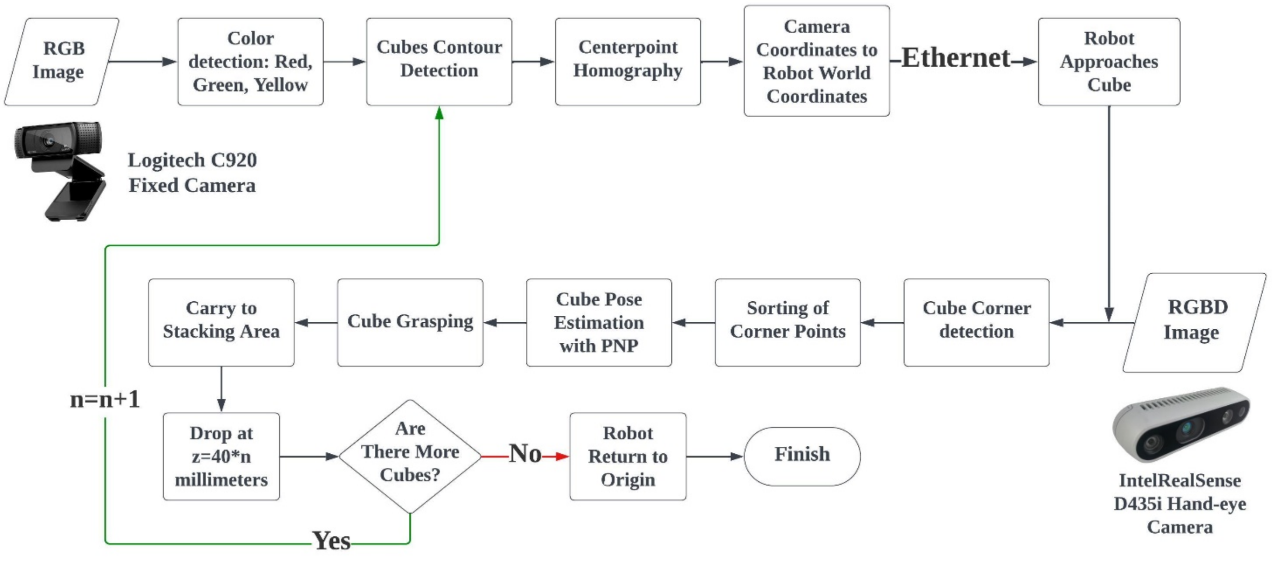
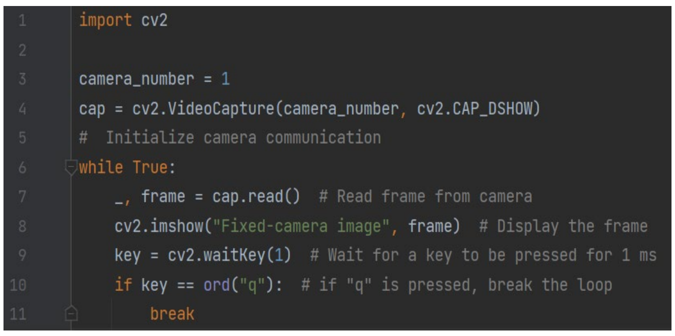


Figure 1. *Flow chart of the system.*

The figure above displays the process to autonomously control the robot by means of visual-assistance from the cameras. A Python code is used from a computer to process the camera images, perform the computation of matrix multiplications and send the coordinates for the robot to move to.

Visit the following website to access the Python functions used for each stage.

