', background) |
| --- |

سپس تابع این سمپل‌ها به تابع do\_bgmodel داده می‌شوند و این تابع بک‌گراند را بر می‌گرداند. برای اینکه این سمپل‌ها که از آن استفاده کردیم با سمپل‌هایی که در آینده می‌گیریم تداخلی نداشته باشند، در آخر کل فولدر background که حاوی سمپل‌ها بود را پاک می‌کنیم. در نهایت خروجی تابع do\_bgmodel را که یک عکس است به صورت background.jpg در روت ذخیره می‌کنیم.



## 💬 **تابع do\_bgmodel**

این تابع درون تابع do\_bgextract فراخوانی می‌شود و هدف آن این است که از روی سمپل‌هایی که قبلا پردازش شده‌اند (محل آبجکت‌های آن سیاه شده است) بک گراند را استخراج کند. ایده‌ی این تابع یک میانگین‌گیری ساده ازسمپل‌ها و استخراج بک‌گراند است اما از آنجا که روش‌های دیگری نظیر استفاده از فاصله‌ی ماهالانوبیس در نسخه‌های اولیه آن مدنظر بودند، به صورت یک تابع جداگانه نوشته شده است.

| r\_matrix = np.zeros(  (images[0].shape[0], images[0].shape[1], len(images)), dtype='uint8')  g\_matrix = np.zeros(  (images[0].shape[0], images[0].shape[1], len(images)), dtype='uint8')  b\_matrix = np.zeros(  (images[0].shape[0], images[0].shape[1], len(images)), dtype='uint8')  for idx in range(len(images)): # separating r, g, b  img = images[idx]  r\_matrix[:, :, idx] = img[:, :, 2]  g\_matrix[:, :, idx] = img[:, :, 1]  b\_matrix[:, :, idx] = img[:, :, 0] |
| --- |

Images یک لیست از سمپل‌ها هست. در ابتدا کانال‌های r‌و g‌و b این سمپل‌ها را جدا می‌کنیم و به ترتیب در ماتریس‌های r\_matrix و g\_matrix و b\_matrix می‌ریزیم .این ماتریس‌ها سه بعدی هستند به طوری که بعد سوم آن به تعداد سمپل‌های موجود است و به ازای هر سمپل یک کانال آن سمپل را نگه‌داری می‌کند.

همچنین ماتریس‌های دو بعدی زیر را تعریف می‌کنیم که ماتریس‌های num (صورت) مجموع ماتریس‌های بالا (مجموع روی بعد سوم)‌ را نگه‌داری می‌کنند و ماتریس‌های denum (مخرج) ماتریس‌هایی دو بعدی هستند که هر درایه‌ی آن متناظر با یک پیکسل از سمپل‌ها است و نشان می‌دهد چه تعداد سمپل در آن پیکسل خاص مقدار غیر صفر داشته‌اند.

| r\_num = np.sum(r\_matrix, axis=2)  g\_num = np.sum(g\_matrix, axis=2)  b\_num = np.sum(b\_matrix, axis=2)  r\_denum = np.sum(r\_matrix != 0, axis=2)  g\_denum = np.sum(g\_matrix != 0, axis=2)  b\_denum = np.sum(b\_matrix != 0, axis=2) |
| --- |

حالا با ایگنور کردن حالت‌های 0/0 و تبدیل مقادیر INF به 0‌ می‌توانیم میانگین مورد نظر را، که همان تقسیم صورت بر مخرج است، برای هر کانال بنویسیم.

| with np.errstate(divide='ignore', invalid='ignore'):  r\_mean = np.true\_divide(  r\_num, r\_denum)  r\_mean[r\_mean == np.inf] = 0 # replacing inf with 0  r\_mean = np.nan\_to\_num(r\_mean)  g\_mean = np.true\_divide(  g\_num, g\_denum)  g\_mean[g\_mean == np.inf] = 0 # replacing inf with 0  g\_mean = np.nan\_to\_num(g\_mean)  b\_mean = np.true\_divide(  b\_num, b\_denum)  b\_mean[b\_mean == np.inf] = 0 # replacing inf with 0  b\_mean = np.nan\_to\_num(b\_mean) |
| --- |

در نهایت با ترکیب سه کانال بک‌گراند به دست می‌اید

| background = np.dstack((b\_mean, g\_mean, r\_mean)).astype(np.uint8) |
| --- |

## 

## 💬 **تابع do\_motion**

این تابع بعد از خواندن هر فریم از ورودی اجرا می‌شود و از مقایسه‌ی فریم فعلی و فریم قبلی motion‌ را استخراج می‌کند. ورودی‌ها به این شکل هستند:



