Color Block Tracking and Grasping

Before starting this function, you need to close the main program and APP processes. If you need to restart the main program and APP later, start them from the terminal:

```
bash ~/dofbot_pro/APP_DOFBOT_PRO/start_app.sh
```

1. Function Description

After the program starts, it recognizes color blocks and tracks them, making the color block center coincide with the image center. After the robotic arm is stationary for 2-3 seconds, if the depth distance is valid (not 0), the buzzer will sound once, and the robotic arm will adjust its posture to grasp the color block. After grasping, it places it at the designated position, then returns to the color block recognition posture.

2. Startup and Operation

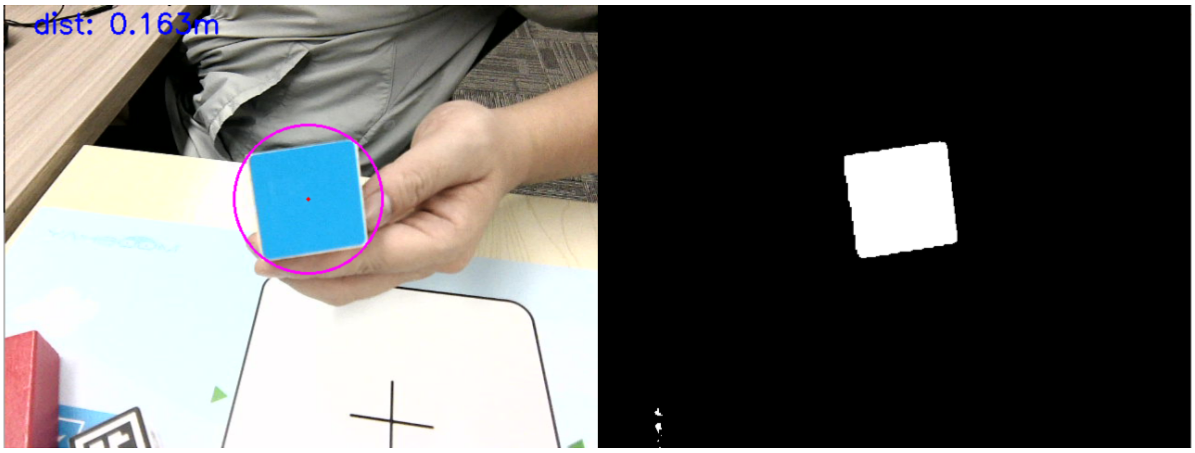
2.1. Startup Commands

Enter the following commands in the terminal to start:

```
# Start camera:
ros2 launch orbbec_camera dabai_dcw2.launch.py
# Start low-level control:
ros2 run dofbot_pro_driver arm_driver
# Start inverse kinematics program:
ros2 run dofbot_pro_info kinemarics_dofbot
# Start color block tracking and grasping program:
ros2 run dofbot_pro_color color_follow
```