The first step of the flowchart is to get an RGB image from the fixed camera to look at the workplace. The **Test Camera.py** code in the Test folder can be ran to get the image from the fixed camera.



```python
import cv2

camera_number = 1
cap = cv2.VideoCapture(camera_number, cv2.CAP_DSHOW)
#  Initialize camera communication
while True:
    _, frame = cap.read()  # Read frame from camera
    cv2.imshow("Fixed-camera image", frame)  # Display the frame
    key = cv2.waitKey(1)  # Wait for a key to be pressed for 1 ms
    if key == ord("q"):  # if "q" is pressed, break the loop
        break
```

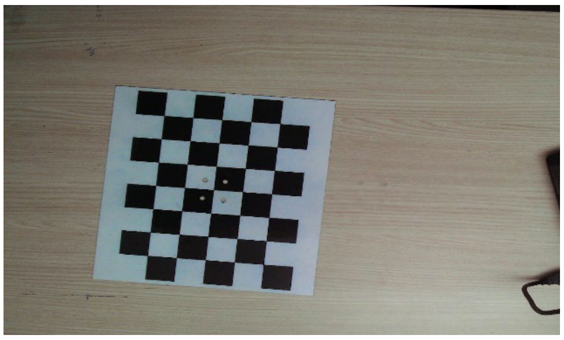


Figure 2. *Test Camera code & Resulting frame.*

Once the image is acquired, it is transformed from RGB to HSV using the built-in function cvtColor() in Python OpenCV. The **Test Color Detection.py** code can be used to test the detection of a yellow color in the workspace. Later on in the text, a simple way to find the lower and upper values for each color will be introduced.

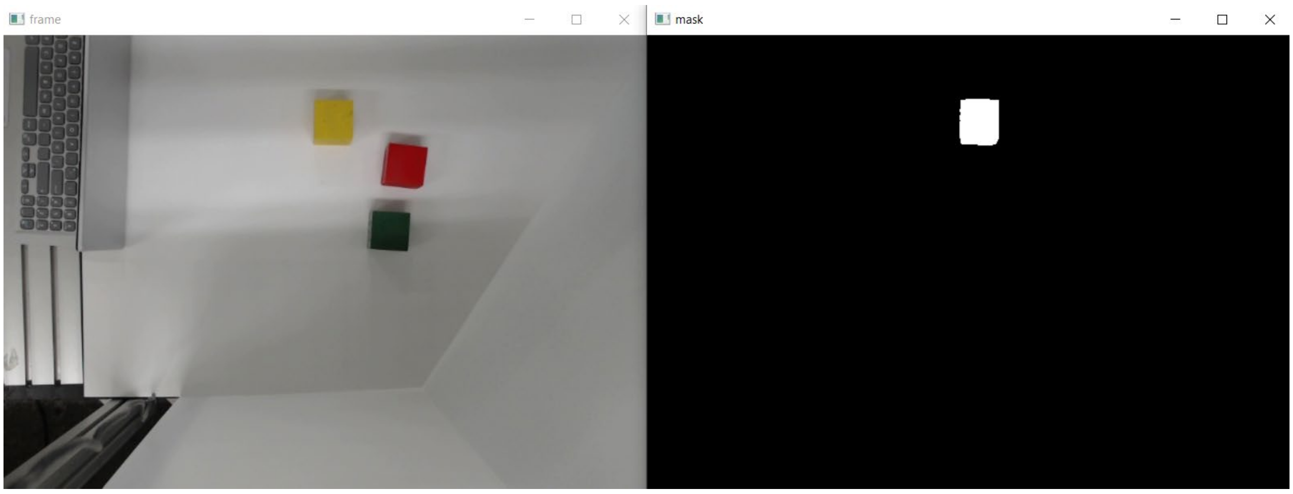


Figure 3. *Color detection from the test.*

The mask has been obtained; it can be used to get the contour of the yellow block. This can be tested in the **Test Contour Detection.py**. A result from this code can be seen in the figure below.

