

ROB-UY 3203: Robot Vision Group Project Phase 1 Report

For phase 1 of the project, we were given two sets of 2D images taken by the same camera, in folders named “A” and “B”. Based on what was taught during week 2 to week 5, we were tasked with calculating the different quantities based on these two sets of images, given that the side length of the AprilTag in folder “B” is 150.1 mm as shown in Figure 1. The quantities calculated were the length (L), width (W), and height of the box (H). In addition to this, the diameter (d) and the height of the soldering wire scroll (h) was also calculated. This report details and explains the process of the code used to determine these results. Also, this report will critique any difficulties encountered and potential issues that could be resolved to improve the quality of the code.

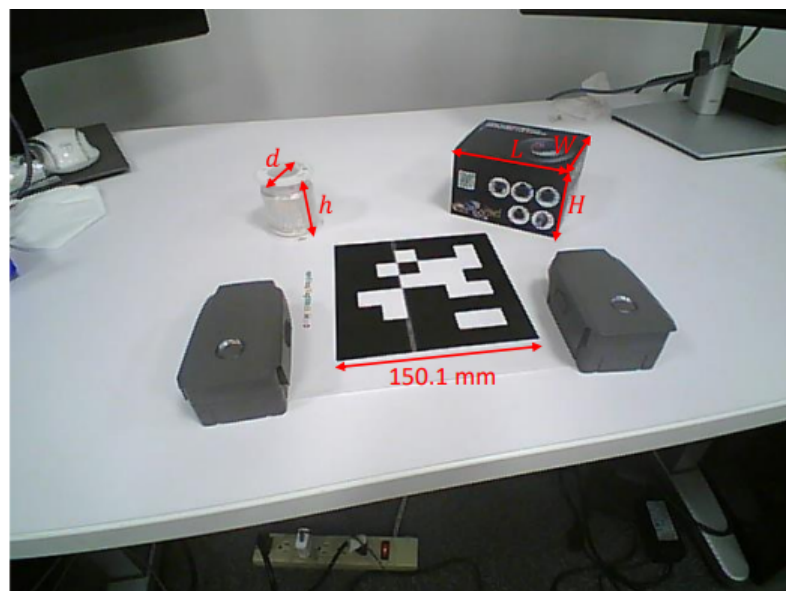
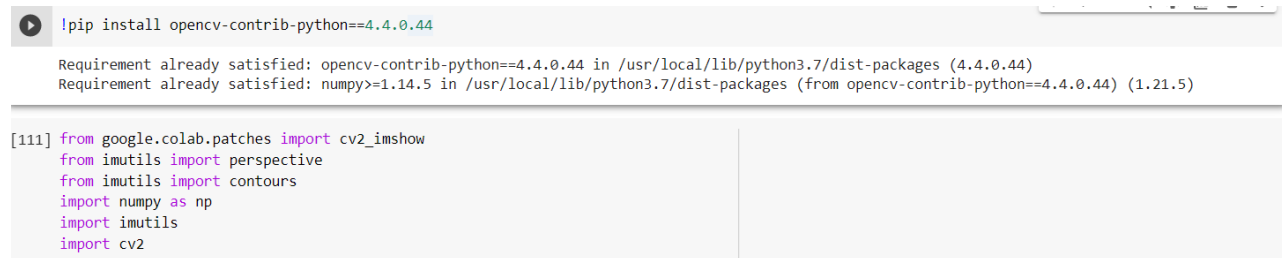


Figure 1: Example image with given information and quantities that need to be calculated

To begin, I started my program by verifying the opencv version before using the ArUco module and importing any relevant packages needed for the code on Google Colabs as seen in Figure 2. Due to the fact that I was using Google Colab, it was necessary to include “from google.colab.patches import cv2_imshow” in order to display images.



```
!pip install opencv-contrib-python==4.4.0.44

Requirement already satisfied: opencv-contrib-python==4.4.0.44 in /usr/local/lib/python3.7/dist-packages (4.4.0.44)
Requirement already satisfied: numpy>=1.14.5 in /usr/local/lib/python3.7/dist-packages (from opencv-contrib-python==4.4.0.44) (1.21.5)

[111] from google.colab.patches import cv2_imshow
      from imutils import perspective
      from imutils import contours
      import numpy as np
      import imutils
      import cv2
```

