Robotics Lab 6: Basic Computer Vision

Authors: Stuart Harley, Nathan Chapman

Date: 10/17/2021

Abstract

In robotics, utilization of a vision sensor, or camera, can allow for a lot of information to be gained. However, this information can be difficult to extract from the recorded images. While vision comes naturally to us as humans, this is not the case for robots. For robots to utilize vision sensors they often need to use computer vision methods to identify and track objects object of interest in images. In this lab, we implement a variety of basic computer vision functionality, specifically thresholding and blob detection, dilation and erosion, edge detection, and connected components to both a static image and a live video feed. We were able to implement all functionality, allowing us to track an object such as a balloon around the screen.

Methods

In this lab, we work with the open-source cross-platform computer vision library OpenCV. This is a library of programming functions aimed at real-time computer vision. Every method that we hand code in this lab already exists in the OpenCV library. We use the OpenCV library in this lab to check that our hand-coded methods are performing correctly. We also use it to speed up the run-time since these methods are optimized in the OpenCV library. In this lab, we did not focus on optimizing our hand-coded solutions. We only focused on the correctness of our solutions. While the OpenCV library is fast to use and we do not need to implement our own solutions to use it, it can be confusing to use. The methods contain several parameters that can be confusing to understand exactly what they do. Also, the documentation on OpenCV's website is not great so we had to look at other web pages for examples of how to correctly use the methods.

In this lab, we also deal with loading/viewing images. We were provided with the code to do this, but we did run into some minor issues. When using our own implementation for the computer vision functions, the rgb values were between –128 and 127. But when using OpenCV the RGB values were between 0 and 255. Because of this, we had to be careful when specifying the colors for the lines and centroids we wanted in our images.

In this lab, we also dealt with a live camera feed. We were provided with the code to get this from our laptop's webcam. However, when using this video feed to run segmentation on we found that the frame rate of the video slowed down due to the computation of the segmentation functions.

For tasks 1 through 3, we use the test image shown below in Image 1 to test our implementations. We are focusing on the dark blue/purplish color contained in Image 1 when testing our solutions.

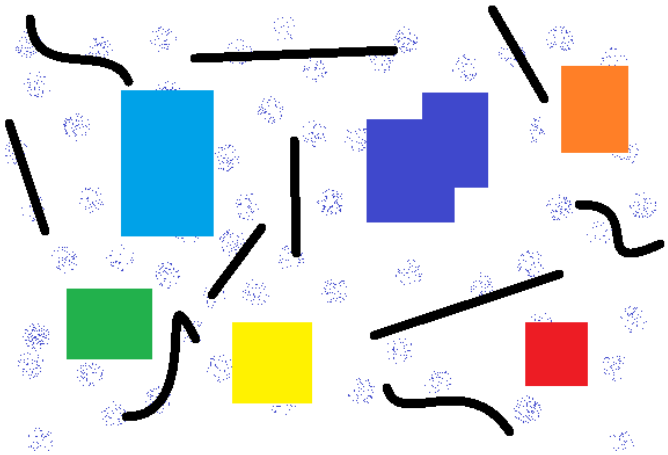

Image 1: Test Image used for testing our solutions

Task 1

In task 1 we implement Image Segmentation. Image segmentation, also known as blob tracking, involves identifying pixels of an object of interest in an image. After identifying the pixels, those pixels are set to white while everything else is set to black. In this lab, we are implementing image segmentation using an image threshold. This means that we develop a range of rgb values for the object that we want to segment. Based on the range we can identify all pixels of interest and create a threshold/segmented image. While this method is simple, it is not very robust, running into issues with lighting and shading in the image. It can also be difficult to identify a threshold that is not too narrow or wide as to allow for all pixels in the desired object to be segmented but not allow too much noise.

