Track and grab machine code

Before starting this function, you need to close the process of the big program and APP. If you need to start the big program and APP again later, start the terminal,

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

1. Function description

After the program is started, after the program recognizes the machine code, it will track the machine code so that the center of the machine code coincides with the center of the image; after the robot 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 robot arm will adjust the posture to clamp the machine code. After clamping, place it at the set position, and then return to the posture of recognizing the machine code.

2. Start and operate

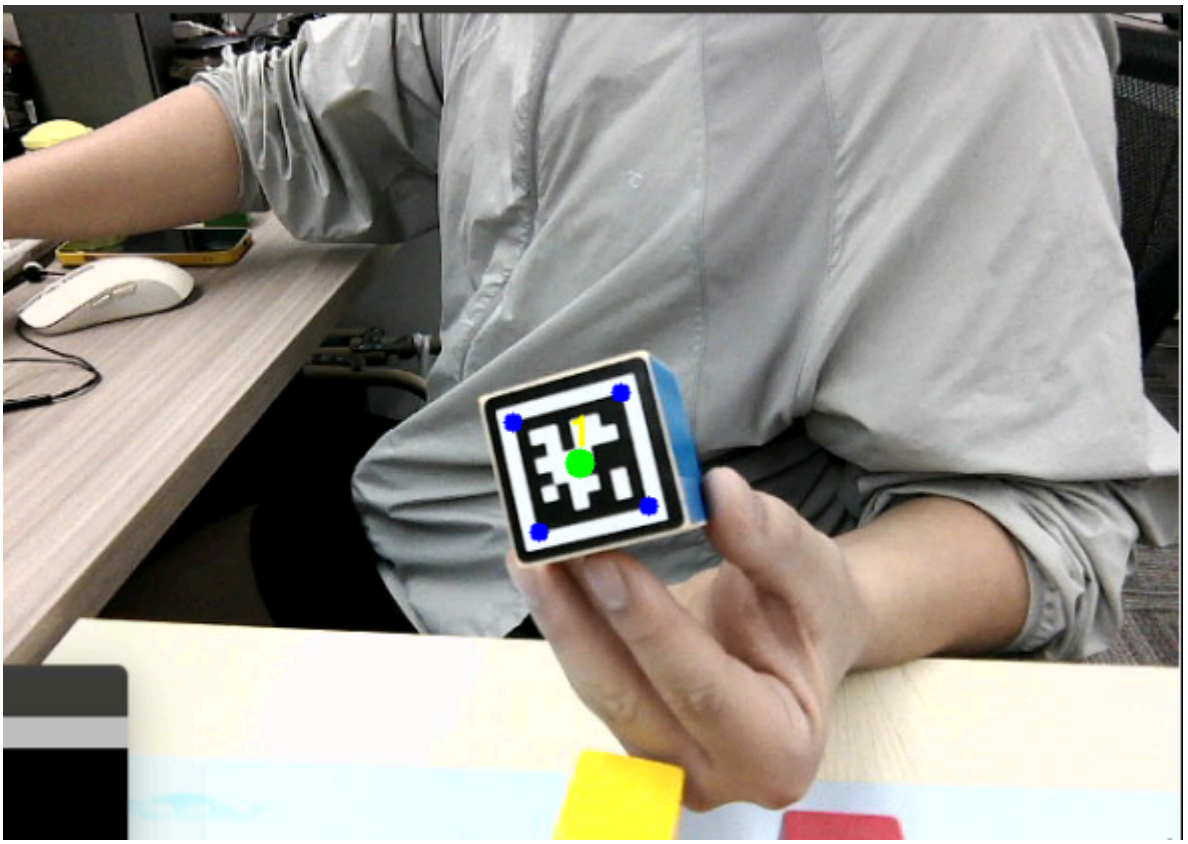
2.1. Start command

Enter the following command in the terminal to start,

```
#Start the camera:
ros2 launch orbbec_camera dabai_dcw2.launch.py
#Start the underlying control:
ros2 run dofbot_pro_driver arm_driver
#Start the inverse program:
ros2 run dofbot_pro_info kinemarics_dofbot
#Start the machine code tracking and gripping program:
ros2 run dofbot_pro_apriltag apriltag_follow
```