Figure 2: Part 1 of Code

I began with performing camera calibration using Zhang’s method and all the images provided in folder A. The code was adjusted for a 5x7 chessboard as seen in Figure 3A and uses the glob package to collect all the photos provided for calibration. The word test was added to front of each image’s name before utilizing them, in order to prevent overlap with the names of the images in Folder B. The for loop finds the corners(intersections) on the photos of checkerboard by using OpenCV's IO to read one image at a time based on files grabbed by glob package and convert them to grayscale. Then it feeds the image to cv2.findChessboardCorners. If cv2.findChessboardCorners does find corners, we use cv2.cornerSubPix to further improve the accuracy of such corners. Afterwards, it appends the coordinates of “ideal points” to objpoints, and the corner coordinates to imgpoints. Each of the corners we found on the photo using cv2.drawChessboardCorners is visualized until all the photos are used.

```

import numpy as np
import cv2
import matplotlib.pyplot as plt
import glob
from google.colab.patches import cv2_imshow

objp = np.zeros((5*7,3), np.float32)
objp[:, :2] = np.mgrid[0:7,0:5].T.reshape(-1,2)

plt.scatter(objp[:, 0], objp[:, 1]);

# Arrays to store object points and image points from all the images.
objpoints = [] # 3d point in real world space
imgpoints = [] # 2d points in image plane.

# termination criteria
criteria = (cv2.TERM_CRITERIA_EPS + cv2.TERM_CRITERIA_MAX_ITER, 30, 0.001)

# use images captured by camera
images = glob.glob('test*.jpg')

for fname in images:
    img = cv2.imread(fname)

    # Resize to make sure the image does not contain too much pixels for OpenCV to find patterns

    img = cv2.resize(img, (640,480))
    gray = cv2.cvtColor(img,cv2.COLOR_BGR2GRAY)
    # Find the chess board corners, we use 8*6 grid
    ret, corners = cv2.findChessboardCorners(gray, (7,5),None)
    # If found, add object points, image points (after refining them)
    if ret == True:
        objpoints.append(objp)
        # cv2.cornerSubPix() increases the accuracy of the corner coordinates
        corners2 = cv2.cornerSubPix(gray,corners,(11,11),(-1,-1),criteria)
        imgpoints.append(corners2)
        # Draw and display the corners
        img = cv2.drawChessboardCorners(img, (7,5), corners2,ret)
        cv2_imshow(img)

```

Figure 3A: Camera Calibration Code

The figure (3B) below shows the result of the calibrated camera by displaying the camera matrix and distortion coefficients found through the images.

```

[ ] # cv2.calibrateCamera() returns the camera matrix, distortion coefficients, rotation vectors, and translation vectors
ret, mtx, dist, rvecs, tvecs = cv2.calibrateCamera(objpoints, imgpoints, gray.shape[:-1],None,None)

[ ] # print the values
print(mtx)
print(dist)
print(rvecs)
print(tvecs)

[[496.83944364  0.          439.59806822]
 [ 0.          495.76684561 306.58791208]
 [ 0.           0.           1.           ]]
[[ 0.05805106  0.20132603  0.00184402  0.00191373 -0.71333197]]

```

Figure 3B: Camera Calibration Result Code

With all the necessary preparations in order, I downloaded all the images given to me from Folder “B” and used one of the images. The code seen in Figure 4, edits the image, in this case “6.jpg”, and creates a blue box on the objects we want to measure. This is done by providing coordinates to draw a box on the objects. This new image is seen in Figure 5 and is necessary as the code then sets a range of blue which filters everything else besides the box we want to measure. All this helps create a “mask” of the image which is displayed in the final line of code seen in Figure 4, “cv2_imshow(mask)”.

```
img = "6.jpg"

# Read image...
image = cv2.imread(img)

# Length and Height of Box
cv2.line(image, (298,77),(393,91),(255,0,0),3)
cv2.line(image, (298,77),(298,129),(255,0,0),3)
cv2.line(image, (298,129),(383,138),(255,0,0),3)
cv2.line(image, (393,91),(383,138),(255,0,0),3)

# Length and Width of Box
cv2.line(image, (300,68),(309,40),(255,0,0),3)
cv2.line(image, (300,68),(393,82),(255,0,0),3)
cv2.line(image, (309,40),(394,48),(255,0,0),3)
cv2.line(image, (394,48),(393,82),(255,0,0),3)

# Height and Diameter of Scroll
cv2.line(image, (120,128),(165,115),(255,0,0),3)
cv2.line(image, (120,128),(135,175),(255,0,0),3)
cv2.line(image, (165,115),(180,162),(255,0,0),3)
cv2.line(image, (180,162),(135,175),(255,0,0),3)

hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)

# define range of blue color in HSV
lower_blue = np.array([110,35,35])
upper_blue = np.array([130,255,255])

# Threshold the HSV image to get only blueish colors
mask = cv2.inRange(hsv, lower_blue, upper_blue)

cv2_imshow(mask)
```