For our hand-coded solution, we created a segment function which takes in 3 parameters: orig – the original image, lower – the lower rgb bound, and upper – the upper rgb bound. We then loop through every pixel in the image and check if the pixel values are equal to or between the two bounds. If they are, we set the corresponding pixel in the return image to white, otherwise we set the pixel to black.

For this task and the following tasks, we also hand-coded a get_center method to find the centroid of all the selected pixels in an image. To do this we take in a segmented image and then sum the x coordinates and the y coordinates of all the white pixels. We then divide these sums by the total number of white pixels to find the centroid of the selected pixels. We then used this centroid to draw a small circle on our images to show where the centroid is located.

Task 2

In task 2 we implement Dilation and Erosion. These are filtering techniques that are used after image segmentation to improve the results. Dilation adds pixels around the edges of the blobs found in image segmentation. This can help fill in gaps in the thresholding to get a more

complete blob. Doing the opposite, Erosion removes isolated pixels and pixels around the edge of the blob. This can eliminate noise and clean up the segmentation.

For our hand-coded solution, we created a dilate function which takes in an image parameter. We then created a return array of 2 less pixels in the x and y directions because we are not dealing with edge cases in our implementations. We then loop through all the pixels and check the 3x3 area at each pixel location except the edges. If any pixel in this area is found to be white, then the return value at that location is set to white. If none of the 9 pixels in this area are white (all are black), then the return value at that location is set to black.

For our hand-coded erosion method, we did the opposite of our dilate function. We created a erode function which takes in an image parameter. We then created a return array of 2 less pixels in the x and y directions because we are not dealing with edge cases in our implementations. We then loop through all the pixels and check the 3x3 area at each pixel location except the edges. If any pixel in this area is found to be black, then the return value at that location is set to black. If none of the 9 pixels in this area are black (all are white), then the return value at that location is set to white.

Task 3

In task 3 we implement Edge detection. Edge detection is a common imaging processing step. Using it we can identify the outline of objects and features of interest. To implement edge detection we are using kernels and convolution. Two of the simplest edge detector kernels that we use in this lab are the Sobel and Laplacian kernels. The Sobel kernel has horizontal and vertical versions that find horizontal and vertical edges, respectively. The Laplacian edge kernel that we use in this lab finds both horizontal and vertical edges.

For our hand-coded solution, we created a Laplacian function which takes in an image parameter. We then created a return array of 2 less pixels in the x and y directions because we are not dealing with edge cases in our implementations. We then loop through all the pixels and apply the 3x3 Laplacian kernel to each pixel not on an edge of the image. The return value of each pixel is the result of the kernel being applied to the image at that pixel location.

For our hand-coded solution Sobel edge detection, we created 2 functions, hor_sobel and vert_sobel, which each take in an image parameter. We then created a return array of 2 less pixels in the x and y directions because we are not dealing with edge cases in our implementations. We then loop through all the pixels and apply the 3x3 Horizontal Sobel kernel or the 3x3 Vertical Sobel kernel to each pixel not on an edge of the image. The return value of each pixel is the result of the kernel being applied to the image at that pixel location.

Task 4

In task 4 we implement thresholding on a live video we take using the webcam of our laptop. We used a red balloon as the object we wanted to segment in the video. To do this, we first

identified rgb threshold values to only select the red balloon pixels in the video. Then, for each frame of the video, we run OpenCV's inRange function to select the red balloon pixels. We then identify the centroid of the selected balloon pixels and draw a grey center point on the image. While this was running, we moved the balloon around in the video to check if the selected pixels and the centroid moved properly with it.

Task 5

In task 5, we were asked to implement task 4, but by using OpenCV's connectedComponentsWithStats to track the largest blob in the image. That means that we take a live video feed, segment it to detect the threshold of the red balloon pixels, and then pass that image to the method. This method then returns statistics about the components in the image. Using these, we can get the component with the largest area, and then draw a circle on the centroid of that item. We check to ensure we are, in fact, getting the largest component, and that the centroid is following that object.

