Desktop tracking and gripping machine code

1. Content Description

This function allows the program to capture an image through the camera, recognize the machine code on the desktop, and then slowly move the machine code on the desktop. The robot, in conjunction with the chassis and robotic arm, will track the machine code. After stopping the machine code, the robot adjusts the robotic arm back to its initial position and adjusts the chassis so that the center of the machine code appears in the center of the image. After the adjustment is completed, the robot can choose to continue tracking or clamp the machine code on the desktop using a button.

This section requires entering commands in the terminal. The terminal you open depends on your motherboard type. This lesson uses the Raspberry Pi 5 as an example. For Raspberry Pi and Jetson-Nano boards, you need to open a terminal on the host computer and enter the command to enter the Docker container. Once inside the Docker container, enter the commands mentioned in this section in the terminal. For instructions on entering the Docker container from the host computer, refer to this product tutorial **[Configuration and Operation Guide]--[Enter the Docker (Jetson Nano and Raspberry Pi 5 users, see here)]**.

Simply open the terminal on the Orin motherboard and enter the commands mentioned in this section.

2. Program startup

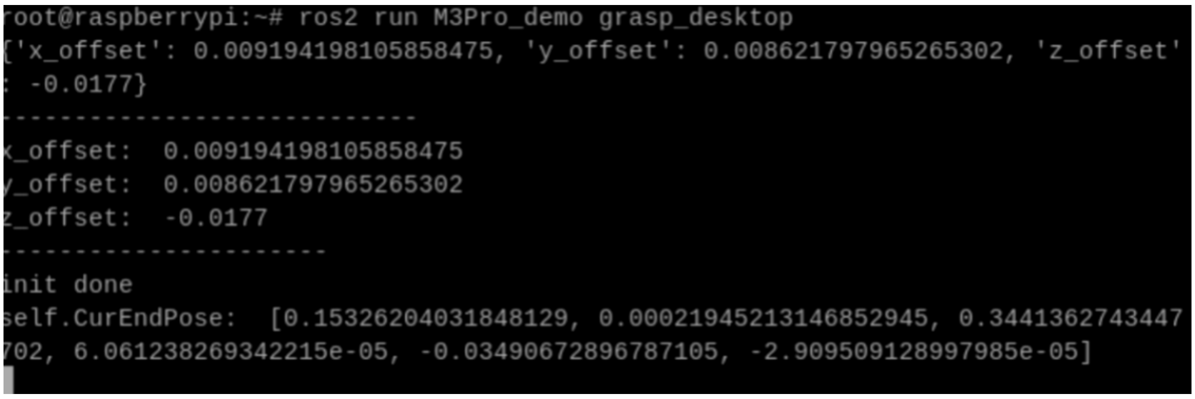
First, open the terminal and enter the following command to start the robot arm solver and camera driver,

```
ros2 launch M3Pro_demo camera_arm_kin.launch.py
```

Then, open another terminal and enter the following command to start the robotic arm gripping program:

```
ros2 run M3Pro_demo grasp_desktop
```

After running, it is shown as follows:

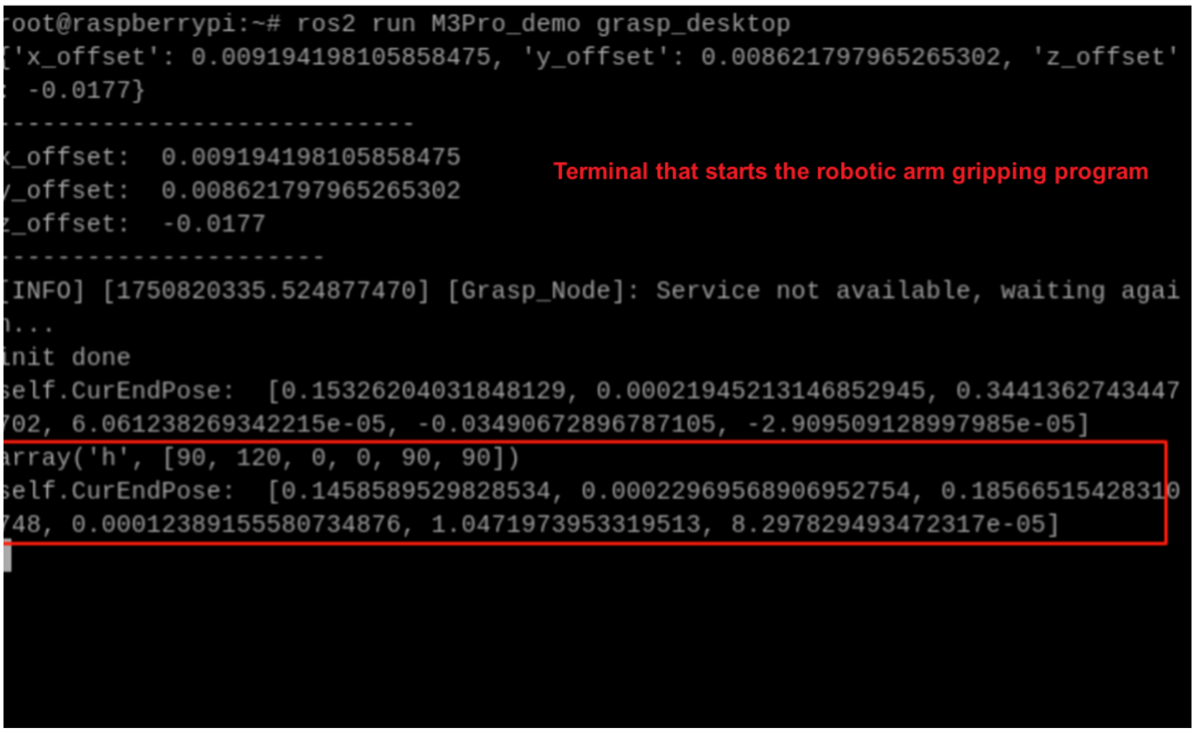


```
root@raspberrypi:~# ros2 run M3Pro_demo grasp_desktop
{'x_offset': 0.009194198105858475, 'y_offset': 0.008621797965265302, 'z_offset'
: -0.0177}
----------------------------
x_offset:  0.009194198105858475
y_offset:  0.008621797965265302
z_offset:  -0.0177
----------------------
init done
self.CurEndPose:  [0.15326204031848129, 0.00021945213146852945, 0.3441362743447
702, 6.061238269342215e-05, -0.03490672896787105, -2.909509128997985e-05]
```

Then enter the following command in the third terminal to start the desktop tracking and grabbing machine code program,

```
ros2 run M3Pro_demo apriltag_track_desktop
```

After starting this command, the second terminal should receive the current angle topic information sent in one frame and calculate the current posture once, as shown in the figure below.



```
oot@raspberrypi:~# ros2 run M3Pro_demo grasp_desktop
'x_offset': 0.009194198105858475, 'y_offset': 0.008621797965265302, 'z_offset'
 -0.0177}
---------------------------
_offset:  0.009194198105858475
_offset:  0.008621797965265302
_offset:  -0.0177
---------------------
INFO] [1750820335.524877470] [Grasp_Node]: Service not available, waiting agai
...
nit done
elf.CurEndPose:  [0.15326204031848129, 0.00021945213146852945, 0.3441362743447
02, 6.061238269342215e-05, -0.03490672896787105, -2.909509128997985e-05]
rray('h', [90, 120, 0, 0, 90, 90])
elf.CurEndPose:  [0.1458589529828534, 0.00022969568906952754, 0.18566515428310
48, 0.00012389155580734876, 1.0471973953319513, 8.297829493472317e-05]
```

If the current angle information is not received and the current posture is not calculated, the gripping posture will be inaccurate when the coordinate system is converted. Therefore, you need to close the sorting desktop tracking gripping robot code program with ctrl c and restart the desktop tracking gripping robot code program until the robot arm gripping program obtains the current angle information and calculates the current end position.

The first time you run the program, it will enter tracking mode. Slowly move the robot code on the table. The chassis and robotic arm will simultaneously track the robot code, achieving coordinated tracking. When you stop moving the robot code, the program will control the robotic arm to return to the recognition posture and move the chassis to center the robot code in the image. You can choose between the following modes: press the following button to select the mode.

- M key or M key: Tracking mode. In this mode, continue to slowly move the machine code wooden block on the desktop, and the robotic arm and chassis will track the machine code wooden block;
- Spacebar: Clamping mode. In this mode, the program will control the lower claw of the robotic arm to clamp the machine code wooden block and place it in the set position. Finally, the robotic arm returns to its initial posture.

After completing the first clamping, you need to press the m or M key to enter the tracking mode for the second tracking.

3. Core code analysis

Program code path:

- Raspberry Pi and Jetson-Nano board

  The program code is in the running docker. The path in docker is /root/yahboomcar_ws/src/M3Pro_demo/M3Pro_demo/ apriltag_track_desktop.py

- Orin Motherboard

  The program code path is /home/jetson/yahboomcar_ws/src/M3Pro_demo/M3Pro_demo/apriltag_track_desktop.py

