```python
#################################################################################################
### IMPAGE PROCESSING TO DETECT THE PLANKS THAT MUST BE SORTED AND COVERT THEIR COORDINATES ###
#################################################################################################

import cv2 # pip install opencv-python
import numpy as np
from copy import deepcopy

#############################
### RESOURCES AND CREDITS ###
#############################

# https://www.youtube.com/watch?v=Z846tkgl9-U
# https://www.youtube.com/watch?v=oXlwWbU8l2o

########################
### CAMERA FUNCTIONS ###
########################

def openCapture(camera_number_on_pc) :
    """
    This function opens and returns a camera *capture*. It represents the camera being active, and allows to
recover frames.
    Once a program is done recovering frames from the camera, the capture *must* be closed with the following
lines
    (or with the closeCapture function below) :

    capture.release()
    cv2.destroyAllWindows()
    """

    print("Opening capture ...")
    capture = cv2.VideoCapture(camera_number_on_pc)
    print("capture is opened:",capture.isOpened())

    while not capture.isOpened() :
        capture = cv2.VideoCapture(camera_number_on_pc)
        print("capture is opened:",capture.isOpened())

    print("-----")
    return capture



def closeCapture(capture) :
    capture.release()
    cv2.destroyAllWindows()
```

```python
def undistortImage(img, mtx, newcameramtx, dist) :
    h,w = img.shape[:2]

    # Method 1 to undistort the image
    undistorted = cv2.undistort(img, mtx, dist, None, newcameramtx)

    # Method 2 to undistort the image
    mapx,mapy=cv2.initUndistortRectifyMap(mtx,dist,None,newcameramtx,(w,h),5)
    undistorted = cv2.remap(img,mapx,mapy,cv2.INTER_LINEAR)

    return undistorted



mtx = np.array([[426, 0, 332], [0, 426, 198], [0, 0, 1]])
distorsion_coefficients = np.array([[-0.266, 0.07, -0.006, 0.0008, 0.0058]])
# See section II.1.c of the report



##################################
### IMAGE PROCESSING FUNCTIONS ###
##################################

def chopChop(img, additional_crop) :
    nb_lines, nb_columns, nb_colors = img.shape
    crop = int((nb_columns - nb_lines)/2)
    return img[additional_crop:nb_lines-additional_crop-1 , crop+additional_crop:nb_columns-crop-
additional_crop-1]



def colorQuantization(img, k) : # To reduce the number of colors in an image
    return (np.array(img)//k)*k



def cannyEdgeDetector(img, threshold1, threshold2, color_divider = 8, thickness = 5) :

    # Color reduction
    reduced = colorQuantization(img, color_divider)

    # Blur
    blur = cv2.GaussianBlur(reduced, (7, 7), cv2.BORDER_DEFAULT)

    # Convert to grayscale :
    # gray = cv2.cvtColor(blur, cv2.COLOR_BGR2GRAY)

    # Canny edge detector
    canny = cv2.Canny(blur, threshold1, threshold2)
```

```
    # Dilate the edges to make them thicker
    canny = cv2.dilate(canny, np.ones((thickness,thickness)), iterations=1)

    return canny



def cleanContours(edges_img, edges_img_canvas, contours, min_area, max_area) :
    # edges_img is an image of edges, like the one returned by cannyEdgeDetector().
    # contours is a list of its contours as detected by cv2, using cv2.findContours().
    # This function will sort through these contours and eliminate the unwanted ones
    # by erasing them directly on edges_img.

    # As explained in section II.2.b of the report, we will keep only the contours with an area
    # contained within min_area and max_area and with 4 corners.

    found_noisy_contour = False

    for contour in contours :
        # Some of the contours might be noise. One way to filter them out is to compute their area.
        area = cv2.contourArea(contour)

        if area > max_area or min_area > area :
            found_noisy_contour = True
            cv2.drawContours(edges_img, contour, -1, color = (0, 0, 0), thickness = 5) # erase noisy contour
            cv2.drawContours(edges_img_canvas, contour, -1, color = (128, 128, 128), thickness = 5) # color
noisy contour in gray on the canvas

        else :
            # Each contour is a huge array of points. Fortunately, cv2 can approximate its shape :
            perimeter = cv2.arcLength(contour, closed = True)
            epsilon = 0.03*perimeter
            approximate_shape_corners = cv2.approxPolyDP(contour, epsilon, closed = True)

            # We wish to detect 2D rectangular wooden planks, so we will only process trapezes (4 edges) :
            if len(approximate_shape_corners) != 4 :
                found_noisy_contour = True
                cv2.drawContours(edges_img, contour, -1, color = (0, 0, 0), thickness = 5)
                cv2.drawContours(edges_img_canvas, contour, -1, color = (128, 128, 128), thickness = 5)

    return found_noisy_contour



def findCentersGravity(img, edges_img, edges_img_canvas, min_area, max_area) :