Results

Task 1

Shown below in images 2 and 3 are the results of our hand-coded Image segmentation method and OpenCV's inRange function. We selected a rgb range that represented only the dark blue/purplish color from the test image. Contained in both images is a grey circle showing the centroid of the white pixels of the image. As can be seen below, the results of both are the same, showing that our hand-coded method is functioning correctly.

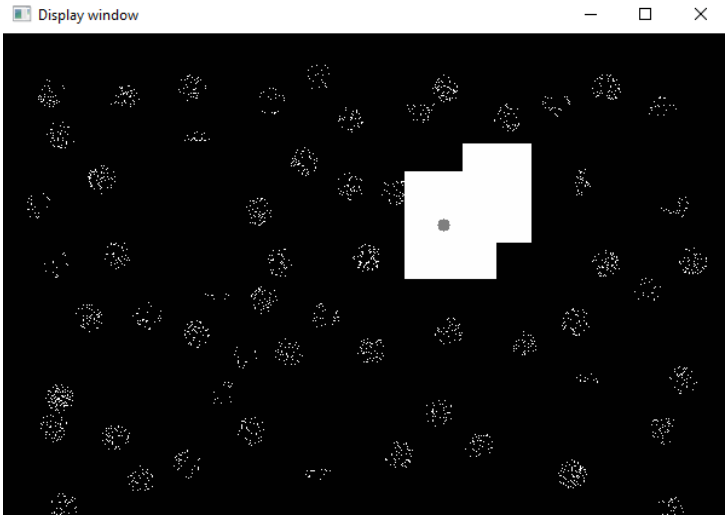

Image 2: Result of our hand-coded Image Segmentation function

Image 3: Result of OpenCV's inRange function

Task 2

Shown below in images 4 and 5 are the results of OpenCV's inRange function followed by our hand-coded dilate method and OpenCV's dilate method. Contained in both images is a grey circle showing the centroid of the white pixels of the image. As can be seen below, the results of both are the same, showing that our hand-coded method is functioning correctly.

Image 4: Result of our hand-coded Dilation method

Image 5: Result of OpenCV's Dilation method

Shown below in images 6 and 7 are the results of OpenCV's inRange function followed by our hand-coded erosion method and OpenCV's erosion method. Contained in both images is a grey circle showing the centroid of the white pixels of the image. As can be seen below, the results of both are the same, showing that our hand-coded method is functioning correctly.