Import the necessary library files,

```
import cv2
import os
import numpy as np
import message_filters
from M3Pro_demo.vutils import draw_tags
from M3Pro_demo.compute_joint5 import *
from dt_apriltags import Detector
from cv_bridge import CvBridge
import cv2 as cv
from arm_interface.srv import ArmKinemarics
from arm_interface.msg import AprilTagInfo,CurJoints
from arm_msgs.msg import ArmJoints
from std_msgs.msg import Float32,Bool,Int16,UInt16
import time
import transforms3d as tfs
import tf_transformations as tf
import yaml
import math
from rclpy.node import Node
import rclpy
from message_filters import Subscriber,
TimeSynchronizer,ApproximateTimeSynchronizer
from sensor_msgs.msg import Image
from geometry_msgs.msg import Twist
from M3Pro_demo.PID import *
import threading
```

Program initialization and creation of publishers and subscribers,

```
def __init__(self, name):
    super().__init__(name)
    self.init_joints = [90, 120, 0, 0, 90, 90]
    self.cur_joints = self.init_joints
    self.rgb_bridge = CvBridge()
    self.depth_bridge = CvBridge()
    self.pubPos_flag = False
    self.pr_time = time.time()
    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)

    self.CurEndPos = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    self.camera_info_K = [477.57421875, 0.0, 319.3820495605469, 0.0,
477.55718994140625, 238.64108276367188, 0.0, 0.0, 1.0]
    self.EndToCamMat = np.array([[ 0 ,0 ,1 ,-1.00e-01],
                                   [-1  ,0 ,0  ,0],
                                   [0  ,-1  ,0 ,4.82000000e-02],
                                   [ 0.00000000e+00 , 0.00000000e+00 ,
0.00000000e+00 , 1.00000000e+00]])

    self.pos_info_pub = self.create_publisher(AprilTagInfo,"PosInfo",1)
    self.CmdVel_pub = self.create_publisher(Twist,"cmd_vel",1)
    self.sub_grasp_status =
self.create_subscription(Bool,"grasp_done",self.get_graspStatusCallBack,100)
    self.TargetAngle_pub = self.create_publisher(ArmJoints, "arm6_joints", 10)
    self.TargetJoint5_pub = self.create_publisher(Int16, "set_joint5", 10)
    self.rgb_image_sub = Subscriber(self, Image, '/camera/color/image_raw')
    self.depth_image_sub = Subscriber(self, Image, '/camera/depth/image_raw')
    self.client = self.create_client(ArmKinemarics, 'get_kinemarics')
    self.pub_cur_joints = self.create_publisher(CurJoints,"Curjoints",1)
    self.pub_beep = self.create_publisher(UInt16, "beep", 10)
    self.get_current_end_pos()
    self.pubSix_Arm(self.init_joints)
    self.pubCurrentJoints()
    self.ts = ApproximateTimeSynchronizer([self.rgb_image_sub,
self.depth_image_sub], 1, 0.5)
    self.ts.registerCallback(self.callback)

    self.x_offset = offset_config.get('x_offset')
    self.y_offset = offset_config.get('y_offset')
    self.z_offset = offset_config.get('z_offset')
    self.done_flag = True
    self.start = 0.0
    self.scale = 1000
    self.joint5 = Int16()
    #Define the flag of chassis movement. If the value is True, it means entering
tracking mode. The program controls the chassis movement to track the machine
code block.
    self.move_flag = True
    self.RemovePID = (60, 0, 20) #60 0 20
    self.PID_init()

    self.target_servox=90
    self.target_servoy=180
    #Define and initialize the PID of the robot tracking
    self.xservo_pid = PositionalPID(1, 0.4, 0.2) # 0.5 0.2 0.1
    self.yservo_pid = PositionalPID(0.5, 0.2, 0.1)
    self.y_out_range = False
    self.x_out_range = False
    self.a = 0
    self.b = 0
    #The distance to move left and right
    self.recover_offset = 0.0
    # Left and right translation speed
    self.recover_lin_y = 0.12
    #Define the flag that indicates the recovery mode is complete. If the value
is True, it means that the robot arm has returned to its initial posture in
tracking mode.
    self.recover_done = False
```