    # The following lines will "extract the contours from the image of the edges".
    # This might seem odd at first glance, why not call this alone and skip detecting the edges beforehand ?
    # That's merely because the canny edge detector is easier to calibrate.
    # All in all, this line is simply used here to separate really fast the closed shapes drawn by the edges.
```

```
    contours, hierarchy = cv2.findContours(edges_img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
    noisy_contour_present = cleanContours(edges_img, edges_img_canvas, contours, min_area, max_area)

    while noisy_contour_present :
        contours, hierarchy = cv2.findContours(edges_img, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
        noisy_contour_present = cleanContours(edges_img, edges_img_canvas, contours, min_area, max_area)


    planks_list = []

    for contour in contours :

        # Each contour is a huge array of points. Fortunately, cv2 can approximate its shape :
        perimeter = cv2.arcLength(contour, closed = True)
        epsilon = 0.03*perimeter
        approximate_shape_corners = cv2.approxPolyDP(contour, epsilon, closed = True)
        # The variable approximate_shape_corners now contains a very crucial information :
        # The coordinates of each approximated corner
        # (as in, the lines and the columns of the matrix where they are located).
        # This allows the computation of their center of gravity as well as a change of coordinates
        # to make them usable by the robot.

        # Let us compute the coordinates of the center of gravity
        sum_column = 0
        sum_line = 0
        points_list = [] #This will store a reordered approximate_shape_corners with the center of gravity as
a bonus

        for corner in approximate_shape_corners :
            corner_column = corner[0][0]
            corner_line = corner[0][1]
            sum_column += corner_column
            sum_line += corner_line
            points_list.append((corner_line, corner_column))

        gravity_column = int(sum_column/4)
        gravity_line = int(sum_line/4)
        points_list = [(gravity_line, gravity_column)] + points_list

        not_the_robot = img[gravity_line][gravity_column][0] > 100 or img[gravity_line][gravity_column][1] >
100 or img[gravity_line][gravity_column][2] > 100
        if not_the_robot :
            planks_list.append(points_list)

    return planks_list



###################################
### PLANK PROCESSING FUNCTIONS ###
```

```
###################################

def isPlankMisaligned(robot_line, robot_column, plank) :
    # plank is a list of 5 2D points : the center of gravity and the 4 corners of a wooden plank.
    # this function will determine whether this plank is properly aligned to be seized by the robot,
    # and will do so by computing a scalar product.
    min_cos_beta = 0.67 + 0.03 # See report, section II.3.b

    x_o, y_o = robot_line, robot_column
    x_g, y_g = plank[0] # center of gravity
    x_a, y_a = plank[1] # first corner, called A
    x_b, y_b = plank[2] # next corner counterclockwise, called B
    x_c, y_c = plank[3] # second next corner counterclockwise, called C

    scalar_product_og_ab = (x_g - x_o)*(x_b - x_a) + (y_g - y_o)*(y_b - y_a)
    scalar_product_og_bc = (x_g - x_o)*(x_c - x_b) + (y_g - y_o)*(y_c - y_b)

    norm_og = ((x_g - x_o)**2 + (y_g - y_o)**2)**(1/2)

    norm_ab = ((x_b - x_a)**2 + (y_b - y_a)**2)**(1/2)
    norm_bc = ((x_c - x_b)**2 + (y_c - y_b)**2)**(1/2)

    if norm_ab > norm_bc : # if AB is the longest edge of the plank
        cos_beta = abs(scalar_product_og_ab/(norm_og*norm_ab))
    else : # if BC is the longest edge of the plank :
        cos_beta = abs(scalar_product_og_bc/(norm_og*norm_bc))

    return cos_beta <= min_cos_beta



def euclidianNorm2D(u) :
    return (u[0]**2 + u[1]**2)**(1/2)



def is_among_planks(plank_0, planks_list, epsilon) :