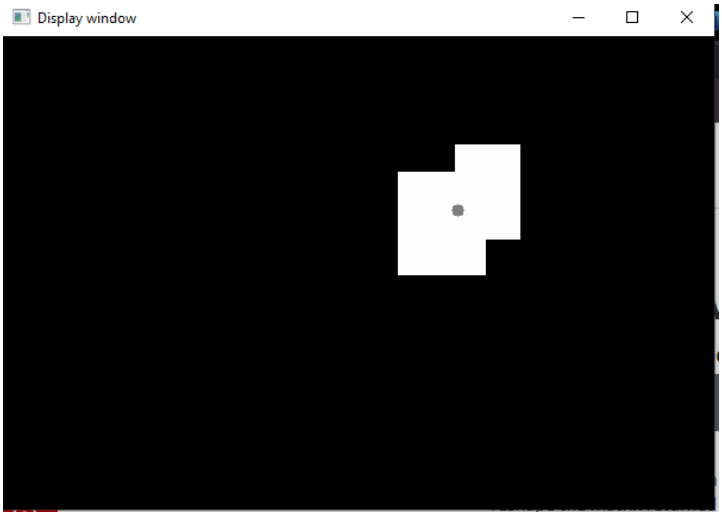

Image 6: Result of our hand-coded Erosion method

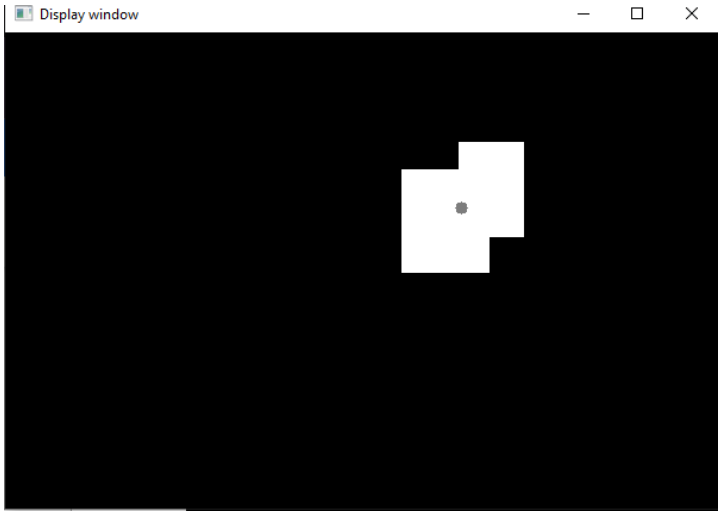

Image 7: Result of OpenCV's Erosion method

Task 3

Shown below in images 8 and 9 are the results of OpenCV's inRange function followed by OpenCV's erosion method and then our hand-coded Laplacian filter edge detection method and OpenCV's Laplacian filter edge detection method. Contained in both images is a grey circle showing the centroid of the white pixels of the image. As can be seen below, the results are slightly different but they both contain all the edges showing that our hand-coded method is functioning correctly.

Image 8: Result of our hand-coded Laplacian filter edge detection method

Image 9: Result of OpenCV's Laplacian filter edge detection method

Shown below in images 10 and 11 are the results of OpenCV's inRange function followed by OpenCV's erosion method and then our hand-coded Laplacian filter edge detection method and OpenCV's Laplacian filter edge detection method followed by our hand-coded horizontal Sobel filter edge detection method and OpenCV's horizontal Sobel filter edge detection method, respectively. As can be seen below, the results are slightly different but they both only contain only the horizontal edges which shows that our method is functioning correctly.

Image 10: Result of our hand-coded horizontal Sobel filter edge detection method

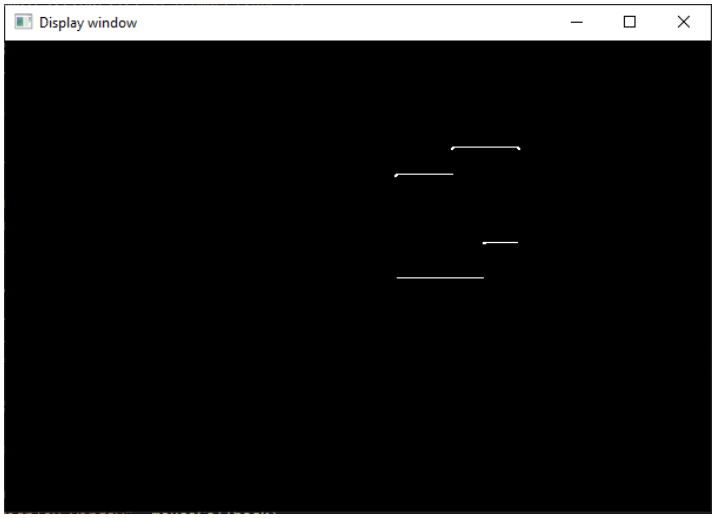

Image 11: Result of OpenCV's Sobel filter edge detection method

Shown below in images 12 and 13 are the results of OpenCV's inRange function followed by OpenCV's erosion method and then our hand-coded Laplacian filter edge detection method and OpenCV's Laplacian filter edge detection method followed by our hand-coded vertical Sobel filter edge detection method and OpenCV's vertical Sobel filter edge detection method, respectively. As can be seen below, the results are slightly different but they both only contain only the vertical edges which shows that our method is functioning correctly.

