Multimodal Large Model + Robotic Arm Gripping（Text Version）

Before running the function, you need to close the App and large programs. For the closing method, refer to [4. Preparation] - [1. Manage APP control services].

1. Function Description

After the program runs, input robotic arm gripping action commands through the terminal. Based on the type of object to be gripped, the large model will start external programs to control the robotic arm to grip the target object.

2. Startup

Users with Jetson-Nano mainboard version need to enter the docker container first and then input the following command. Users with Orin motherboard can directly open the terminal and input the following command:

```
ros2 launch largemodel largemodel_control.launch.py text_chat_mode:=True
```

Then open a second terminal and input the following command:

```
ros2 run text_chat text_chat
```

The robotic arm gripping has the following action commands:

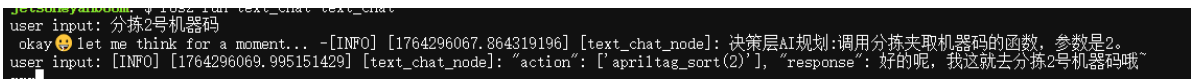
- **Grip by machine code ID**: Sort machine codes 1/2/3/4
- **Grip by machine code height**: Remove machine codes with height higher than x cm (where x represents height), for example: Remove machine codes with height higher than 5 cm
- **Grip by color block height**: Remove y color blocks with height higher than x cm (where x represents height, y represents color with values red/green/yellow/blue), for example: Remove red blocks with height higher than 5 cm
- **Grip other objects**: Grip the small yellow duck on the desktop, grip the blue cuboid on the desktop

2.1. Grip by Machine Code ID

2.1.1. Startup

Input in the terminal that started text_chat:

```
Sort machine code 2
```



Then two terminals will open, starting the machine code recognition program and gripping program respectively. After recognizing machine code 2, the gripping program will control the robotic arm to descend and grip machine code 2, then place it at the set position, and finally return to the machine code recognition posture to prepare for the next gripping. This continues until no more machine code 2 is recognized, then it reports back to the large model and the task ends.

2.1.2. Task Planning

Plan the action function apriltag_sort(2), where the parameter 2 represents the ID of the machine code to be gripped.

2.1.3. Core Code Analysis

apriltag_sort function, source code located at:
`LargeModel_ws/src/largemodel/largemodel/action_service.py`

```
def apriltag_sort(self, target_id):  # Grip machine code
    #self.check_apriltag_sort()
    target_idf = float(target_id)
    #Start gripping program
    cmd1 = "ros2 run largemodel_arm grasp
    #Start machine code recognition program, passing parameter target_id
    cmd2 = f"ros2 run largemodel_arm apriltag_sort --ros-args -p target_id:=
{target_idf:.1f}"
    subprocess.Popen(
        [
            "gnome-terminal",
            "--title=grasp_desktop_apritag",
            "--",
            "bash",
            "-c",
            f"{cmd1}; exec bash",
        ]
    )
    subprocess.Popen(
        [
            "gnome-terminal",
            "--title=apriltag_sort",
            "--",
            "bash",
            "-c",
            f"{cmd2}; exec bash",
        ]
    )
```

Gripping program grasp, source code path at:
`LargeModel_ws/src/largemodel_arm/largemodel_arm/grasp.py`

```
#Create subscriber, subscribe to position information topic
self.sub = self.create_subscription(AprilTagInfo,'PosInfo',self.pos_callback,1)
#Position information topic callback function
def pos_callback(self,msg):
    pos_x = msg.x
    pos_y = msg.y
    pos_z = msg.z
    self.cur_tagId = msg.id
    if pos_z!=0.0:
        print("xyz id : ",pos_x,pos_y,pos_z,self.cur_tagId)
        #Start coordinate system transformation
        camera_location = self.pixel_to_camera_depth((pos_x,pos_y),pos_z)
        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)
        #Convert rotation transformation matrix to xyz and Euler angles to get
target pose
        pose_T, pose_R = self.mat_to_xyz_euler(WorldPose)
        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)
        if self.grasp_flag == True :
            print("Take it now.")
            self.grasp_flag = False
            #Pass target position information to grasp function for inverse
kinematics to get each robotic arm angle value
            threading.Thread(target=self.grasp, args=(pose_T,)).start()
#Call inverse kinematics service, pass target position, get values for servos 1-
5
def grasp(self,pose_T):
    print("--------------------------------------------------")
    print("pose_T: ",pose_T)

    request = Kinemarics.Request()
    request.tar_x = pose_T[0]
    request.tar_y = pose_T[1] + 0.01
    request.tar_z = pose_T[2] + 0.01
    request.kin_name = "ik"
    request.roll = -1.0 #((2.5*request.tar_y*100)-207.5)*(math.pi / 180)

    # print("calcutelate_request: ",request)

    try:
        future = self.client.call_async(request)
        rclpy.spin_until_future_complete(self, future, timeout_sec=5.0)
        response = future.result()
        print("calcutelate_response: ",response)
        joints = [0.0, 0.0, 0.0, 0.0, 0.0,0.0]
        joints[0] = response.joint1 #response.joint1
        joints[1] = response.joint2
        joints[2] = response.joint3
```