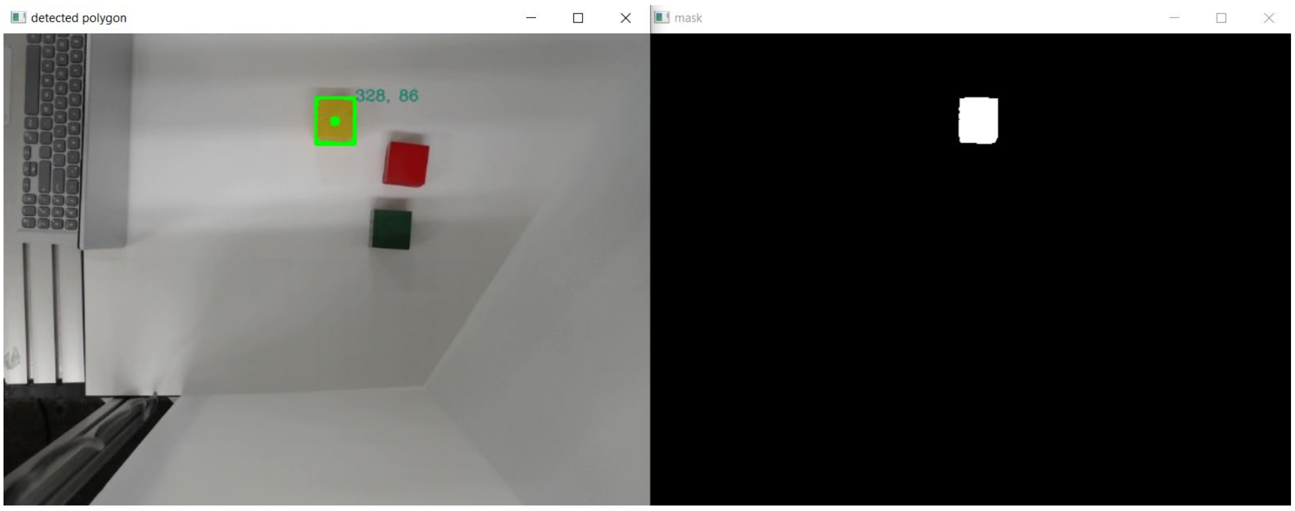


Figure 4. *Detecting contour & center point with the mask.*

The resulting pixel coordinates in the example of the center point are (328, 86), given that the height of the camera to the workplace is known, the **pinhole camera model** can be used to convert pixel coordinates to real life coordinates by

$$z_c \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = K \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \qquad (1)$$

Where (x, y, z) represent the coordinates in real camera coordinates, measured in millimeters, and (u, v) are the pixel point coordinates measured in pixels. Thus, pixel coordinates and real-life camera coordinates of the target are related by a *K* matrix called the **Camera Matrix** or **Intrinsic**

Camera Matrix. In other words, the *K* matrix projects real-life points into pixel coordinates in the camera. For the purposes of this project, the inverse process is usually required, real-life camera coordinates are to be found from the pixel coordinates of the camera.

$$z_c K^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} x_c \\ y_c \\ z_c \end{bmatrix} \tag{2}$$

The *K* matrix is found from a process called **Camera Calibration**, this matrix is constant and one only needs to perform the process one time per camera, although redoing the process is required after the camera has been subjected to large impacts and other harsh environments. Zhang's method is used in this work to find the *K* matrix of both cameras used.

Camera calibration save images.py and **Camera calibration calculation.py** are used to get the values inside the *K* matrix.

The first code saves images when the user presses "s" if the calibration board (a chess board) is detected. After enough images (about 20) have been taken with the calibration board in different angles and positions throughout the camera image, the second code is used to process the detected chess board poses and calculate the *K* matrix of the used camera using [1].

