Track and grab color blocks

Before starting this function, you need to close the process of the big program and APP. Enter the following program in the terminal to close the process of the big program and APP.

```
sh ~/app_Arm/kill_YahboomArm.sh
sh ~/app_Arm/stop_app.sh
```

If you need to start the big program and APP again later, start the terminal.

```
sudo systemctl start yahboom_arm.service
sudo systemctl start yahboom_app.service
```

1. Function description

After the program is started, use the mouse to select an area on the handheld color block and obtain the HSV value of the color block. The program will identify the color block, and the robotic arm will start tracking the color block so that the center of the color block coincides with the center of the image; after the robotic arm stops, wait for 2-3 seconds. If the depth distance is valid (not 0) at this time, the buzzer will sound, and then the robotic arm will adjust its posture to grab the color block. After gripping, place it at the set position and then return to the position of identifying the color block.

2. Start and operate

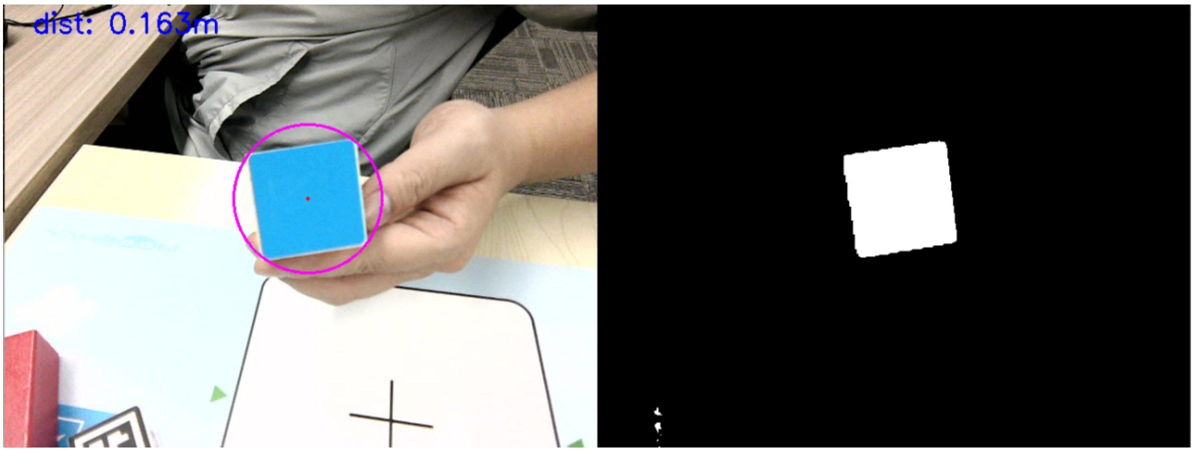
2.1. Start command

The terminal starts with the following command,

```
#Start the camera
roslaunch orbbec_camera dabai_dcw2.launch
#Start the underlying control robot
rosrun dofbot_pro_info arm_driver.py
#Start the inverse solution program
rosrun dofbot_pro_info kinemarics_dofbot_pro
#Start the machine code tracking and gripping program
rosrun dofbot_pro_color color_follow.py
```