Image 10: Result of our hand-coded vertical Sobel filter edge detection method

Image 11: Result of OpenCV's vertical Sobel filter edge detection method

Task 4

Shown below in image 12 are 3 screenshots of running OpenCV's image segmentation inRange function on a live video. In the video we are holding a red balloon which we based the rgb threshold values on. Contained in the image is a grey circle showing the centroid of the white pixels of the image. This is effectively the center of the balloon.

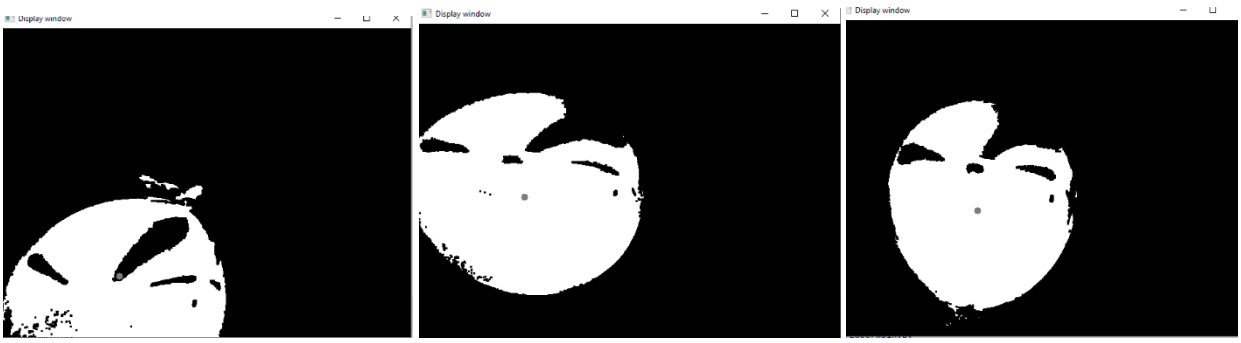

Image 12: Results of Image Detection on a live video containing a red balloon

Task 5

In image 13 (below), there are 3 screenshots of running OpenCV's connectedComponentsWithStats with a segmented image threshold focused on the red of the balloon. After converting the image to the segmented image (1 channel image) we then pass it to the connectedComponentsWithStats method, which returns information about the detected components. We then get the largest component (which in the middle image is the black, or non-red part of the image), and print out the centroid of that component. As shown, the centroid follows the largest component, moving right in the first image, left and up in the

second (since we are negating many black pixels in the lower right), and finally UP and left in the last image.

Image 13: Results of Image Detection on a live video containing red balloon.

Discussion

1. The OpenCV methods run much faster than our hand-coded methods. The main way in which they do so is by utilizing parallelism. Because the images are simply matrices of pixel values, the thresholding and kernel operations in dilation/erosion and edge detection can be run in parallel to speed up the run-time of these methods. In our hand-coded solutions, we loop through each pixel individually which takes much longer. NumPy is specifically designed to handle matrix operations very quickly by storing values in memory sequentially, and the OpenCV methods take advantage of these capabilities.

2. In task 3 we can get all the edges using the Laplacian filter. We were then able to get only the vertical or horizontal edges using the correct Sobel filter. There is an advantage in being able to differentiate between the horizontal and vertical edges out of all the edges. This is because there are many objects like roads, entryways, corners, etc. that show up as vertical or horizontal lines in an image. If we can identify only the vertical or horizontal lines in an image, we can more easily identify these kinds of objects in images because we can get rid of noisy non-vertical or horizontal edges.

3. Connected Components works by detecting all the items in the image. Our theory is that it uses a few different algorithms such as line detection to help determine the outline of items. Then, it gives statistics about the items in the image, such as the area, centroid, and minimum and maximum x and y values.

Supplementary Materials

Contained in this section are code screenshots for the computer vision methods and tasks 1 through 5. The only python file submitted is main.py which contains all the functionality of this lab.