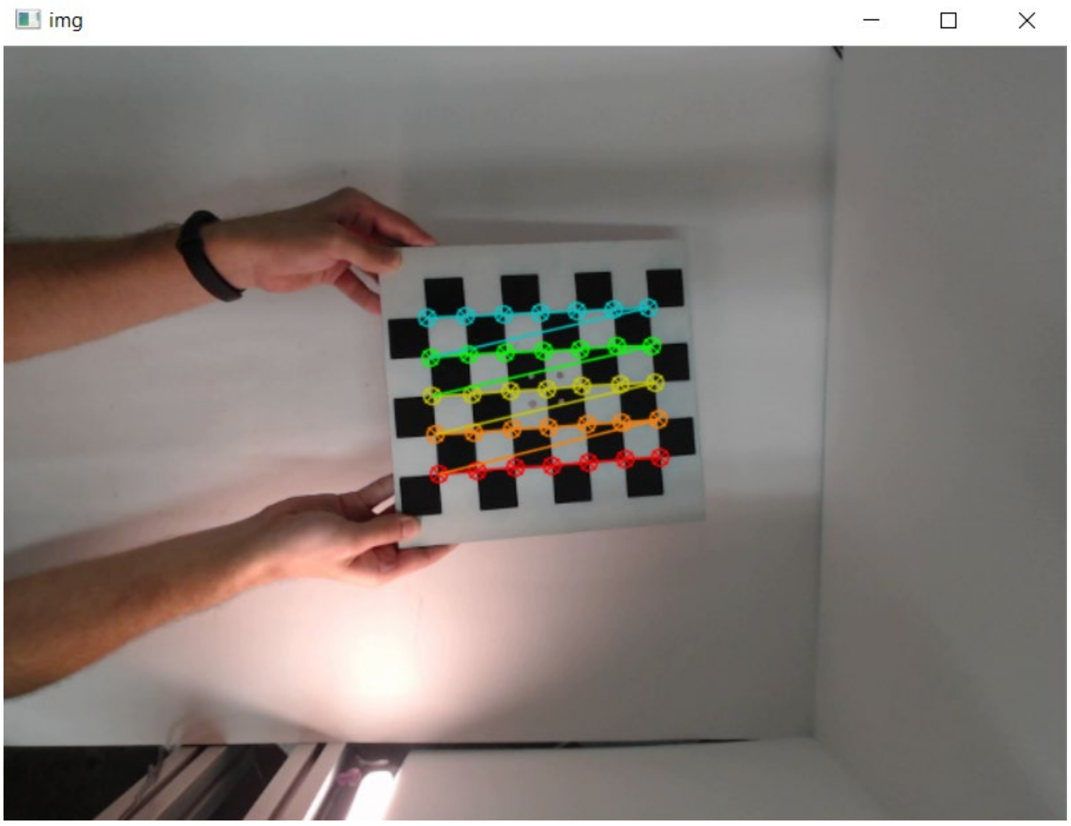


Figure 5. *Detected corner points of the chess board.*

A successful run of the second code will generate an .npz file called MultiMatrix_fixed_640_480 (by default), this file contains the camera matrix *K* and the distortion coefficients of the camera. The **main.py** code provides examples on how to retrieve this data from the .npz file. The exact same process can be used with the hand camera to retrieve its camera matrix.

After retrieving the *K* matrix, the pixel coordinates of the target can be transformed to camera coordinates by (2). Given that the distance and orientation from the camera axes to the robot world axes are known, the coordinates from the robot to the target in robot coordinates can be calculated by homogeneous transformations using

$$P^R = H_C^R P^C \tag{3}$$

Where P^R are (x, y, z) point coordinates in robot axes. P^C are (x, y, z) point coordinates in camera axes and H_C^R is the homogeneous transformation from camera axes to robot axes. If the previous paragraph was difficult to read, please refer to [2] to learn about homogeneous transformation matrices.

Following the previous operation, the (x, y, z) coordinates of the center point in robot coordinates can be sent to the robot through ethernet connection using TCP communication. The manipulator end effector will move towards the target.

Once the machine approaches the cube, the fixed camera is no longer needed, the hand camera is now used to retrieve images.

Color detection is again used with the movable camera retrieved images to create a mask of the cube, the mask is further processed to remove the saw teeth on the edges. The code **Test Corner Detection.py** can be experimented with to see how the corners can be detected processing the mask using the work in [3].