## 

| optical\_flow = cv2.optflow.DualTVL1OpticalFlow\_create(  nscales=1, epsilon=0.05, warps=1)  frame2\_grayscale = cv2.cvtColor(frame2, cv2.COLOR\_BGR2GRAY)  frame1\_grayscale = cv2.cvtColor(frame1, cv2.COLOR\_BGR2GRAY)  # Optical Flow  flow = optical\_flow.calc(frame1\_grayscale, frame2\_grayscale, None) |
| --- |

این دو ورودی به grayscale تبدیل شده و اپتیکال فلو بین آن دو محاسبه می‌شود

خروجی اپتیکال فلو ابتدا تبدیل به مقدار HSV می‌شود و در متغیر flow\_hsv ریخته می‌شود و سپس این مقدار HSV ابتدا تبدیل به RGB‌ و سپس تبدیل به grayscale می‌شود:

| flow\_hsv = np.zeros\_like(frame2)  flow\_hsv[..., 1] = 255  flow\_mag, flow\_ang = cv2.cartToPolar(flow[..., 0], flow[..., 1])  flow\_hsv[..., 0] = flow\_ang\*180/np.pi/2  flow\_hsv[..., 2] = cv2.normalize(flow\_mag, None, 0, 255, cv2.NORM\_MINMAX)  flow\_rgb = cv2.cvtColor(flow\_hsv, cv2.COLOR\_HSV2BGR)  flow\_gray = cv2.cvtColor(flow\_rgb, cv2.COLOR\_BGR2GRAY) |
| --- |



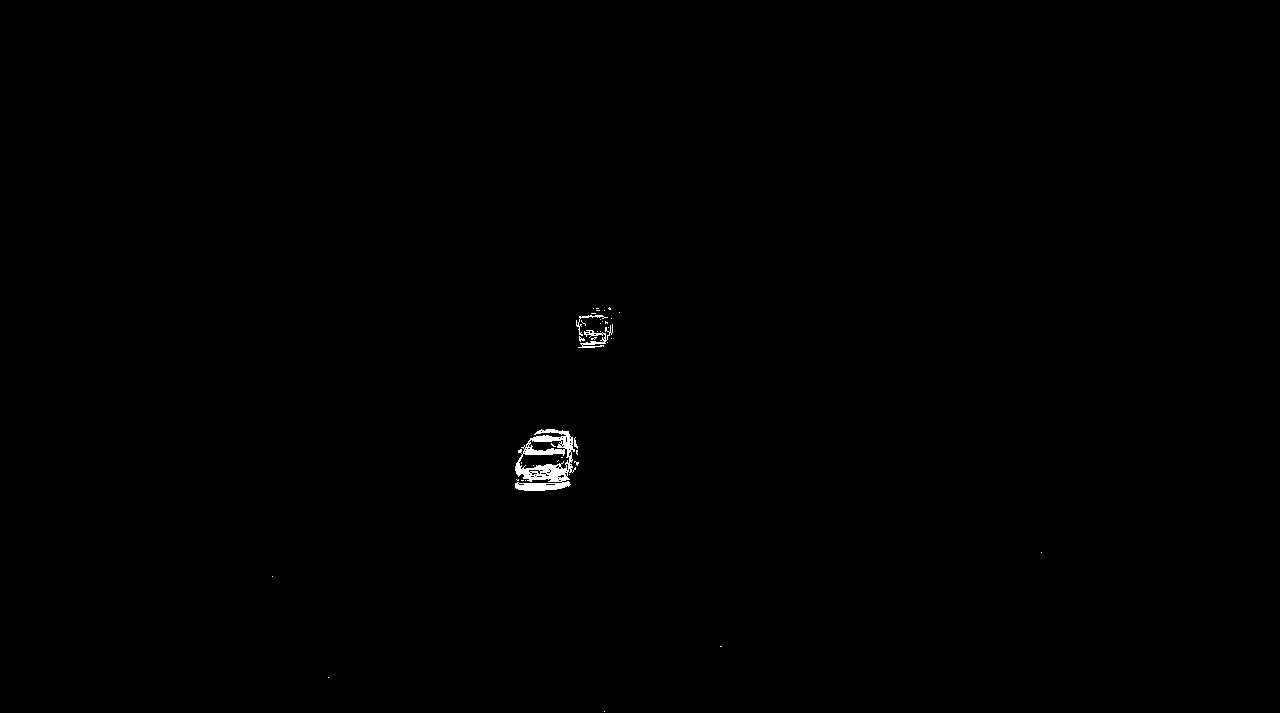
یک بار هم میاییم و اختلاف بین دو فریم را محاسبه می‌کنیم

| frame\_diff = cv2.absdiff(frame2\_grayscale, frame1\_grayscale)  frame\_diff = frame\_diff.astype('uint8') |
| --- |