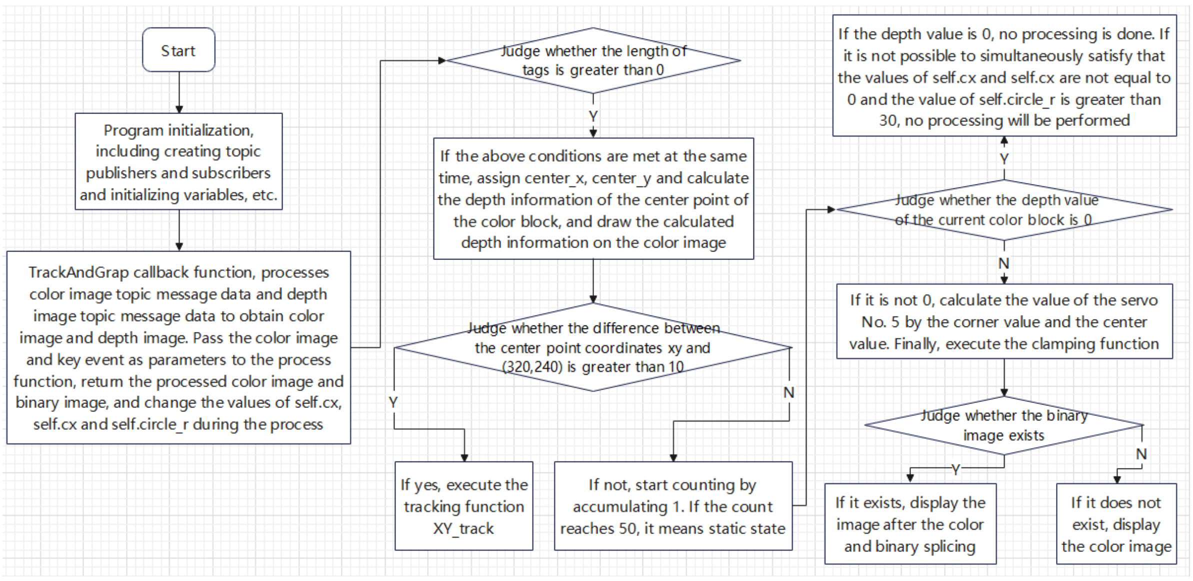
2.2. Operation

After the program starts, hold a 4cm*4cm color block in front of the camera. Use the mouse to select a part of the color block area to obtain the HSV value of the color block. The program starts recognizing the color block. The robotic arm will adjust its posture to make the color block center coincide with the image center. Slowly move the color block, and the robotic arm will follow the color block movement, continuously adjusting its posture. After the color block center coincides with the image center, if the depth information in the upper left corner of the image is not 0 and the distance from the color block to the base is less than 30cm, it means the color block center coordinate depth value is valid and within the robotic arm's grasping range. The buzzer will sound once, and the robotic arm will adjust its posture to grasp the color block based on its position. After grasping, it places it at the designated position and finally returns to the recognition posture. If the robotic arm cannot meet the conditions (depth information not 0 and distance less than 30cm, greater than 18cm), you need to move the color block back and forth again to make it stationary after tracking to meet the grasping conditions.



3. Program Flowchart

color_follow.py


Start
Program initialization, including creating topic publishers and subscribers and initializing variables, etc.
TrackAndGrap callback function, processes color image topic message data and depth image topic message data to obtain color image and depth image. Pass the color image and key event as parameters to the process function, return the processed color image and binary image, and change the values of self.cx, self.cx and self.circle_r during the process
Judge whether the length of tags is greater than 0
Y
If the above conditions are met at the same time, assign center_x, center_y and calculate the depth information of the center point of the color block, and draw the calculated depth information on the color image
Judge whether the difference between the center point coordinates xy and (320,240) is greater than 10
Y
N
If yes, execute the tracking function XY_track
If not, start counting by accumulating 1. If the count reaches 50, it means static state
If the depth value is 0, no processing is done. If it is not possible to simultaneously satisfy that the values of self.cx and self.cx are not equal to 0 and the value of self.circle_r is greater than 30, no processing will be performed
Y
Judge whether the depth value of the current color block is 0
N
If it is not 0, calculate the value of the servo No. 5 by the corner value and the center value. Finally, execute the clamping function
Judge whether the binary image exists
Y
N
If it exists, display the image after the color and binary splicing
If it does not exist, display the color image

4. Core Code Analysis

Code path:

```
/home/jetson/dofbot_pro_ws/src/dofbot_pro_color/dofbot_pro_color/color_follow.py
```

Import necessary libraries:

```python
import cv2
import rclpy
from rclpy.node import Node
import numpy as np
from message_filters import ApproximateTimeSynchronizer, Subscriber
from sensor_msgs.msg import Image
from std_msgs.msg import Float32, Bool
from cv_bridge import CvBridge
import cv2 as cv

encoding = ['16UC1', '32FC1']
import time
import math
import os
#color recognition
```

```
from dofbot_pro_color.astra_common import *
from dofbot_pro_interface.msg import *
from dofbot_pro_color.Dofbot_Track import *
import tf_transformations as tf
import transforms3d as tfs
```