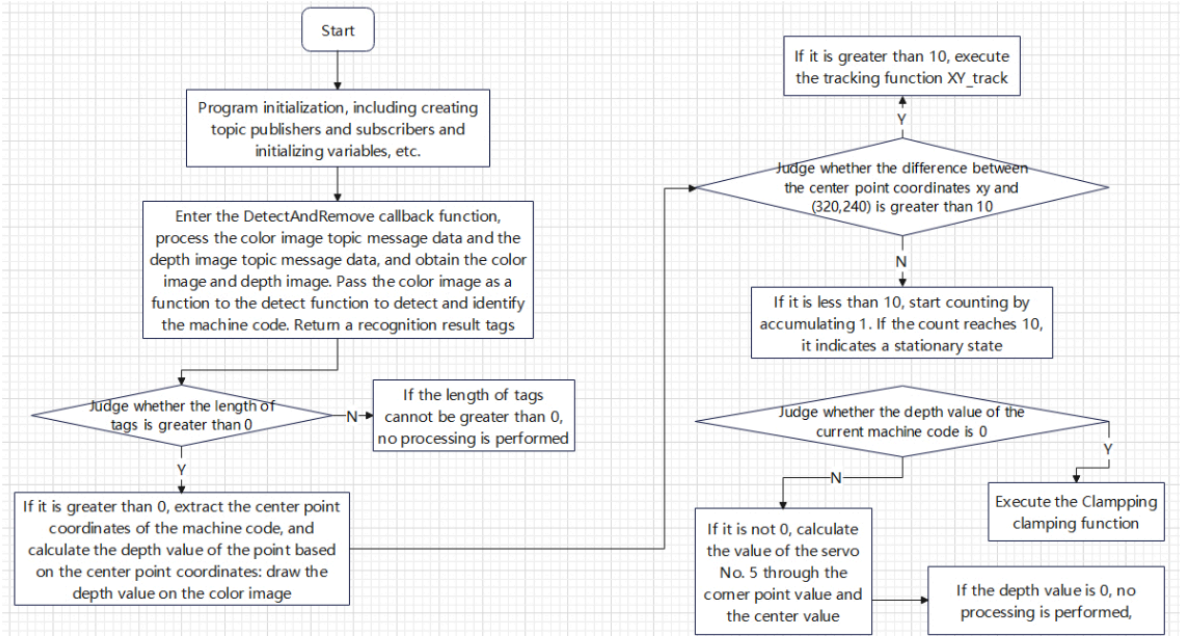
2.2. Operation

After the program is started, a 4cm*4cm machine code block appears in the image, and the robot arm will adjust its posture so that the center of the machine code coincides with the center of the image; slowly move the machine code, and the robot arm follows the movement of the machine code and constantly adjusts its posture. After waiting for the center of the machine code 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 machine code and the base is less than 30cm, it means that the depth value of the center coordinate of the machine code is valid and within the gripping range of the robot arm. The buzzer will sound, and then the robot arm will change its posture to grip the machine code according to the position of the machine code; 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 machine code and the base is less than 30cm (greater than 18cm), it is necessary to move the machine code back and forth again so that it can track again and then stop to meet the gripping conditions.



3. Program flow chart

apriltag_follow.py



4. Core code analysis

Code path:

```
/home/jetson/dofbot_pro_ws/src/dofbot_pro_apriltag/dofbot_pro_apriltag/apriltag_follow.py
```

Import necessary libraries,

```
import rclpy
```

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

from dofbot_pro_apriltag.vutils import draw_tags
from dt_apriltags import Detector
from dofbot_pro_apriltag.Dofbot_Track import *

encoding = ['16UC1', '32FC1']
```

Initialize program parameters, create publishers, subscribers, etc.

```
def __init__(self):
    super().__init__('apriltag_tracking')
    #Create a tracking gripper object
    self.dofbot_tracker = DofbotTrack()
    #Create a machine code object
    self.at_detector = Detector(searchpath=['apriltags'],
        families='tag36h11',
        nthreads=8,
        quad_decimate=2.0,
        quad_sigma=0.0,
        refine_edges=1,
        decode_sharpening=0.25,
        debug=0)
    #Counting variable, used to record the number of times the condition is met
    self.cnt = 0
    #The depth value of the current machine code center
    self.cur_distance = 0.0
    #Create a bridge for converting color and depth image topic message data to
image data
    self.rgb_bridge = CvBridge()
    self.depth_bridge = CvBridge()
    # ROS2 subscriber (message synchronization)
    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.DetectAndRemove)
```

Mainly look at the DetectAndRemove callback function,

```
def DetectAndRemove(self,color_frame,depth_frame):
    #Receive a color image topic message and convert the message data into image
data
```