Figure 4: Reading and Gray-scaling image

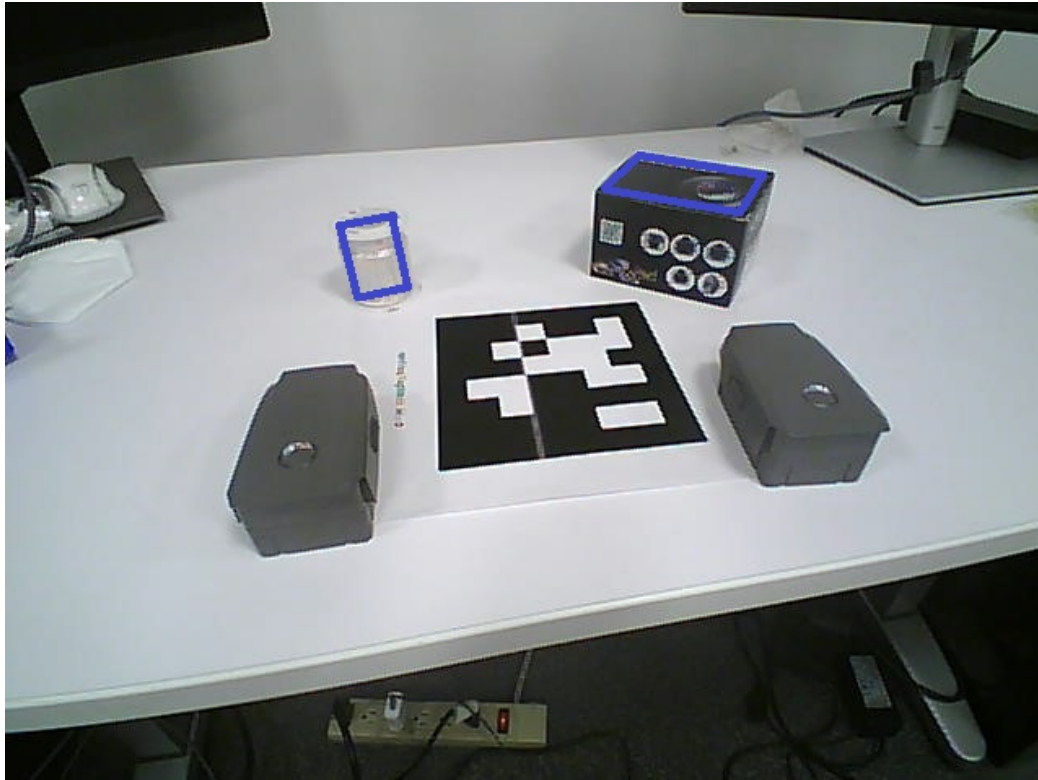


Figure 5: Edited image

The following lines of code in Figure 6 find contours within the image as the `findContours` method identifies what would be considered contours of various whole objects in the grayscale image. Among these contours, the code will likely identify the box and soldering wire scroll among them. In order to not have too many excess contours, some contours are removed based on the contour area as seen in the final line of code in Figure 6.

```
# Find contours in image
cnts = cv2.findContours(edged.copy(), cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
cnts = imutils.grab_contours(cnts)

# Remove contours extra contours
cnts = [x for x in cnts if cv2.contourArea(x) > 200]
```

Figure 6: Code for Contours in Image

The next step is to use a reference within the image in order to calculate the measurements of all the contours the code has identified as seen in Figure 7. In the image, there is an aruco marker with a known side of 150.1 mm. The following lines of code will identify the location of the aruco and highlight it for the user. It does this by detecting the 4 corners of the aruco marker and with the coordinates, it can now be used to determine the ratio of pixels to mm in order to measure the lines of the other contours.

```
# Reference object dimensions
apriltag_img = cv2.imread(img)

# show the apriltag use matplotlib. If you use your local environment, you can use cv2.imshow() function as you did before
imshow(apriltag_img[:, :, :-1])

# load Tag36h11 in aruco dictionary
ARUCO_DICT = {"DICT_APRILTAG_36h11": cv2.aruco.DICT_APRILTAG_36h11}
arucoDict = cv2.aruco.Dictionary_get(ARUCO_DICT["DICT_APRILTAG_36h11"])

arucoParams = cv2.aruco.DetectorParameters_create()
(image_points, ids, rejected) = cv2.aruco.detectMarkers(apriltag_img, arucoDict,
    parameters=arucoParams)

# by default, the four corners are (x,y) coordinates in the image, with pixel as unit.
# The order of the four corners are top-left, top-right, bottom-right, and bottom-left

# Ensure that at least one tag is detected, then prepare the corners for drawing:
if len(image_points) > 0:
    # flatten the ArUco IDs list
    ids = ids.flatten()
    # loop over the detected ArUco corners
    for (markerCorner, markerID) in zip(image_points, ids):
        # extract the marker corners
        image_points = markerCorner.reshape((4, 2))
        (topLeft, topRight, bottomRight, bottomLeft) = image_points
        # convert each of the (x, y)-coordinate pairs to integers
        topRight = (int(topRight[0]), int(topRight[1]))
        bottomRight = (int(bottomRight[0]), int(bottomRight[1]))
        bottomLeft = (int(bottomLeft[0]), int(bottomLeft[1]))
        topLeft = (int(topLeft[0]), int(topLeft[1]))

    # draw the bounding box of the ArUco detection
    cv2.line(apriltag_img, topLeft, topRight, (0, 255, 0), 4)
    cv2.line(apriltag_img, topRight, bottomRight, (0, 255, 0), 4)
    cv2.line(apriltag_img, bottomRight, bottomLeft, (0, 255, 0), 4)
    cv2.line(apriltag_img, bottomLeft, topLeft, (0, 255, 0), 4)
```