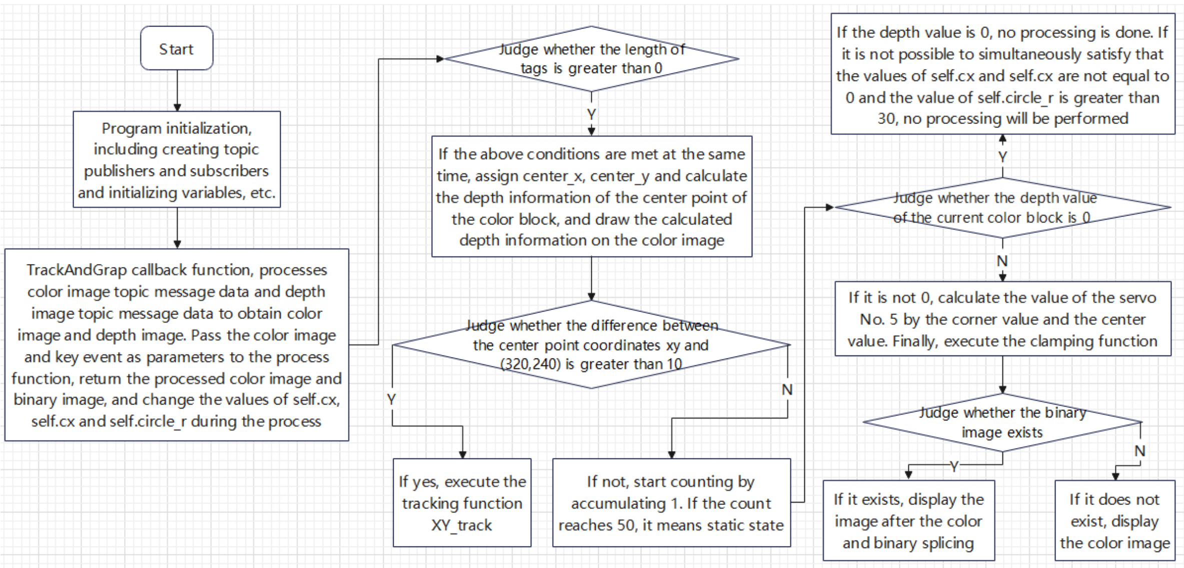
2.2. Operation

After the program is started, hold a 4cm*4cm color block in the image, select a part of the color block with the mouse, obtain the HSV value of the color block, and the program starts to recognize the color block; the robot arm will adjust its posture so that the center of the color block coincides with the center of the image; slowly move the color block, the robot arm follows the movement of the color block, and constantly adjusts its posture. After waiting for the center of the color block to coincide with the center of the image, if the depth information of the upper left corner of the image is not 0 and the distance between the color block and the base is less than 30cm, it means that the depth value of the center coordinate of the color block is valid and within the gripping range of the robot arm. The buzzer will sound once, and then the robot arm will change its posture to grip the color block according to the position of the color block; after gripping, it will be placed at the set position and finally return to the recognized posture. If the robot arm cannot meet the condition that the depth information of the upper left corner of the image is not 0 and the distance between the color block and the base is less than 30cm (greater than 18cm), it is necessary to move the color block back and forth again so that it can track again and then stop to meet the gripping conditions.



3. Program flow chart

color_follow.py



4. Core code analysis

Code path: `/home/jetson/dofbot_pro_ws/src/dofbot_pro_color/scripts/color_follow.py`

Import necessary libraries,

```
import cv2
import rospy
import numpy as np
from sensor_msgs.msg import Image
import message_filters
from std_msgs.msg import Float32,Bool
import os
from cv_bridge import CvBridge
import cv2 as cv
#Import custom tracking and gripping library
from Dofbot_Track import *
#Deep image encoding format
encoding = ['16UC1', '32FC1']
```

```
import time
#color recognition
#Import custom image processing library
from astra_common import *
#Import dynamic parameter service library
from dynamic_reconfigure.server import Server
from dynamic_reconfigure.client import Client
import rospkg
from dofbot_pro_color.cfg import ColorHSVConfig
import math
from dofbot_pro_info.msg import *
```