Machine code recognition program apriltag_sort, source code path at: `LargeModel_ws/src/largemodel_arm/largemodel_arm/apriltag_sort.py`,

```
#Get target sorting id through command line arguments
self.declare_parameter('target_id', 0.0)
self.TargetID =
int(self.get_parameter('target_id').get_parameter_value().double_value)

#Create publisher, publish position information topic
self.pos_info_pub = self.create_publisher(AprilTagInfo, "PosInfo",
qos_profile=10)

#Image callback function
def TagDetect(self,color_frame,depth_frame):
    #rgb_image
    rgb_image = self.rgb_bridge.imgmsg_to_cv2(color_frame,'rgb8')
    result_image = np.copy(rgb_image)
```

```
    #depth_image
    depth_image = self.depth_bridge.imgmsg_to_cv2(depth_frame, encoding[1])
    depth_to_color_image = cv.applyColorMap(cv.convertScaleAbs(depth_image,
alpha=1.0), cv.COLORMAP_JET)
    frame = cv.resize(depth_image, (640, 480))
    depth_image_info = frame.astype(np.float32)
    #Machine code recognition
    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)
    draw_tags(result_image, tags, corners_color=(0, 0, 255), center_color=(0,
255, 0))
    key = cv2.waitKey(10)
    if self.count==True:
        if (time.time() - self.start_time)>10:
            self.pubPos_flag = True
            self.count = False
    if key == 32:
        self.pubPos_flag = True
    if len(tags) > 0 :
        for i in range(len(tags)):
            center_x, center_y = tags[i].center
            cur_id = tags[i].tag_id
            cv.circle(depth_to_color_image,(int(center_x),int(center_y)),1,
(255,255,255),10)
            #If current id is target sorting id, it means found, so
self.detect_flag value is True, and record current array index for subsequent
value retrieval
            if self.pubPos_flag == True and cur_id == self.TargetID:
                self.detect_flag = True
                print("Found the target.")
                self.index = i
                center_x, center_y = tags[self.index].center
                cv.circle(result_image, (int(center_x),int(center_y)), 10,
(0,210,255), thickness=-1)

                pos = AprilTagInfo()
                #Get xy coordinates of target machine code, center point depth
value and id
                pos.id = tags[self.index].tag_id
                pos.x = center_x
                pos.y = center_y
                pos.z = depth_image_info[int(center_y),int(center_x)]/1000
                vx = int(tags[self.index].corners[0][0]) -
int(tags[self.index].corners[1][0])
                vy = int(tags[self.index].corners[0][1]) -
int(tags[self.index].corners[1][1])
                target_joint5 = compute_joint5(vx,vy)
                if pos.z>0:
                    self.joint5.data = int(target_joint5)
                    self.TargetJoint5_pub.publish(self.joint5)
                    #Publish position information topic data message
                    self.pos_info_pub.publish(pos)
                    self.pubPos_flag = False
                else:
                    print("Invalid distance.")
        #Machine code not found
```

```
        if  self.detect_flag == False and self.TargetID!=0 and
self.pubPos_flag==True:
            self.pubPos_flag = False
            print("Did not find the target.")
            self.largemodel_arm_done_pub.publish(
String(data='apriltag_sort_done'))
            self.TargetID = 0
```

2.2. Grip by Machine Code Height

2.2.1. Startup

Input in the terminal that started text_chat:

```
Remove machine codes with height higher than 3 cm
```

Then two terminals will open, starting the machine code recognition height calculation program and gripping program respectively. After recognizing machine codes with height higher than 3 cm, the gripping program will control the robotic arm to descend and grip the abnormal height machine codes, then place them at the set position, and finally return to the machine code recognition posture to prepare for the next gripping. This continues until no more abnormal height machine codes are recognized, then it reports back to the large model and the task ends.