Figure 7: Use Aurco Marker as Reference Image

```

world_points = np.array([[0., 1., 0.], [1., 1., 0.], [1., 0., 0.], [0., 0., 0.]]) # ensure here we use float64 type, instead of int.
K = np.array([[496.83944364, 0.0, 439.59806822],
              [0.0, 495.76684561, 306.58791208],
              [0.0, 0.0, 1.0]])

distorC = np.array([0.05805106, 0.20132603, 0.00184402, 0.00191373, -0.71333197])
_, rot, trans = cv2.solvePnP(world_points, image_points, K, distorC)

print(image_points)
draw_points_3d = [np.array([0., 1., 2.]), np.array([1., 1., 2.]), np.array([1., 0., 2.]), np.array([0., 0., 2.])]
draw_points_2d = []
for i in range(len(draw_points_3d)):
    tmp = cv2.projectPoints(draw_points_3d[i], rot, trans, K, distorC)[0][0][0]
    draw_points_2d.append(tmp)

int_corners = np.int0(image_points)
cv2.polylines(image, [int_corners], True, (0, 255, 0), 4)

distance_x = (image_points[0][0] - image_points[3][0])**2
distance_y = (image_points[0][1] - image_points[3][1])**2
distance_formula = math.sqrt(distance_x + distance_y)
pixel_ratio = (distance_formula) / (150.1) # ratio conversion (pixel to mm)

```

Figure 8: Pixel to mm calculations

Since the aruco marker has a known side of 150.1 mm we can use this information by comparing that value to its length in pixels. Using the camera matrix, distortion coefficients, and coordinates of the 4 corners of the Aruco maker (image_points) we can determine the length of the known side of the marker in pixels as seen in Figure 8. Taking the coordinates of the two corners and applying the distance formula provides us the length in pixels. By dividing that value with 150.1 cm, we now have the ratio of pixels to mm in any image.

The final step of the code as seen in Figure 9, involves using a for loop to draw a box for all the contours. As the code loops through each contour, it is drawing what it determines to be an appropriate box for the contour and writing in the image, the box's width and height in millimeters. In the end, the code will display the newly edited image with the values of each contour's width and height.

```

# Draw remaining contours
for cnt in cnts: # for loop of contours...
    box = cv2.minAreaRect(cnt)
    (x,y), (w,h), angle = box

    obj_w = w / pixel_ratio # width
    obj_h = h / pixel_ratio # height

    box = cv2.boxPoints(box)
    box = np.int0(box)

    cv2.polyline(image, [box], True, (255, 0, 0), 2)
    cv2.putText(image, "W = {:.1f}mm".format(round(obj_w,1)), (int(x - 150), int(y - 25)), cv2.FONT_HERSHEY_SIMPLEX, .5, (255, 255, 0), 2)
    cv2.putText(image, "H = {:.1f}mm".format(round(obj_h,1)), (int(x - 150), int(y + 25)), cv2.FONT_HERSHEY_SIMPLEX, .5, (255, 255, 0), 2)

cv2.imshow(image)

```

Figure 9: For loop of Contours

Figure 10 provides the result image of the code for the image “6.jpg”. However, several issues were encountered while developing the code for this project. One particular issue is that the code has difficulty identifying some of the items without the edited blue box. Specifically the soldering wired scroll blends into the background of the table due to its similar coloring. Originally the code would convert the whole image to grayscale and didn’t manually draw a box in the image. As a result, the code often has difficulty drawing a box for the scroll in order to determine its values. This can be said for the box as well occasionally, as the code would often draw a single rectangle for the box, making it difficult to determine length, width and height of it. Both issues can be resolved with some manual adjustments to some of the values used in the code but it would be better if the code could handle it automatically. In addition, the code identifies one image at a time based on a manual input. Although this was done intentionally due to the fact that the coordinates to draw the blue box differs between images, it also means the code could be more efficient. Overall, the code could be better and more accurate when determining contours and the value of each box.