Program parameter initialization, create publishers, subscribers, etc.:

```
def __init__(self):
    super().__init__('color_detect')
    self.declare_param()
    self.window_name = "depth_image"
    self.init_joints = [90.0, 150.0, 12.0, 20.0, 90.0, 30.0]
    self.dofbot_tracker = DofbotTrack()
    self.cx = 0
    self.cy = 0
    self.pubPoint = self.create_publisher(ArmJoint, "TargetAngle", 1)
    self.grasp_status_sub = self.create_subscription(Bool, 'grab',
self.grabStatusCallback, 1)

    self.depth_image_sub  = Subscriber(self, Image, "/camera/color/image_raw",
qos_profile=1)
    self.rgb_image_sub = Subscriber(self, Image, "/camera/depth/image_raw",
qos_profile=1)
    self.TimeSynchronizer = ApproximateTimeSynchronizer([self.depth_image_sub,
self.rgb_image_sub],queue_size=10,slop=0.5)
    self.TimeSynchronizer.registerCallback(self.TrackAndGrap)
    #Count variable, used to record the number of times conditions are met
    self.cnt = 0
    #Create bridges for converting color and depth image topic message data to
image data
    self.rgb_bridge = CvBridge()
    self.depth_bridge = CvBridge()
    #color
    #Initialize area coordinates
    self.Roi_init = ()
    #Initialize HSV values
    self.hsv_range = ()
    #Initialize recognized color block information, here representing color block
center x coordinate, center y coordinate and minimum enclosing circle radius r
    self.circle = (0, 0, 0)
    #Dynamic parameter adjustment flag, True means perform dynamic parameter
adjustment
    self.dyn_update = True
    #Mouse selection flag
    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 area row and column coordinates
    self.cols, self.rows = 0, 0
    #Initialize mouse selected xy coordinates
    self.Mouse_XY = (0, 0)
    #Store color block center value coordinates xy
    self.cx = 0
    self.cy = 0
    #Default HSV threshold file path, this file stores last saved HSV values
    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)
    #Minimum enclosing circle radius of color block after image processing
    self.circle_r = 0
    #Current depth value of color block center coordinates
    self.cur_distance = 0.0
    #Corner coordinates xy, used to calculate servo 5 value
    self.corner_x = self.corner_y = 0.0
    exit_code = os.system('rosservice call /camera/set_color_exposure  50')
```