تا اینجا موشن را از دو روش به دست آوردیم. یکی اختلاف بین optical flow دو فریم متوالی و یکی اختلاف بین خود دو فریم متوالی. اما در صورت threshold گذاشتن روی مقادیر خروجی مشاهده می‌شود که نویز، به خصوص در جاهایی مثل چمن‌ها که با وزش باد حرکت کرده‌اند، بسیار زیاد است. بنابراین برای حذف کردن نویز، جواب نهایی را به صورت ضرب این دو جواب (AND کردن) در نظر می‌گیریم و آنگاه روی این جواب threshold می‌گیریم:

| frame\_diff\_flow = cv2.multiply(flow\_gray/255, frame\_diff/255) \* 255  frame\_diff\_flow = frame\_diff\_flow.astype('uint8')  \_, frame\_diff\_flow\_theresh = cv2.threshold(  frame\_diff\_flow, 1, 255, cv2.THRESH\_BINARY) |
| --- |



مشاهده می‌شود که خروجی کمترین نویز ممکن را دارد

## 💬 **تابع do\_fix\_box**

گاهی باکس‌هایی که از YOLO پیدا می‌شوند دقیق نیستند به طوری‌که در لبه‌ها بیشتر یا کمتر از شکلی که واقعا وجود دارد بریده شده است. این تابع با کمک motion که در do\_motion استخراج شد سعی می‌کند خروجی YOLO را از طریق کوچک و بزرگ‌ کردن باکس‌ها اصلاح کند. این تابع frame فعلی، frame\_motion و یک bbox (که باید اصلاح شود) را دریافت می‌کند:

| pts = np.where(frame\_motion == 255) # mask points  pts\_x = pts[1] # mask points X  pts\_y = pts[0] # mask points Y |
| --- |

در ابتدا نقاطی از motion که مقدار 255 دارند جدا شده و مقدار x و y آن ها در pts\_x و pts\_y قرا رمی‌گیرد (دقت شود که motion مقادیر باینری ۲۵۵ و ۰ دارد)

همچنین مختصات باکس از bbox استخراج می‌شود

| (x, y, w, h) = bbox # extracting values  x1 = x  y1 = y  x2 = x + w  y2 = y + h |
| --- |

که x1 و y1 سمت چپ باکس و x2 و y2‌ سمت راست باکس هستند. پارامتر window نشان می‌دهد سایز پنجره‌ها حداکثر چند درصد می‌تواند تغییر کند. مثلا اگر window = 0.2 یعنی بعد از اجرای تابع do\_fix\_box سایز پنجره نهایتا می‌تواند بین 1.2 تا 0.8 سایز فعلی شود. از روی این پارامتر، مقدار تغییرات طول و عرض همسایگی مجاز به این شکل محاسبه می‌شود:

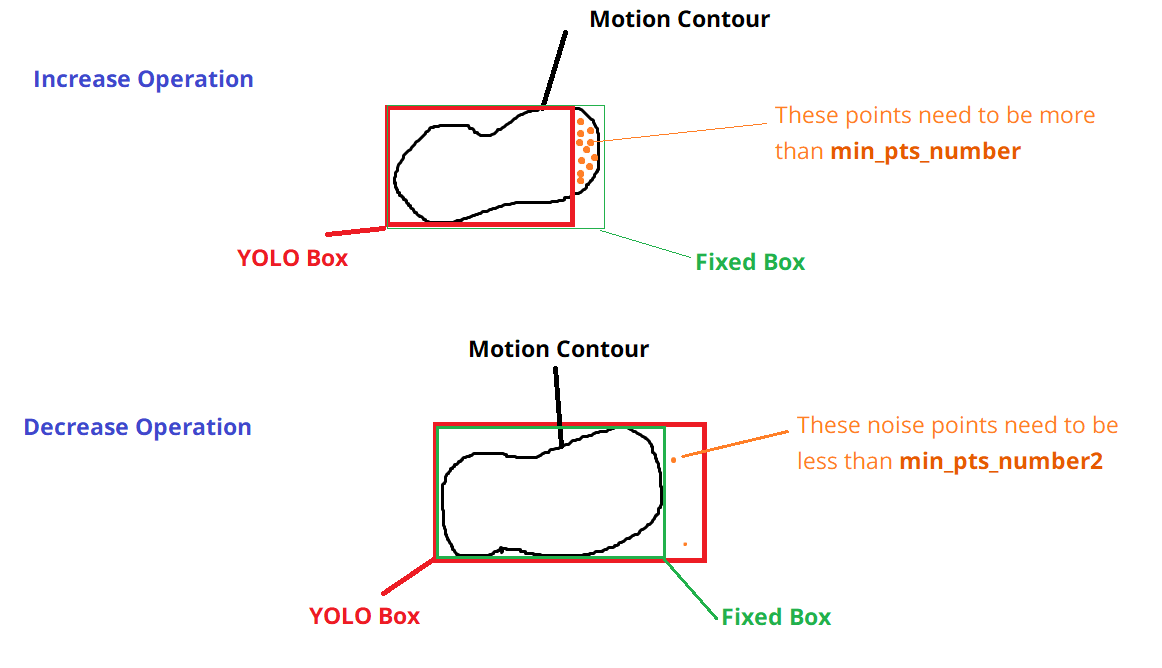
| window\_w = round(window \* w/2) # increase in width  window\_h = round(window \* h/2) # increase in depth |
| --- |

گاهی باکس‌های ورودی تابع do\_fix\_box بسیار کوچک هستند که اکثرا مربوط به آبجکت‌های بسیار دور هستند و در این صورت دلیل ندارد این باکس‌ها کوچک‌تر بشوند (اما می‌توانند بزرگ‌تر شوند). بنابراین min\_window\_area را حداقل مساحتی تعیین می‌کنیم که کاهش سایز باکس می‌تواند انجام شود و باکس‌هایی که زیر این مساحت هستند فقط می‌توانند افزایش یابند.

دو پارامتر min\_pts\_number و min\_pts\_number2 به ترتیب دو آستانه‌ای هستند که افزایش یا کاهش باکس را تصمیم‌گیری می‌کنند

| min\_pts\_number = round(20 \* np.sqrt(80\*window\_area/(H\*W)))  min\_pts\_number2 = round(20 \* np.sqrt(80\*window\_area/(H\*W))) |
| --- |

همان‌طور که دیده می‌شود سعی شده این تابع، یک تابعی غیر خطی از نسبت مساحت آبجکت (window\_area) به مساحت کل فریم (H\*W) باشد. یعنی هر قدر آبجکت بزرگ‌تر است این آستانه هم بزرگ‌تر خواهد بود. در ادامه خواهیم دید که این آستانه تعداد نقاط خارج (یا داخل) مرز‌های باکس فعلی را مشخص می‌کند یعنی مثلا اگر min\_pts\_number = 10 بود و در همسایگی خارجی مرز راست باکس حداقل ۱۰ نقطه motion وجود داشت یعنی این باکس باید به سمت راست افزایش پیدا کند. همچنین اگر min\_pts\_number2 = 10 و درهمسایگی داخلی مرز راست کمتر از ۱۰ نقطه motion وجود داشته باشد یعنی این باکس باید به سمت چپ کاهش پیدا کند.



مفاهیم توضیح داده شده در شرط‌های ۱ تا ۹ ظاهر شده است

| cond1 = pts\_x[(pts\_x < x1) & (pts\_x > x1 - window\_w)  & (pts\_y > y1 - window\_h) & (pts\_y < y2 + window\_h)]  cond2 = pts\_y[(pts\_y < y1) & (pts\_y > y1 - window\_h)  & (pts\_x > x1 - window\_w) & (pts\_x < x2 + window\_w)]  cond3 = pts\_x[(pts\_x > x2) & (pts\_x < x2 + window\_w)  & (pts\_y < y2 + window\_h) & (pts\_y > y1 - window\_h)]  cond4 = pts\_y[(pts\_y > y2) & (pts\_y < y2 + window\_h)  & (pts\_x < x2 + window\_w) & (pts\_x > x1 - window\_w)]  cond5 = pts\_x[(pts\_x > x1) & (pts\_x < x1 + window\_w)  & (pts\_y > y1 - window\_h) & (pts\_y < y2 + window\_h)]  cond6 = pts\_y[(pts\_y > y1) & (pts\_y < y1 + window\_h)  & (pts\_x > x1 - window\_w) & (pts\_x < x2 + window\_w)]  cond7 = pts\_x[(pts\_x < x2) & (pts\_x > x2 - window\_w)  & (pts\_y < y2 + window\_h) & (pts\_y > y1 - window\_h)]  cond8 = pts\_y[(pts\_y < y2) & (pts\_y > y2 - window\_h)  & (pts\_x < x2 + window\_w) & (pts\_x > x1 - window\_w)]  cond9 = window\_area >= min\_window\_area # to ignore fixing small boxes |
| --- |