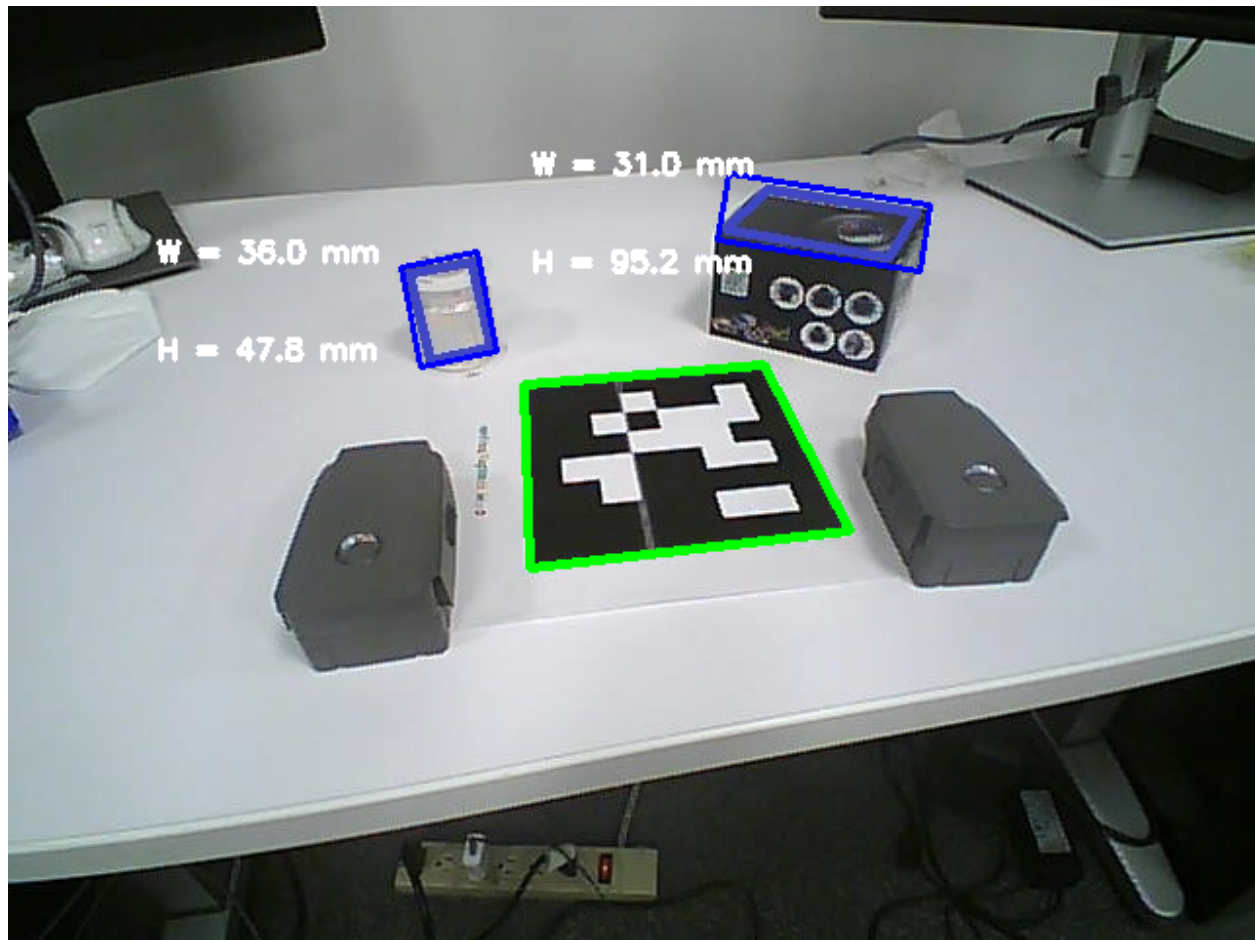


Figure 10: Results

After using the code on more images, I have determined that the measurements for the box to approximately be length = 95 mm, width = 40 mm, and height = 50 mm. For the soldering wire scroll, I found that the diameter was approximately 45 mm and the height to be 50 mm.