Focus on the TrackAndGrap callback function:

```
def TrackAndGrap(self,color_frame,depth_frame):
    #Receive color image topic message, convert message data to image data
    rgb_image = self.rgb_bridge.imgmsg_to_cv2(color_frame,'bgr8')
    result_image = np.copy(rgb_image)
    #Receive depth image topic message, convert message data to 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 parameter to process, and also pass
keyboard event action
    result_frame, binary = self.process(result_image,action)
    #Check if stored color block center value coordinates xy are not 0 and color
block minimum enclosing circle radius is greater than 30
    if self.cx!=0 and self.cy!=0 and self.circle_r>30 :
        #Check if stored color block center value coordinates xy are within
valid range
        if self.cx<=640 or self.cy <=480:
            center_x, center_y = self.cx,self.cy
            #Calculate depth value of color block center point
            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 center point depth value on color image
            cv.putText(result_frame, dist,  (30, 30), cv.FONT_HERSHEY_SIMPLEX,
1.0, (255, 0, 0), 2)
            #If color block center point coordinates differ from image center
point (320, 240) by more than 10, meaning not within acceptable range, execute
tracking program to adjust robotic arm state to bring color block center value
within acceptable range
            if abs(center_x-320) >10 or abs(center_y-240)>10:
                #Execute tracking program, input is current color block center
value
                self.dofbot_tracker.XY_track(center_x,center_y)
            #If color block center point coordinates differ from image center
point (320, 240) by less than 10, can be considered that color block center value
is in the middle of the image
            else:
                #Under satisfying conditions, accumulate self.cnt
                self.cnt = self.cnt + 1
                #When accumulated count reaches 50, it means color block center
value is stationary in the middle of the image
                if self.cnt==50:
                    #Clear self.cnt count
                    self.cnt = 0
                    print("take it now!")
                    #Check if current depth value is 0, non-zero means the value
is valid
                    if self.cur_distance!=0:
                        #Calculate servo 5 value through corner 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 grasping program, calling the Clamping function
of the created dofbot_tracker object, input parameters are color block center
value and center point depth value

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

    #Check if binary image exists, if so display color and binary images,
otherwise only display 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 = []
    #Check key event, when i or I is pressed, change state to recognition mode
    if action == ord('i') or action == ord('I'): self.Track_state = "identify"
    #Check key event, when r or R is pressed, reset all parameters, enter color
selection mode
    elif action == ord('r') or action == ord('R'): self.Reset()
    #Check state value, if init, means initial state value, can use mouse to
select area at this time
    if self.Track_state == 'init':
        #Select an area color within the specified window
        cv.namedWindow(self.windows_name, cv.WINDOW_AUTOSIZE)
        cv.setMouseCallback(self.windows_name, self.onMouse, 0)
        #Check color selection flag, true means can select 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 selected area exists
            if self.Roi_init[0] != self.Roi_init[2] and self.Roi_init[1] !=
self.Roi_init[3]:
```

```
                #Call Roi_hsv function in created color detection object
self.color, returns processed color image and HSV values
                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'
    #Check state value, if "identify", means can perform color recognition

    elif self.Track_state == "identify":
        #Check if HSV threshold file exists, if so read values and assign to
hsv_range
        if os.path.exists(self.hsv_text): self.hsv_range =
read_HSV(self.hsv_text)
        #If not exists, change state to init for color selection
        else: self.Track_state = 'init'
    if self.Track_state != 'init':
        #Check self.hsv_range value length, meaning check if this value exists,
when length is not 0, enter color detection function
        if len(self.hsv_range) != 0:
            #Call object_follow function in created color detection object
self.color, pass color image and self.hsv_range (hsv threshold), returns
processed color image, binary image and information about graphics meeting hsv
threshold, including center point coordinates and its minimum enclosing circle
radius
            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 return values to stored center values self.cx and self.cy,
minimum enclosing circle radius assigned to self.circle_r
            self.cx = self.circle[0]
            self.cy = self.circle[1]
            self.circle_r = self.circle[2]
            #Check dynamic parameter update flag, True means can update hsv_text
file and modify values on parameter server
            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 look at the implementation process of the Clamping function of the created dofbot_tracker object. This function is located in the Dofbot_Track library, Dofbot_Track library location /home/jetson/dofbot_pro_ws/src/dofbot_pro_color/scripts/Dofbot_Track.py

```
def Clamping(self,cx,cy,cz):
    #Get current robotic arm end position and pose
    self.get_current_end_pos(self.cur_joints)
    #Start coordinate system conversion, finally get color block center
coordinates position in 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 offset parameters to compensate for deviations caused by servo value
differences
    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 inverse kinematics IK algorithm, input parameter is pose_T obtained
through coordinate system conversion, which is color block center coordinates
position in world coordinate system, calculate 6 servo values through IK
    #Call inverse kinematics service, calling ik service content, assign required
request parameters
    request = kinemaricsRequest()
    #Target x value of robotic arm end, unit is m
    request.tar_x = pose_T[0]
    #Target y value of robotic arm end, unit is m
    request.tar_y = pose_T[1]
    #Target z value of robotic arm end, unit is m, 0.2 is scaling factor, make
minor adjustments based on actual situation
    request.tar_z = pose_T[2]
    #Specify service content as ik
    request.kin_name = "ik"
    #Target Roll value of robotic arm end, unit is radians, this value is current
robotic arm end roll value
    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 service returned joint1-joint6 values 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 distance from color block to robotic arm base coordinate
system
        dist = math.sqrt(request.tar_y ** 2 + request.tar_x** 2)
        #If distance is between 18cm and 30cm, control robotic arm to move to
grasping point
        if dist>0.18 and dist<0.30:
            self.Buzzer()
            print("compute_joints: ",joints)
            #Execute pubTargetArm function, pass calculated joints values as
parameters
            self.pub_arm(joints)
            #Execute grasping action and movement, placement actions
```

```
                self.move()
            else:
                print("It's too far to catch it!Please move it forward a bit. ")
        except Exception:
           rospy.loginfo("run error")
```