    # Since the camera is perturbed by a lot of noise, we will have to take several pictures in order to be
sure
    # that we detected all the planks.
    # However, this means that we will certainly detect the same plank twice or more.
    # The aim of this function is to tell whether plank_0 was already detected or not.

    among_planks = False
    g0x, g0y = plank_0[0] # center of gravity
    a0x, a0y = plank_0[1] # point A
    b0x, b0y = plank_0[2] # point B, next to A counterclockwise
    n = len(plank_0) # = 5

    for plank in planks_list :
```

```
        same_plank = False

        gx, gy = plank[0]
        same_gravity = euclidianNorm2D((g0x-gx, g0y-gy)) < epsilon

        if same_gravity :
            # if the planks have the same center of gravity, it is enough to check whether they have
            # two adjacent points in common to determine if they are the same.

            for i in range(1, n) :
                ax, ay = plank[i]
                if i == 4 :
                    bx, by = plank[1]
                else :
                    bx, by = plank[i+1]

                same_a = euclidianNorm2D((a0x-ax, a0y-ay)) < epsilon
                same_b = euclidianNorm2D((b0x-bx, b0y-by)) < epsilon

                if same_a and same_b :
                    same_plank = True


        among_planks = among_planks or same_plank
    return among_planks



def convert_point_coordinates(column, line, column_center, line_center, distance_ratio) :
    # See section II.1.d of the report

    x_c = distance_ratio*(column - column_center)
    y_c = -distance_ratio*(line - line_center)
    x = -y_c+0.11
    y = x_c
    return x, y




def is_in_square(x, y, square) :
    xa, ya, xb, yb, xc, yc, xd, yd = square
    # The aim of this function is to check whether point M(x, y) is within the square.
    # A very simple way to do that is to sum the 4 angles AMB, BMC, CMD and DMA. If the sum equals pi, then
    # M is in the square. A good way to compute the angles is through a scalar product.

    # "sp" here refers to a scalar product
    sp_ma_mb = (xa - x)*(xb - x) + (ya - y)*(yb - y)
    sp_mb_mc = (xb - x)*(xc - x) + (yb - y)*(yc - y)
    sp_mc_md = (xc - x)*(xd - x) + (yc - y)*(yd - y)
    sp_md_ma = (xd - x)*(xa - x) + (yd - y)*(ya - y)
```

```
    norm_ma = euclidianNorm2D((xa-x, ya-y))
    norm_mb = euclidianNorm2D((xb-x, yb-y))
    norm_mc = euclidianNorm2D((xc-x, yc-y))
    norm_md = euclidianNorm2D((xd-x, yd-y))

    amb = np.arccos(sp_ma_mb/(norm_ma*norm_mb))
    bmc = np.arccos(sp_mb_mc/(norm_mb*norm_mc))
    cmd = np.arccos(sp_mc_md/(norm_mc*norm_md))
    dma = np.arccos(sp_md_ma/(norm_md*norm_ma))

    sum = amb + bmc + cmd + dma
    # print("sum :", sum)
    in_square = 2*np.pi*(1-1/8) <= sum and sum <= 2*np.pi*(1+1/8)
    return in_square



def convertPlankCoordinates(planks_list, nb_lines_frame, nb_columns_frame, robot_line, robot_column, 
distance_ratio) :
    # Last but not least, we need to convert the coordinates of the planks into ones usable by the robot.
    # Or, more specifically, we only need to give the robot the coordinates of the center of gravity, and 
maybe of
    # a few points to visit to realign a plank if necessary.

    planks_converted = []
    column_center = nb_columns_frame/2 # Center of the frame
    line_center = nb_lines_frame/2 # Center of the frame

    for plank in planks_list :
        plank_converted = []

        misaligned = isPlankMisaligned(robot_line, robot_column, plank)
        line_g, column_g = plank[0] # center of gravity
        x_g, y_g = convert_point_coordinates(column_g, line_g, column_center, line_center, distance_ratio)

        plank_converted.append(misaligned)
        plank_converted.append((x_g, y_g))

        if misaligned :
            # in this case, the robot cannot seize the plank and needs to move or rotate it.
            # One way to do this is to tackle (literally) one of the corners of the plank, since the robot 
has no "wrist".