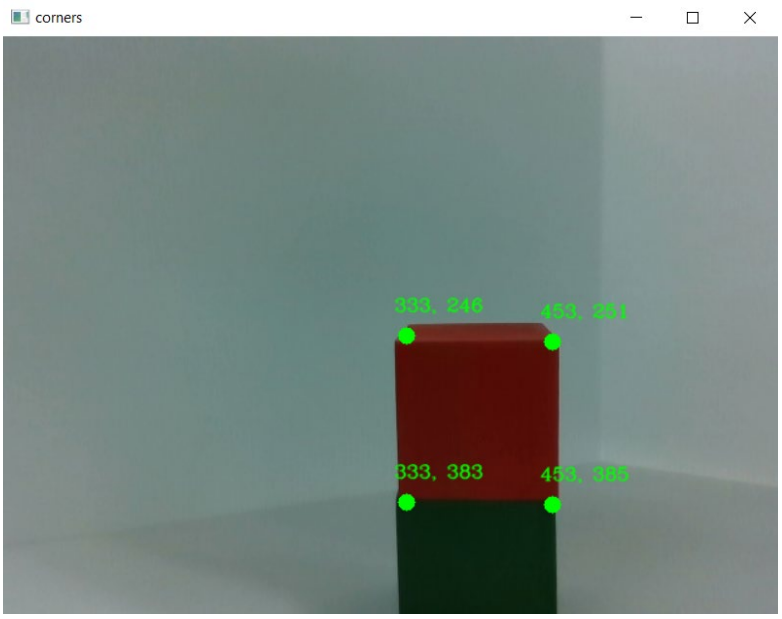


Figure 6. *Detected corners from the image given in the example code.*

Having the corner points, the pose of the target in camera coordinates can be approximated by utilizing **Perspective-n-Point** (PnP), which is the problem of estimating the pose of a target given a set of *n* real world 3D points and their corresponding projected 2D image points.

The 3D points shown on *Figure 6* are (0, 0, 0), (50, 0, 0), (50, 50, 0) and (0, 50, 0) in mm.

The 2D points values are shown in the figure, the arrangement is clockwise starting from (333, 246)

Using the solveP3P function from the OpenCV library, the 2D points and 3D points of the corners, camera matrix *K* and distortion coefficients from the camera are utilized together to get the homogeneous transformation from the target axes to camera axes. This can be again converted to a transformation from target to robot coordinates using homogeneous transformation matrices from the robot origin to its end-effector axes and from the end-effector axes to the camera axes. Mathematically, this is expressed as