2.2.2. Task Planning

Plan the action function apriltag_remove_higher(30.0), where the parameter 30 represents the height threshold. Machine codes with height higher than this value will be removed, unit is millimeters mm.

2.2.3. Core Code Analysis

apriltag_remove_higher function, source code located at:
`LargeModel_ws/src/largemodel/largemodel/action_service.py`

```
def apriltag_remove_higher(self, target_high):  # Remove machine codes of
specified height
    target_highf = float(target_high) / 100
    #Robotic arm gripping program
    cmd1 = "ros2 run largemodel_arm grasp"
    #Machine code height calculation program, passing parameter target_high,
which is the height threshold
    cmd2 = f"ros2 run largemodel_arm apriltag_remove_higher --ros-args -p
target_high:={target_highf:.2f}"
    subprocess.Popen(
        [
            "gnome-terminal",
            "--title=grasp_desktop_remove",
            "--",
            "bash",
            "-c",
            f"{cmd1}; exec bash",
        ]
```

```
    )
    subprocess.Popen(
        [
            "gnome-terminal",
            "--title=apriltag_remove_higher",
            "--",
            "bash",
            "-c",
            f"{cmd2}; exec bash",
        ]
    )
```

The grasp program was analyzed in the previous section and will not be repeated here. Now let's analyze the apriltag_remove_higher program, whose source code is located at: `LargeModel_ws/src/largemodel_arm/largemodel_arm/apriltag_remove_higher.py`

```
#Get height threshold through command line arguments
self.declare_parameter('target_high', 0.0)
self.Target_height =
self.get_parameter('target_high').get_parameter_value().double_value * 100
#Height calculation function, with three parameters xyz, xy represents the center
point coordinates of the machine code, z is the depth information of the center
point
def compute_heigh(self,x,y,z):
    #Start coordinate system transformation
    camera_location = self.pixel_to_camera_depth((x,y),z)
    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)
    #Convert rotation transformation matrix to xyz and Euler angles to get target
pose, where pose_T[2] represents the height value
    pose_T, pose_R = self.mat_to_xyz_euler(WorldPose)
    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
    return pose_T
#Get current machine code height
compute_height = round(pose[2],4)*1000
heigh = 'heigh: ' + str(compute_height) + 'mm'
cv.putText(result_image, heigh, (int(cx)+5, int(cy)-15),
cv.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
#If current height value is greater than height threshold, change
self.detect_flag value to true, and set self.compute_height to false, this is to
avoid recalculating height when gripping action is completed
if compute_height > self.Target_height and self.pubPos_flag == True and
self.compute_height == True:
    print("Found the target.")
    self.found_cnt = self.found_cnt + 1
    self.compute_height = False
    self.index = i
    self.detect_flag = True
```

2.3. Grip by Color Block Height

2.3.1. Startup

Input in the terminal that started text_chat:

```
Remove green blocks with height higher than 3 cm
```

Then two terminals will open, starting the color block recognition height calculation program and gripping program respectively. After recognizing green blocks with height higher than 3 cm, the gripping program will control the robotic arm to descend and grip these blocks, then place them at the set position, and finally return to the recognition posture to prepare for the next gripping. This continues until no more abnormal height green blocks are recognized, then it reports back to the large model and the task ends.

2.3.2. Task Planning

Plan the action function color_remove_higher('green',30.0), 'green' indicates the color block to be sorted is green blocks, the parameter 30 represents the height threshold. Machine codes with height higher than this value will be removed, unit is millimeters mm.