Python Code

```
import cv2 as cv
import sys
import numpy as np

callBackImg = None
cX = 0
cY = 0


def printRGB(event, x, y, flags, params):
    # EVENT_MOUSEMOVE
    if event == cv.EVENT_LBUTTONDOWN:
        print(str(x) + " " + str(y) + " " + str(callBackImg[y][x][2]) + " " + str(callBackImg[y][x][1]) + " " + str(
            callBackImg[y][x][0]))


def mouseCallback(event, x, y, flags, param):
    # EVENT_MOUSEMOVE
    global cX, cY
    if event == cv.EVENT_MOUSEMOVE:
        cX = x
        cY = y
    elif event == cv.EVENT_LBUTTONDOWN:
        print(str(x) + " " + str(y) + " (" + str(callBackImg[y][x]) + ", " + str(callBackImg[y][x]) + ", " + str(
            callBackImg[y][x]) + ")")


def myGray(orig):
    (height, width, channels) = orig.shape
    ret = np.zeros((height, width, 1), dtype=np.int8)
    for i in range(height):
        for j in range(width):
            g = int((int(orig[i][j][0]) + int(orig[i][j][1]) + int(orig[i][j][2])) / 3.0) - 128
            ret[i][j] = g
    return ret

def readVid():
    global callBackImg
    cap = cv.VideoCapture(0)
    if not cap.isOpened():
        print("Cannot open camera")
        exit()
    while True:
        # Capture frame-by-frame
        ret, callBackImg = cap.read()
        # if frame is read correctly ret is True
        if not ret:
            print("Can't receive frame (stream end?). Exiting ...")
            break

        cv.circle(callBackImg, (cX, cY), 10, (255, 0, 0), 1)
        cv.imshow("Display window", callBackImg)
        cv.setMouseCallback("Display window", mouseCallback)

        k = cv.waitKey(1)
        if k == ord('s'):
            cv.imwrite('savedFrame.png', callBackImg)
        elif k == ord('q'):
            break

    # When everything done, release the capture
    cap.release()
    cv.destroyAllWindows()


def readIMG():
    global callBackImg
    orig = cv.imread(cv.samples.findFile("testImage1.png"))
    print(orig.shape)
    gray = myGray(orig)
    callBackImg = gray
    cv.imshow("Display window", gray)
    cv.setMouseCallback("Display window", mouseCallback)

    while True:
        k = cv.waitKey(1)
        if k == ord('s'):
            cv.imwrite('testSave.png', callBackImg)
        elif k == ord('q'):
            break
```

```
def segment(orig, lower, upper):
    (height, width, channels) = orig.shape
    ret = np.zeros((height, width, 1), dtype=np.int8)
    for i in range(height):
        for j in range(width):
            if lower[0] <= orig[i][j][0] <= upper[0] >= orig[i][j][1] >= lower[1] and upper[2] >= orig[i][j][2] >= \
                    lower[2]:
                ret[i][j] = 127
            else:
                ret[i][j] = -128
    return ret


def get_center(blue):
    (height, width, channels) = blue.shape
    sumx = 0
    sumy = 0
    count = 0
    for i in range(height):
        for j in range(width):
            if blue[i][j] > 100:
                sumx += j
                sumy += i
                count += 1
    return int(sumx / count), int(sumy / count)


def dilate(orig):
    (height, width, channels) = orig.shape
    ret = np.zeros((height - 2, width - 2, 1), dtype=np.int8)

    for i in range(1, height - 1):
        for j in range(1, width - 1):
            found = False
            for x in range(-1, 2):
                for y in range(-1, 2):
                    if orig[i + x][j + y] == 255:
                        found = True
            if found:
                ret[i - 1][j - 1] = 127
            else:
                ret[i - 1][j - 1] = -128

    return ret

def erode(orig):
    (height, width, channels) = orig.shape