Initialize program parameters, create publishers, subscribers, etc.

```
def __init__(self):
    nodeName = 'color_detect'
    rospy.init_node(nodeName)
    self.window_name = "depth_image"
    self.init_joints = [90.0, 150, 12.0, 20.0, 90, 30]
    #Create an object to track gripping
    self.dofbot_tracker = DofbotTrack()
    #Create a publisher to publish the target angle of the robot arm
    self.pubPoint = rospy.Publisher("TargetAngle", ArmJoint, queue_size=1)
    #Create a subscriber to subscribe to the result of gripping completion
    self.grasp_status_sub = rospy.Subscriber('grab', Bool,
self.grabStatusCallback, queue_size=1)
    #Create two subscribers to subscribe to the color image topic and the depth
image topic
    self.depth_image_sub =
message_filters.Subscriber('/camera/depth/image_raw',Image)
    self.rgb_image_sub =
message_filters.Subscriber('/camera/color/image_raw',Image)
    #Time-synchronize the messages subscribed to color and depth images
    self.TimeSynchronizer =
message_filters.ApproximateTimeSynchronizer([self.rgb_image_sub,self.depth_image
_sub],10,0.5)
    #Callback function TagDetect that handles synchronization messages. The
callback function is connected to the subscribed message so that it can be
automatically called when a new message is received
    self.TimeSynchronizer.registerCallback(self.TrackAndGrap)
    #Counting variable, used to record the number of times that meet the
conditions
    self.cnt = 0
    #Create a bridge for color and depth image topic message data to image data
    self.rgb_bridge = CvBridge()
    self.depth_bridge = CvBridge()
    #color
    #Initialize region coordinates
    self.Roi_init = ()
    #Initialize HSV values
    self.hsv_range = ()
    #Initialize the information of the recognized color block, which represents
the center x coordinate, center y coordinate and minimum circumscribed circle
radius r of the color block
    self.circle = (0, 0, 0)
    #Flag for dynamic parameter adjustment, if True, dynamic parameter adjustment
is performed
```

```
        self.dyn_update = True
        #Flag for mouse selection
        self.select_flags = False
        self.gTracker_state = False
        self.windows_name = 'frame'
        self.Track_state = 'init'
        #Create color detection object
        self.color = color_detect()
        #Initialize row and column coordinates of region coordinates
        self.cols, self.rows = 0, 0
        #Initialize xy coordinates of mouse selection
        self.Mouse_XY = (0, 0)
        #Store xy coordinates of color block center value
        self.cx = 0
        self.cy = 0
        #Default path of HSV threshold file, which stores the last saved HSV value
        self.hsv_text = rospkg.RosPack().get_path("dofbot_pro_color") +
"/scripts/colorHSV.text"
        Server(ColorHSVConfig, self.dynamic_reconfigure_callback)
        self.dyn_client = Client(nodeName, timeout=60)
        #The minimum circumscribed circle radius of the color block obtained after
image processing
        self.circle_r = 0
        #The depth value of the current center coordinate of the color block
        self.cur_distance = 0.0
        #The xy coordinates of the corner point, used to calculate the value of the
No. 5 servo
        self.corner_x = self.corner_y = 0.0
        exit_code = os.system('rosservice call /camera/set_color_exposure  50')
```

主要看TrackAndGrap回调函数，

```
def TrackAndGrap(self,color_frame,depth_frame):
        #Receive the color image topic message and convert the message data into
image data
        rgb_image = self.rgb_bridge.imgmsg_to_cv2(color_frame,'bgr8')
        result_image = np.copy(rgb_image)
        #Receive the depth image topic message and convert the message data into
image data
        depth_image = self.depth_bridge.imgmsg_to_cv2(depth_frame, encoding[1])
        frame = cv.resize(depth_image, (640, 480))
        depth_image_info = frame.astype(np.float32)
        action = cv.waitKey(10) & 0xFF
        result_image = cv.resize(result_image, (640, 480))
        #Pass the obtained color image as a parameter to process, and pass the
keyboard event action at the same time
        result_frame, binary = self.process(result_image,action)
       #Judge whether the coordinate xy of the color block center value is not 0 and
the minimum circumscribed circle radius of the color block is greater than 30
        if self.cx!=0 and self.cy!=0 and self.circle_r>30 :
            #Judge whether the xy coordinates of the center value of the color block
are within the valid range
            if self.cx<=640 or self.cy <=480:
                center_x, center_y = self.cx,self.cy
                #Calculate the depth value of the center point of the color block
                self.cur_distance =
depth_image_info[int(center_y),int(center_x)]/1000.0
```

```
            print("self.cur_distance: ",self.cur_distance)
            dist = round(self.cur_distance,3)
            dist = 'dist: ' + str(dist) + 'm'
            #Draw the depth value of the center point on the color image
            cv.putText(result_frame, dist,  (30, 30), cv.FONT_HERSHEY_SIMPLEX,
1.0, (255, 0, 0), 2)
            #If the coordinates of the center point of the color block and the
center point of the image (320, 240) are greater than 10, that is, they are not
within the acceptable range, the tracking program is executed and the state of
the robot arm is adjusted to make the center value of the color block within the
acceptable range
            if abs(center_x-320) >10 or abs(center_y-240)>10:
                #Execute the tracking program, the input is the center value of
the current color block
                self.dofbot_tracker.XY_track(center_x,center_y)
            #If the coordinates of the center point of the machine code and the
center point of the image (320, 240) are less than 10, it can be considered that
the center value of the machine code is in the middle of the image
            else:
                #If the conditions are met, accumulate self.cnt
                self.cnt = self.cnt + 1
                #When the cumulative number reaches 50, it means that the center
value of the machine code can be stationary in the middle of the image.
                if self.cnt==50:
                    # Clear the count of self.cnt
                    self.cnt = 0
                    print("take it now!")
                    #Judge whether the current depth value is 0. If it is not
zero, it means the value is valid
                    if self.cur_distance!=0:
                        # Calculate the value of servo No. 5 through the corner
point coordinates
                        angle_radians = math.atan2(self.corner_y, self.corner_x)
                        angle_degrees = math.degrees(angle_radians)
                        print("angle_degrees: ",angle_degrees)
                        if abs(angle_degrees) >90:
                            compute_angle = abs(angle_degrees) - 45
                        else:
                            compute_angle = abs(angle_degrees)
                        print("compute_angle: ",compute_angle)
                        self.dofbot_tracker.set_joint5 = compute_angle
                        #Execute the clamping program, calling the Clamping
function of the created dofbot_tracker object. The input parameters are the
center value of the color block and the depth value of the center point.

 self.dofbot_tracker.Clamping(center_x,center_y,self.cur_distance)

    # Check if the binary image exists. If it does, display the color and binary
images. Otherwise, only display the color image.
    if len(binary) != 0: cv.imshow(self.windows_name, ManyImgs(1,
([result_frame, binary]))
    else:
        cv.imshow(self.windows_name, result_frame)
```