شرط‌های ۱ تا ۴ شرط‌های لازم برای افزایش سایز باکس در ۴ ضلع مستطیل و شرط‌های ۵ تا ۸ شرط‌های لازم برای کاهش سایز باکس در ۴ ضلع مستطیل هستند. همچنین شرط ۹ تضمین می‌کند باکس از یک مساحتی بزرگ‌تر است (در غیر این صورت عملیات کاهش نمی‌تواند روی آن انجام شود). به طور نمونه شرط ۱ را در شکل زیر می‌بینیم:



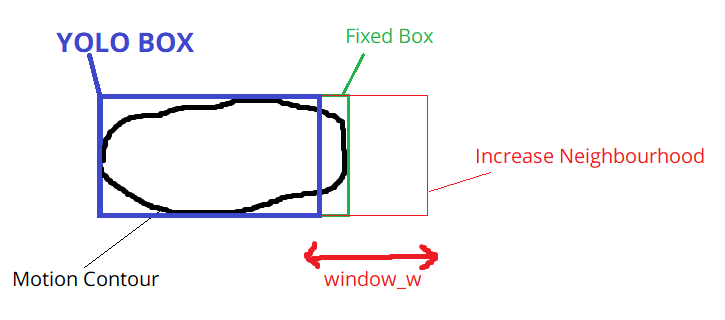
با توجه به شرط‌های مطرح شده می‌توان حالت‌های عملیات افزیش را به این شکل نوشت

| if len(cond1) > min\_pts\_number:  delta = abs(round(x1 - np.min(cond1)))  x1 = max(x1 - delta, 0)  if len(cond2) > min\_pts\_number:  delta = abs(round(y1 - np.min(cond2)))  y1 = max(y1 - delta, 0)  if len(cond3) > min\_pts\_number:  delta = abs(round(np.max(cond3) - x2))  x2 = min(x2 + delta, W)  if len(cond4) > min\_pts\_number:  delta = abs(round(np.max(cond4) - y2))  y2 = min(y2 + delta, H) |
| --- |

همچنین حالت‌های عملیات کاهش به این صورت است

| if len(cond5) < min\_pts\_number2 and cond9:  if len(cond5) == 0:  delta = window\_w  else:  delta = abs(round(x1 - np.min(cond5)))  if len(cond5) == 0 and len(cond6) == 0 and len(cond7) == 0 and len(cond8) == 0:  delta = 0  x1 = min(x1 + delta, W)  if len(cond6) < min\_pts\_number2 and cond9:  if len(cond6) == 0:  delta = window\_h  else:  delta = abs(round(y1 - np.min(cond6)))  if len(cond5) == 0 and len(cond6) == 0 and len(cond7) == 0 and len(cond8) == 0:  delta = 0  y1 = min(y1 + delta, H)  if len(cond7) < min\_pts\_number2 and cond9:  if len(cond7) == 0:  delta = window\_w  else:  delta = abs(round(np.max(cond7) - x2))  if len(cond5) == 0 and len(cond6) == 0 and len(cond7) == 0 and len(cond8) == 0:  delta = 0  x2 = max(x2 - delta, 0)  if len(cond8) < min\_pts\_number2 and cond9:  if len(cond8) == 0:  delta = window\_h  else:  delta = abs(round(np.max(cond8) - y2))  if len(cond5) == 0 and len(cond6) == 0 and len(cond7) == 0 and len(cond8) == 0:  delta = 0  y2 = max(y2 - delta, 0) |
| --- |

دقت شود که در تمام شرط‌های بالا یک ماکسیموم همسایگی افزایش (یا کاهش) به اندازه window\_w و window\_h داریم. اما مطابق شکل زیر به اندازه‌ی دور ترین نقطه‌ی کانتور motion که در همسایگی وجود دارد این افزایش را انجام می‌دهیم.



در پایان مختصات اصلاح شده را بر می‌گردانیم

| bbox = (x1, y1, x2 - x1, y2 - y1) |
| --- |