            # In particular, aiming for the corner of the plank that is the farthest from the robot
            # is a good way to rotate the plank efficiently. Let us determine this point, called T.

            line_t, column_t = plank[1]
            for line, column in plank[2:] :
                if euclidianNorm2D((line_t - robot_line, column_t - robot_column)) < euclidianNorm2D((line - 
robot_line, column - robot_column)) :
```

```
                    line_t, column_t = line, column

            x_t, y_t = convert_point_coordinates(column_t, line_t, column_center, line_center, 
distance_ratio)

            # Lastly, in order to properly charge point T, the robot should move in a straight line.
            # Its charge should begin from a point on the (OT) line. For instance, the point located at a 
fifth
            # of the coordinates of point T. But not too close to the robot either.
            # We will now append these two pieces of information to plank_converted.

            x_i, y_i = x_t/5, y_t/5
            if abs(x_i) < 0.05 : # If too close to the robot
                x_i = 0.05*x_t/abs(x_t)
            if abs(y_i) < 0.05 :
                y_i = 0.05*y_t/abs(y_t)


            plank_converted.append((x_i, y_i))
            plank_converted.append((x_t, y_t))

        else :
            # In this case, the robot does not need to charge the plank.
            # We will append harmless and useless coordinates.
            plank_converted.append((0, 0.1))
            plank_converted.append((0, 0.1))
        if euclidianNorm2D((x_g, y_g)) > 0.05 :
            planks_converted.append(plank_converted)

    return planks_converted



#################################
### CAMERA VISION CALIBRATION ###
#################################

# The point of this section is to avoid having to manually calibrate the Canny Edge Detector
# and to manually set the position of the robot (and of points A and B from section II.1.d of the report)
# each time we wish to start the robot.

# The idea is to use the same principle as QR codes : 3 of the 4 corners of the robot's workspace will be 
marked
# with a colored (red) sticker, so as to detect these three points.



def red_filter(img) : # to only see the red part of each pixel.
    # Bear in mind that a cv2 image is in BGR, meaning that red is the last component of a pixel here.
    red = deepcopy(img)
```

```
    for i in range(len(img)) :
        for j in range(len(img[0])) :
            red[i][j][0] = 0
            red[i][j][1] = 0
            if img[i][j][2] < 200 or img[i][j][0] >= 100 or img[i][j][1] >= 100 :
                # If not red enough, or if too green or too blue in addition to red
                red[i][j][2] = 0
    return red



def blue_filter(img) : # to only see the blue part of each pixel.
    # Bear in mind that a cv2 image is in BGR, meaning that blue is the first component of a pixel here.
    blue = deepcopy(img)
    for i in range(len(img)) :
        for j in range(len(img[0])) :
            blue[i][j][1] = 0
            blue[i][j][2] = 0
            if img[i][j][0] < 100 or img[i][j][1] >= 100 or img[i][j][2] >= 100 :
                # If not blue enough, or if too green or too red in addition to blue
                blue[i][j][0] = 0
    return blue



def find_red_stickers(img) :
    for i in range(2) :
        img = cv2.GaussianBlur(img, (5, 5), cv2.BORDER_DEFAULT)
    # cv2.imshow("blur", img)

    red = red_filter(img)
    for i in range(0) :
        red = cv2.GaussianBlur(red, (5, 5), cv2.BORDER_DEFAULT)
    # cv2.imshow("red filter", red)

    canny = cannyEdgeDetector(red, 10, 17)
    # cv2.imshow("canny", canny)

    # cv2.waitKey(0)
    # cv2.destroyAllWindows()

    contours, hierarchy = cv2.findContours(canny, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_NONE)
    # Hopefully the contours detected here are only the three stickers.

    gravity_centers = []

    for contour in contours :
        perimeter = cv2.arcLength(contour, closed = True)
        epsilon = 0.03*perimeter
        approximate_shape_corners = cv2.approxPolyDP(contour, epsilon, closed = True)
```

```python
        # The stickers are probably round, but we don't care about approximating correctly their shape.
        # All we need is to recover the center of gravity of each sticker.

        # Let us compute the coordinates of the center of gravity of each red sticker
        sum_column = 0
        sum_line = 0

        for corner in approximate_shape_corners :
            corner_column = corner[0][0]
            corner_line = corner[0][1]
            sum_column += corner_column
            sum_line += corner_line

        gravity_column = int(sum_column/len(approximate_shape_corners))
        gravity_line = int(sum_line/len(approximate_shape_corners))
        gravity_centers.append((gravity_line, gravity_column))

    return gravity_centers



def stickers2robot(img) :
    gravity_centers = find_red_stickers(img) # centers of gravity of the stickers

    # let us find the sticker at the top-left corner, and let's call it A.
    # the norm of its center of gravity, computed with the top left corner as origin,
    # is the smallest among the stickers.

    a_line, a_column = gravity_centers[0]
    for x, y in gravity_centers[1:] :
        if x**2 + y**2 < a_line**2 + a_column**2 :
            a_line, a_column = x, y
    cv2.circle(img, (a_column, a_line), 10, (255, 0, 0), thickness = 5)