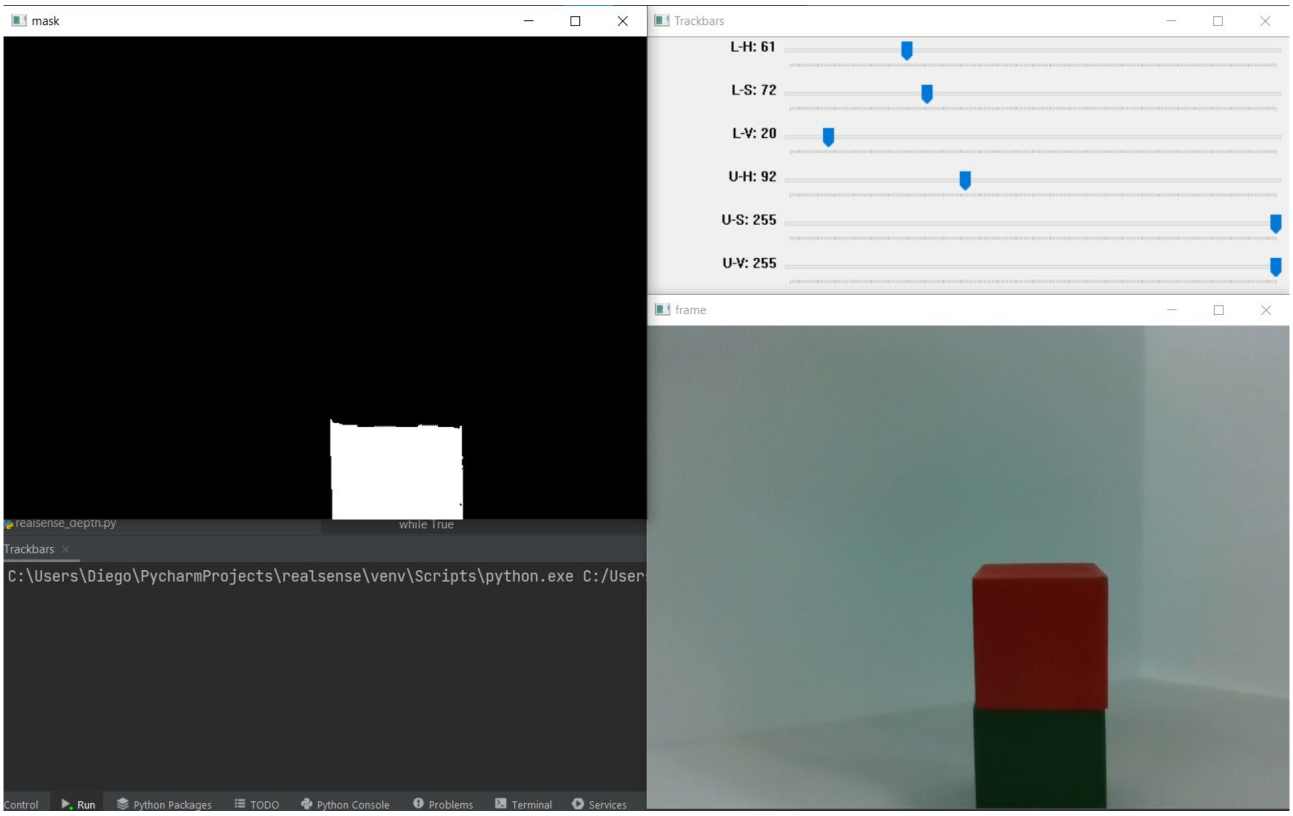
$$H_T^R = H_G^R H_C^G H_T^C \tag{4}$$

The matrix H_G^R is known from the robot end-effector's position. H_T^C is acquired through the PnP solution. H_C^G , the transformation from camera coordinates to gripper (end-effector) coordinates is still unknown, and it needs to be acquired through the performance of a **hand-eye calibration**, which allows to relate camera position with gripper position. The **Hand-eye-calib** folder explains the process to find this matrix [4]. Any point on target coordinates can be transformed to robot coordinates using the homogeneous transformation from target coordinates to robot coordinates by (3). Moreover, the gripper can be oriented parallel to the cube axes by making use of the rotation matrix inside H_T^R to find its roll-pitch-yaw angles and commanding the manipulator to move its Rx, Ry, Rz angles to the calculated values. With this knowledge, we can now grasp the cubes by positioning the manipulator above the center, rotating the gripper by the roll-pitch-yaw angles, moving the gripper down to clamp the target, and closing the gripper.

To control ROBOTIQ gripper, the function gripMove() in **Funciones.py** can be used.

The system was created by assembling all of these functions and tests together into one single code. The result of this assembly can be accessed in the **main.py** program.

For ease of finding ranges for the creation of masks, a function in **Funciones.py** is used. create_trackbars() can be used to display tracks for each of the HSV lower and upper values. By using the function trackbar(frame, "track"), a mask is created using the values displayed on the trackbars created. **Test trackbars.py** can be used to try this.



L-H: 61
L-S: 72
L-V: 20
U-H: 92
U-S: 255
U-V: 255

```
C:\Users\Diego\PycharmProjects\realsense\venv\Scripts\python.exe C:/User
```

Figure 7. *Real time mask creation using trackbars.*

Bibliography

[1] W. Burger, "Zhang's Camera Calibration Algorithm: In-Depth Tutorial and Implementation," University of Applied Sciences Upper Austria, School of Informatics, Communications and Media, Hagenberg, Austria, 2016.

[2] M. W. Spong, S. Hutchinson and M. Vidyasagar, Robot Modeling and Control, Wiley, 2020.

[3] J. Shi and Tomasi, "Good features to track," in *IEEE Conference on Computer Vision and Pattern Recognition*, Seattle, WA, USA, 1994.

[4] R. Y. Tsai and R. K. Lenz, "A new technique for fully autonomous and efficient 3d robotics hand/eye calibration," *IEEE Transactions on Robotics and Automation,* vol. 5, no. 3, pp. 345-358, 1989.