```
    rgb_image = self.rgb_bridge.imgmsg_to_cv2(color_frame,'rgb8')
    result_image = np.copy(rgb_image)
    #Receive a 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 = cv2.resize(depth_image, (640, 480))
    depth_image_info = frame.astype(np.float32)
    #Call the detect function and pass in parameters.
    '''
    cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY): Converts an RGB image to a
grayscale image for label detection.
    False: Indicates that the label's pose is not estimated.
    None: Indicates that no camera parameters are provided, and only simple
detection may be performed.
    0.025: It may be the set label size (usually in meters), which is used to
help the detection algorithm determine the size of the label.
    Returns a detection result, including information such as the location, ID,
and bounding box of each label.
    '''
    tags = self.at_detector.detect(cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY),
False, None, 0.025)
    # Sort the tags in tags, not a necessary step
    tags = sorted(tags, key=lambda tag: tag.tag_id)
    # Call the draw_tags function, which is used to draw the information related
to the recognized machine code on the color image, including corner points,
center points and id values
    draw_tags(result_image, tags, corners_color=(0, 0, 255), center_color=(0,
255, 0))
    # Determine the length of the detection result tags. If it is greater than 0,
it means that the machine code has been detected
    if len(tags) > 0 :
        #print("tag: ",tags)
        #Get the center coordinates of the machine code
        center_x, center_y = tags[0].center
        #Calculate the depth value of the center coordinates of the machine code
        self.cur_distance = depth_image_info[int(center_y),int(center_x)]/1000.0
        dist = round(self.cur_distance,3)
        dist = 'dist: ' + str(dist) + 'm'
        #Draw the depth value of the center point on the color image
        cv2.putText(result_image, dist, (30, 30), cv2.FONT_HERSHEY_SIMPLEX, 1.0,
(255, 0, 0), 2)
        print("cur_distance: ",self.cur_distance)
        #Judge if the center coordinates of the machine code and the image center
(320, 240) are greater than 2, that is, they are not within the acceptable range.
Execute the tracking program and adjust the state of the robot arm to make the
center value of the machine code within the acceptable range
        if abs(center_x-320) >2 or abs(center_y-240)>2:
            #Execute the tracking program, and input the center value of the
current machine code
            self.dofbot_tracker.XY_track(center_x,center_y)
        print("======================================")
        #If the coordinates of the center point of the machine code and the
center point of the image (320, 240) are less than 2, it can be considered that
the center value of the machine code is in the middle of the image
        if abs(center_x-320) <2 and abs(center_y-240)<2:
            print("center_x: ",center_x)
```

```
            print("center_y: ",center_y)
            #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!")
                print("last_joint: ",self.dofbot_tracker.cur_joints)
                #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
                    vx = int(tags[0].corners[0][0]) - int(center_x)
                    vy = int(tags[0].corners[0][1]) - int(center_y)
                    angle_radians = math.atan2(vy, vx)
                    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)
                    set_joint5 = compute_angle*2
                    if set_joint5>135:
                        print("----------------------------------------")
                        self.dofbot_tracker.set_joint5 = set_joint5 - 90
                    else:
                        self.dofbot_tracker.set_joint5 = set_joint5
                    #Execute the clamping program, calling the Clamping function
of the created dofbot_tracker object. The input parameters are the center value
of the center value machine code and the depth value of the center point.

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

    result_image = cv2.cvtColor(result_image, cv2.COLOR_RGB2BGR)
    cv2.imshow("result_image", result_image)
    key = cv2.waitKey(1)
```

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. The location of the Dofbot_Track

```
/home/jetson/dofbot_pro_ws/src/dofbot_pro_apriltag/dofbot_pro_apriltag/Dofbot_Tra
ck.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
    print("pose_T: ",pose_T)
    #Call the inverse IK algorithm. 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. The values ••of
the 6 servos are obtained through IK calculation
    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 actual conditions
    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, which 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] = 30
        print("compute_joints: ",joints)
        #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 to move to
the gripping point
        if dist>0.18 and dist<0.30:
            self.Buzzer()
            print("Clamp Mode.")
            #Execute the pubTargetArm function and pass the calculated joints
value as a parameter
            self.pub_arm(joints)
            time.sleep(3.5)
            #Execute the clamping action and the moving and placing actions
            self.move()
        else:
```

```
            print("Too far or too close.")
    except Exception:
        rospy.loginfo("run error")
```