    # Likewise, let us find the sticker at the botton-right corner, called C.
    # the norm of its center of gravity is the highest.

    c_line, c_column = gravity_centers[0]
    for x, y in gravity_centers[1:] :
        if x**2 + y**2 > c_line**2 + c_column**2 :
            c_line, c_column = x, y
    cv2.circle(img, (c_column, c_line), 10, (0, 255, 0), thickness = 5)

    # Finally, let's find the last sticker, called B, at the bottom-left corner.

    b_line, b_column = 0, 0
    epsilon = euclidianNorm2D((c_line, c_column))/5
    for x, y in gravity_centers :
        if euclidianNorm2D((x-a_line, y-a_column)) > epsilon and euclidianNorm2D((x-c_line, y-c_column)) >
epsilon :
```

```python
            b_line, b_column = x, y
            cv2.circle(img, (b_column, b_line), 10, (255, 255, 255), thickness = 5)

    d_line = int( a_line + (c_line - b_line) )
    d_column = int( a_column + (c_column - b_column) )
    cv2.circle(img, (d_column, d_line), 10, (0, 255, 255), thickness = 5)

    # Now, we can start looking for the robot (point O) using vectors AB and BC.
    # We know that O = A + (1/2)BC + (4/30)AB

    robot_line = int( a_line + (1/2)*(c_line - b_line) + (4/30)*(b_line - a_line) )
    robot_column = int( a_column + (1/2)*(c_column - b_column) + (4/30)*(b_column - a_column) )
    cv2.circle(img, (robot_column, robot_line), 10, (0, 0, 255), thickness = 5)

    distance_ratio = 0.3/euclidianNorm2D((b_line-a_line, b_column-a_column))
    square = [a_line, a_column, b_line, b_column, c_line, c_column, d_line, d_column]

    # print("Is robot in square :", is_in_square(robot_line, robot_column, square))
    # print("Is (0, 0) in square :", is_in_square(0, 0, square))

    cv2.imshow("stickers", img)
    cv2.waitKey(0)

    return robot_line, robot_column, distance_ratio, square



def cameraVision_calibrate(camera_number_on_pc) :
    capture = openCapture(camera_number_on_pc)

    ret, frame = capture.read()
    while not(ret) :
        ret, frame = capture.read()

    frame = undistortImage(frame, mtx, mtx, distorsion_coefficients)
    frame = chopChop(frame, 0)

    robot_line, robot_column, distance_ratio, square = stickers2robot(frame)
    return robot_line, robot_column, distance_ratio, square



#####################
### CAMERA VISION ###
#####################

def cameraVision(camera_number_on_pc, robot_line, robot_column, distance_ratio, square) :
    mtx = np.array([[426, 0, 332], [0, 426, 198], [0, 0, 1]])
    distorsion_coefficients = np.array([[-0.266, 0.07, -0.006, 0.0008, 0.0058]])
    min_area = 3000
```

```python
    max_area = 10000
    threshold1 = 10
    threshold2 = 70

    capture = openCapture(camera_number_on_pc)
    list_of_lists = []

    for i in range(20):
        ret, frame = capture.read()

        if ret:
            frame = undistortImage(frame, mtx, mtx, distorsion_coefficients)
            frame = chopChop(frame, 0)
            nb_lines, nb_columns, nb_colors = frame.shape

            canny = cannyEdgeDetector(frame, threshold1, threshold2)
            canny_canvas = deepcopy(canny)
            planks_list_1frame = findCentersGravity(frame, canny, canny_canvas, min_area, max_area)

            # cv2.imshow("canny", canny)
            # cv2.imshow("canny canvas", canny_canvas)

        # cv2.waitKey(200)
        list_of_lists.append(planks_list_1frame)

    closeCapture(capture)
    planks_list_final = []

    for planks_list in list_of_lists :
        for plank in planks_list :
            gx, gy = plank[0]
            if is_in_square(gx, gy, square) :
                ax, ay = plank[1]
                epsilon = euclidianNorm2D((gx-ax, gy-ay))/3
                if not(is_among_planks(plank, planks_list_final, epsilon)) :
                    planks_list_final.append(plank)
                    cv2.circle(frame, (gy, gx), 10, (255, 0, 0), thickness = 4)

    # cv2.imshow("detected planks :"+str(len(planks_list_final)), frame)
    # cv2.waitKey(200)

    return convertPlankCoordinates(planks_list_final, nb_lines, nb_columns, robot_line, robot_column,
distance_ratio)
```