    ret = np.zeros((height - 2, width - 2, 1), dtype=np.int8)

    for i in range(1, height - 1):
        for j in range(1, width - 1):
            found = False
            for x in range(-1, 2):
                for y in range(-1, 2):
                    if orig[i + x][j + y] == 0:
                        found = True
            if found:
                ret[i - 1][j - 1] = -128
            else:
                ret[i - 1][j - 1] = 127

    return ret


def laplacian(orig):
    (height, width, channels) = orig.shape
    ret = np.zeros((height - 2, width - 2, 1), dtype=np.int8)
    kernel = np.array([[0, -1, 0], [-1, 4, -1], [0, -1, 0]])

    for i in range(1, height - 1):
        for j in range(1, width - 1):
            value = -128
            for x in range(-1, 2):
                for y in range(-1, 2):
                    value = value + orig[i + x][j + y] * kernel[x + 1][y + 1]
            ret[i - 1][j - 1] = value

    return ret


def hor_sobel(orig):
    (height, width, channels) = orig.shape
    ret = np.zeros((height - 2, width - 2, 1), dtype=np.int8)
    kernel = np.array([[1, 2, 1], [0, 0, 0], [-1, -2, -1]])

    for i in range(1, height - 1):
        for j in range(1, width - 1):
            value = -128
            for x in range(-1, 2):
                for y in range(-1, 2):
                    value = value + orig[i + x][j + y] * kernel[x + 1][y + 1]
            ret[i - 1][j - 1] = value

    return ret
```

```python
def vert_sobel(orig):
    (height, width, channels) = orig.shape
    ret = np.zeros((height - 2, width - 2, 1), dtype=np.int8)
    kernel = np.array([[1, 0, -1], [2, 0, -2], [1, 0, -1]])

    for i in range(1, height - 1):
        for j in range(1, width - 1):
            value = -128
            for x in range(-1, 2):
                for y in range(-1, 2):
                    value = value + orig[i + x][j + y] * kernel[x + 1][y + 1]
            ret[i - 1][j - 1] = value

    return ret


def task1():
    global callBackImg
    orig = cv.imread(cv.samples.findFile("testImage1.png"))

    # blue = segment(orig, (203, 71, 62),(205, 73, 64))
    blue = cv.inRange(orig, (203, 71, 62), (205, 73, 64))
    blue = blue.reshape(blue.shape[0], blue.shape[1], 1)

    centerX, centerY = get_center(blue)
    callBackImg = blue
    cv.circle(callBackImg, (centerX, centerY), 5, 125, -1)

    cv.imshow("Display window", blue)
    cv.setMouseCallback("Display window", mouseCallback)
    while True:
        k = cv.waitKey(1)
        if k == ord('s'):
            cv.imwrite('testSave.png', callBackImg)
        elif k == ord('q'):
            break

def task2():
    global callBackImg
    orig = cv.imread(cv.samples.findFile("testImage1.png"))

    blue = cv.inRange(orig, (203, 71, 62), (205, 73, 64))
    blue = blue.reshape(blue.shape[0], blue.shape[1], 1)
    # blue = erode(blue)
    # blue = dilate(blue)

    kernel = np.ones((3, 3), np.uint8)
    # blue = cv.dilate(blue, kernel, iterations=1)
    blue = cv.erode(blue, kernel, iterations=1)
    blue = blue.reshape(blue.shape[0], blue.shape[1], 1)

    centerX, centerY = get_center(blue)
    callBackImg = blue
    cv.circle(callBackImg, (centerX, centerY), 5, 125, -1)

    cv.imshow("Display window", blue)
    cv.setMouseCallback("Display window", mouseCallback)
    while True:
        k = cv.waitKey(1)
        if k == ord('s'):
            cv.imwrite('testSave.png', callBackImg)
        elif k == ord('q'):
            break
```

```python
def task3():
    global callBackImg
    orig = cv.imread(cv.samples.findFile("testImage1.png"))