callback image topic callback function,

```
def callback(self,color_frame,depth_frame):

    #rgb_image
    rgb_image = self.rgb_bridge.imgmsg_to_cv2(color_frame,'rgb8')
    result_image = np.copy(rgb_image)
    result_image = cv.resize(result_image, (640, 480))
    #depth_image
    depth_image = self.depth_bridge.imgmsg_to_cv2(depth_frame, encoding[1])
    depth_to_color_image = cv2.applyColorMap(cv2.convertScaleAbs(depth_image,
alpha=1.0), cv2.COLORMAP_JET)
    frame = cv.resize(depth_image, (640, 480))
    depth_image_info = frame.astype(np.float32)
    tags = self.at_detector.detect(cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY),
False, None, 0.025)
    #tags = sorted(tags, key=lambda tag: tag.tag_id) # It seems that the tags are
sorted in ascending order, so there is no need to sort them manually.
    draw_tags(result_image, tags, corners_color=(0, 0, 255), center_color=(0,
255, 0))

    key = cv2.waitKey(1)
    if key ==32:
        self.pubPos_flag = True
    #If you press the m or M key, you will enter tracking mode and change the
values of self.move_flag, self.recover_done, and self.done_flag
    elif key == 77 or key == 109:
        self.move_flag = True
        self.recover_done = False
        self.done_flag = True


    if len(tags) > 0 :
        center_x, center_y = tags[0].center
        cux, cuy = tags[0].center
        #If the center point of the machine code is not in the middle of the
image and the current mode is tracking mode
        if (abs(center_x-320) >10 or abs(center_y-300)>10) and self.move_flag ==
True:
            print("adjusting.")
            #base_track = threading.Thread(target=self.remove_obstacle, args=
(center_x, center_y,))
            #Start the thread and execute the robot tracking function
            arm_track = threading.Thread(target=self.XY_track, args=(center_x,
center_y,))
            arm_track.start()
            arm_track.join()
            #Start the thread and execute the chassis tracking function
            base_track = threading.Thread(target=self.remove_obstacle, args=
(center_x, center_y,))
            base_track.start()
            base_track.join()
            self.start = time.time()


        #If the center of the machine code is in the middle of the image and
completed
        elif abs(center_x-320) <10 and abs(center_y-300)<10 and self.pubPos_flag
== False and self.done_flag==True and self.recover_done == False:
            self.pubVel(0,0,0)
            #Start the thread and calculate the current pose coordinates of the
end of the robotic arm
            compute_end = threading.Thread(target=self.get_current_end_pos,
args=())
            compute_end.start()
            compute_end.join()
            z = depth_image_info[int(center_y),int(center_x)]/1000
            #Calculate the position of the current machine code block in the
world coordinate system
            dist = self.compute_dist(int(center_x),int(center_y),z)
            # Responsible for the offset in the y direction
            self.recover_offset = dist[1]
            #print("y_offset = ",dist[1])
            recover_time = abs(dist[1]/0.1)
            arm_run_time = int(recover_time*1000)
            #print("recover_time:  ",recover_time)
            #If the current static time exceeds 0.5 seconds, the recovery action
is performed: the robotic arm returns to the initial posture and the chassis is
translated according to the value of self.recover_offset.
            if time.time() - self.start >0.5:
                print("---------------")
                self.move_flag = False
                #self.pubPos_flag = True
                if abs(self.cur_joints[0] - 90)>2:
                    print("Recoverring.")
                    self.done_flag = False
                    print("self.recover_offset: ",self.recover_offset)
                    #The robotic arm returns to its initial posture
                    self.pubSix_Arm(self.init_joints,runtime=arm_run_time)
                    self.cur_joints = self.init_joints
                    #Start the thread and execute the function of chassis
translation
                    recover_ = threading.Thread(target=self.recover_move, args=
(recover_time,))
                    recover_.start()
                    recover_.join()
                else:
                    print("Recover done.")
                    self.recover_done = True
        #If the recovery action is completed and the robot arm is enabled, then
the center coordinate value of the machine code and the corresponding depth
information are obtained, and the machine code position information topic is
published after sorting.
        if self.recover_done == True and self.pubPos_flag == True:
            print("Next staus is grasp.")
            self.pubPos_flag = False
            tag = AprilTagInfo()
            cx, cy = tags[0].center
            tag.id = tags[0].tag_id
            tag.x = cx
            tag.y = cy
            print("cx: ",cx)
            print("cy: ",cy)
            tag.z = depth_image_info[int(tag.y),int(tag.x)]/1000
            vx = int(tags[0].corners[0][0]) - int(tags[0].corners[1][0])
            vy = int(tags[0].corners[0][1]) - int(tags[0].corners[1][1])
            target_joint5 = compute_joint5(vx,vy)
            print("target_joint5: ",target_joint5)
```

```
                self.joint5.data = int(target_joint5)
                if tag.z!=0:
                    self.recover_done = False
                    self.TargetJoint5_pub.publish(self.joint5)
                    #Buzzer sounds once
                    self.Beep_Loop()
                    self.pos_info_pub.publish(tag)
                else:
                    print("Invalid distance.")

        else:
            self.pubVel(0,0,0)
        result_image = cv2.cvtColor(result_image, cv2.COLOR_RGB2BGR)
        cur_time = time.time()
        fps = str(int(1/(cur_time - self.pr_time)))
        self.pr_time = cur_time
        cv2.putText(result_image, fps, (10, 30), cv2.FONT_HERSHEY_SIMPLEX, 1, (0,
255, 0), 2)
        cv2.imshow("result_image", result_image
```