Image processing function self.process,

```
def process(self, rgb_img, action):
```

```
    rgb_img = cv.resize(rgb_img, (640, 480))
    binary = []
    #Judge key events. When i or I is pressed, change the state to identification
mode
    if action == ord('i') or action == ord('I'): self.Track_state = "identify"
    #Judge key events. When r or R is pressed, reset all parameters and enter
color selection mode
    elif action == ord('r') or action == ord('R'): self.Reset()
    #Judge the state value. If it is init, it means the initial state value. At
this time, you can use the mouse to select the area
    if self.Track_state == 'init':
        #Select the color of an area within the specified window
        cv.namedWindow(self.windows_name, cv.WINDOW_AUTOSIZE)
        cv.setMouseCallback(self.windows_name, self.onMouse, 0)
        #Judge the color selection flag, true means you can select the color
        if self.select_flags == True:
            cv.line(rgb_img, self.cols, self.rows, (255, 0, 0), 2)
            cv.rectangle(rgb_img, self.cols, self.rows, (0, 255, 0), 2)
            # Check if the selected area exists
            if self.Roi_init[0] != self.Roi_init[2] and self.Roi_init[1] !=
self.Roi_init[3]:
                #Call the Roi_hsv function in the created color detection object
self.color, and return the processed color image and HSV value
                rgb_img, self.hsv_range = self.color.Roi_hsv(rgb_img,
self.Roi_init)
                self.gTracker_state = True
                self.dyn_update = True
            else: self.Track_state = 'init'
    #Judge the status value. If it is "identify", it means that color recognition
can be performed.
    elif self.Track_state == "identify":
        # Check if there is an HSV threshold file. If so, read the value in it
and assign it to hsv_range
        if os.path.exists(self.hsv_text): self.hsv_range =
read_HSV(self.hsv_text)
        #If it does not exist, change the state to init to select the color
        else: self.Track_state = 'init'
    if self.Track_state != 'init':
        #Judge the length of the self.hsv_range value, that is, whether the
value exists. When the length is not 0, enter the color detection function
        if len(self.hsv_range) != 0:
            #Call the object_follow function in the created color detection
object self.color, pass in the color image and self.hsv_range, which is the hsv
threshold, and return the processed color image, the binary image, and the
information that stores the hsv threshold graphic, including the center point
coordinates and the radius of its minimum circumscribed circle
            rgb_img, binary, self.circle ,corners=
self.color.object_follow(rgb_img, self.hsv_range)
            print("corners[0]: ",corners[0][0])
            print("corners[0]: ",corners[0][1])
            self.corner_x = int(corners[0][0]) - int(self.circle[0])
            self.corner_y = int(corners[0][1]) - int(self.circle[1])
            #Assign the return value to self.cx and self.cy that store the
center value, and assign the radius of the minimum circumscribed circle to
self.circle_r
            self.cx = self.circle[0]
            self.cy = self.circle[1]
            self.circle_r = self.circle[2]
```

```
            #The flag for determining dynamic parameter updates. True means that
the hsv_text file can be updated and the value on the parameter server can be
modified.
            if self.dyn_update == True:
                write_HSV(self.hsv_text, self.hsv_range)
                params = {'Hmin': self.hsv_range[0][0], 'Hmax':
self.hsv_range[1][0],
                          'Smin': self.hsv_range[0][1], 'Smax':
self.hsv_range[1][1],
                          'Vmin': self.hsv_range[0][2], 'Vmax':
self.hsv_range[1][2]}
                self.dyn_client.update_configuration(params)
                self.dyn_update = False

    return rgb_img, binary
```