    blue = cv.inRange(orig, (203, 71, 62), (205, 73, 64))
    blue = blue.reshape(blue.shape[0], blue.shape[1], 1)

    kernel = np.ones((3, 3), np.uint8)
    blue = cv.erode(blue, kernel, iterations=1)
    blue = blue.reshape(blue.shape[0], blue.shape[1], 1)

    # blue = laplacian(blue)
    # blue = hor_sobel(blue)
    # blue = vert_sobel(blue)

    blue = cv.Laplacian(src=blue, dst=blue, ddepth=-1)
    blue = blue.reshape(blue.shape[0], blue.shape[1], 1)
    # blue = cv.Sobel(src=blue, dst=blue, ddepth=-1, dx=0, dy=1, ksize=3)  # Horizontal
    # blue = cv.Sobel(src=blue, dst=blue, ddepth=-1, dx=1, dy=0, ksize=3)  # Vertical

    centerX, centerY = get_center(blue)
    callBackImg = blue
    cv.circle(callBackImg, (centerX, centerY), 5, 125, -1)

    cv.imshow("Display window", blue)
    cv.setMouseCallback("Display window", mouseCallback)
    while True:
        k = cv.waitKey(1)
        if k == ord('s'):
            cv.imwrite('testSave.png', callBackImg)
        elif k == ord('q'):
            break

def task4() :
    global callBackImg
    cap = cv.VideoCapture(0)
    if not cap.isOpened():
        print("Cannot open camera")
        exit()
    while True:
        # Capture frame-by-frame
        ret, callBackImg = cap.read()
        # if frame is read correctly ret is True
        if not ret:
            print("Can't receive frame (stream end?). Exiting ...")
            break
        segmented = cv.inRange(callBackImg, (20, 30, 140), (80, 100, 255))
        segmented = segmented.reshape(segmented.shape[0], segmented.shape[1], 1)


        kernel = np.ones((3, 3), np.uint8)
        segmented = cv.erode(segmented, kernel, iterations=1)
        segmented = segmented.reshape(segmented.shape[0], segmented.shape[1], 1)

        centerX, centerY = get_center(segmented)
        callBackImg = segmented
        cv.circle(callBackImg, (centerX, centerY), 5, 125, -1)
        cv.imshow("Display window", callBackImg)
        cv.setMouseCallback("Display window", mouseCallback)

        k = cv.waitKey(1)
        if k == ord('s'):
            cv.imwrite('savedFrame.png', callBackImg)
        elif k == ord('q'):
            break

    # When everything done, release the capture
    cap.release()
    cv.destroyAllWindows()
```

```python
def task5() :
    global callBackImg
    cap = cv.VideoCapture(0)
    if not cap.isOpened():
        print("Cannot open camera")
        exit()
    while True:
        # Capture frame-by-frame
        ret, callBackImg = cap.read()
        # if frame is read correctly ret is True
        if not ret:
            print("Can't receive frame (stream end?). Exiting ...")
            break
        segmented = cv.inRange(callBackImg, (20, 30, 180), (255, 255, 255))
        segmented = segmented.reshape(segmented.shape[0], segmented.shape[1], 1)

        num_labels, labels, stats, centroids = cv.connectedComponentsWithStats(segmented, 4)

        max = -1
        max_index = 0
        for i in range(num_labels) :
            if stats[i, cv.CC_STAT_AREA] > max :
                max = stats[i, cv.CC_STAT_AREA]
                max_index = i

        cX, cY = centroids[max_index]
        cX = int(cX)
        cY = int(cY)
        print(cX, cY)

        callBackImg = segmented
        cv.circle(callBackImg, (cX, cY), 5, 125, -1)
        cv.imshow("Display window", callBackImg)
        cv.setMouseCallback("Display window", mouseCallback)

        k = cv.waitKey(1)
        if k == ord('s'):
            cv.imwrite('savedFrame.png', callBackImg)
        elif k == ord('q'):
            break


if __name__ == '__main__':
    task5()
    #readVid()
```