Let's take a look at the implementation of the Clamping function of the created dofbot_tracker object. This function is located in the Dofbot_Track library.
/home/jetson/dofbot_pro_ws/src/dofbot_pro_color/scripts/Dofbot_Track.py

```
def Clamping(self,cx,cy,cz):
    #Get the current position and pose of the end of the robot
    self.get_current_end_pos(self.cur_joints)
    #Start the coordinate system conversion and finally get the position of the
center coordinate of the machine code in the world coordinate system
    camera_location = self.pixel_to_camera_depth((cx,cy),cz)
    PoseEndMat = np.matmul(self.EndToCamMat,
self.xyz_euler_to_mat(camera_location, (0, 0, 0)))
    EndPointMat = self.get_end_point_mat()
    WorldPose = np.matmul(EndPointMat, PoseEndMat)
    pose_T, pose_R = self.mat_to_xyz_euler(WorldPose)
    #Add the offset parameter to compensate for the deviation caused by the
difference in servo values
    pose_T[0] = pose_T[0] + self.x_offset
    pose_T[1] = pose_T[1] + self.y_offset
    pose_T[2] = pose_T[2] + self.z_offset
    #Call the IK algorithm for inverse solution. The parameters passed in are
obtained through the coordinate system conversion to pose_T, that is, the
position of the center coordinates of the machine code in the world coordinate
system, and the values of the 6 servos are obtained through IK calculation
    #Call the inverse solution service, calling the ik service content, and
assigning the required request parameters
    request = kinemaricsRequest()
    #The target x value at the end of the robot arm, in m
    request.tar_x = pose_T[0]
    #The target y value at the end of the robot arm, in m
    request.tar_y = pose_T[1]
    #The target z value at the end of the robot arm, in m, 0.2 is the scaling
factor, and small adjustments are made according to the actual situation
    request.tar_z = pose_T[2]
    #Specify the service content as ik
    request.kin_name = "ik"
    #The target Roll value at the end of the robot arm, in radians, this value is
the current roll value at the end of the robot arm
    request.Roll = self.CurEndPos[3]
    print("calcutelate_request: ",request)
    try:
```

```
        response = self.client.call(request)
        joints = [0.0, 0.0, 0.0, 0.0, 0.0,0.0]
        #Assign the joint1-joint6 values returned by the call service to joints
        joints[0] = response.joint1 #response.joint1
        joints[1] = response.joint2
        joints[2] = response.joint3
        if response.joint4>90:
            joints[3] = 90
        else:
            joints[3] = response.joint4
        joints[4] = 90
        joints[5] = 20
        #Calculate the distance between the machine code and the robot base coordinate system
        dist = math.sqrt(request.tar_y ** 2 + request.tar_x** 2)
        #If the distance is between 18 cm and 30 cm, control the robot arm to move to the gripping point
        if dist>0.18 and dist<0.30:
            self.Buzzer()
            print("compute_joints: ",joints)
            #Execute the pubTargetArm function and pass the calculated joints value as a parameter
            self.pub_arm(joints)
            #Execute the actions of grabbing, moving and placing
            self.move()
        else:
            print("It's too far to catch it!Please move it forward a bit. ")
    except Exception:
       rospy.loginfo("run error")
```