2.3.3. Core Code Analysis

color_remove_higher function, source code located at:
`LargeModel_ws/src/largemodel/largemodel/action_service.py`

```
def color_remove_higher(self, color, target_high):
    arm_joints = [90, 110, 0, 0, 90, 0]
    self.pubSix_Arm(arm_joints)
    color = color.strip("'\"")  # Remove single and double quotes
    target_highf = float(target_high) / 10
    #Modify target_color value based on passed value for easy passing to
color_remove_higher
    if color == "red":
        target_color = float(1)
    elif color == "green":
        target_color = float(2)
    elif color == "blue":
        target_color = float(3)
    elif color == "yellow":
        target_color = float(4)
    else:
        self.get_logger().info(
            "Fatal ERROR:Incorrect color input,Does the AI output not meet
expectations?"
        )
        self.action_status_pub(
            "color_remove_higher_failed", color=color, target_high=target_high
        )
        return
    #Robotic arm gripping program
    cmd1 = "ros2 run largemodel_arm grasp"
    #Color block recognition and height calculation node, passing two parameters,
first is target_high height threshold, second is target_color color block color
    cmd2 = f"ros2 run largemodel_arm color_remove_higher --ros-args -p
target_high:={target_highf:.2f} -p target_color:={target_color:.1f}"
    subprocess.Popen(
```

```
        [
            "gnome-terminal",
            "--title=grasp_desktop_remove_color",
            "--",
            "bash",
            "-c",
            f"{cmd1}; exec bash",
        ]
    )
    subprocess.Popen(
        [
            "gnome-terminal",
            "--title=color_remove_higher",
            "--",
            "bash",
            "-c",
            f"{cmd2}; exec bash",
        ]
    )
```

The grasp program was analyzed before and will not be repeated here. Now let's analyze the color_remove_higher program, whose source code is located at:
`LargeModel_ws/src/largemodel_arm/largemodel_arm/color_remove_higher.py`

```
#Get target color block color through command line arguments
self.declare_parameter('target_color', 0.0)
self.target_color =
int(self.get_parameter('target_color').get_parameter_value().double_value)
print("Get self.target_color is ",self.target_color)
#Get height threshold through command line arguments
self.declare_parameter('target_high', 0.0)
self.target_high =
self.get_parameter('target_high').get_parameter_value().double_value * 1.0
print("Get self.target_high is ",self.target_high)

#In the process function, we determine self.hsv_range value based on
self.target_color value, by reading read_HSV to read color hsv file
elif self.Track_state == "identify":
    if self.target_color == 1:
        self.hsv_range = read_HSV(self.red_hsv_text)
        self.cur_color = "red"
        self.text_color = (0, 0, 255)


    elif self.target_color == 2:
        self.hsv_range = read_HSV(self.green_hsv_text)
        self.cur_color = "green"
        self.text_color = (0, 255, 0)


    elif self.target_color == 3:
        self.hsv_range = read_HSV(self.blue_hsv_text)
        self.cur_color = "blue"
        self.text_color = (255, 0, 0)

    elif self.target_color == 4:
        self.hsv_range = read_HSV(self.yellow_hsv_text)
```

```
        self.cur_color = "yellow"
        self.text_color = (255, 255, 0)

    else:
        self.Track_state = 'init'

if self.Track_state != 'init':
    if len(self.hsv_range) != 0:
        #Call external library function object_follow_list for color
recognition, get center point information of command color block
        rgb_img, binary, self.CX_list,self.CY_list,self.R_list_,self.corners=
self.color.object_follow_list(rgb_img, self.hsv_range)
#In callback function
if len(self.CX_list)>0 and self.done_flag==True and self.compute_height==True:
    for i in range(len(self.CX_list)):
        #Here inscribed circle radius judgment greater than 30 is to filter out
some small points
        if self.R_list_[i]>30:
            cx = int(self.CX_list[i])
            cy = int(self.CY_list[i])
            dist = depth_image_info[int(cy),int(cx)]/1000
            pose = self.compute_heigh(cx,cy,dist)
            compute_heigh = round(pose[2],2)*100
            heigh = 'heigh: ' + str(compute_heigh) + ' cm'
            cv.putText(result_frame, heigh, (int(cx)+10, int(cy)-25),
cv.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
            #If current color block height is greater than height threshold,
change self.detect_flag value to true, indicating found
            if  compute_heigh > self.target_high and self.pub_pos_flag == True :
                self.index = i
                self.detect_flag = True
                self.compute_height = False
                self.found_cnt = self.found_cnt + 1

            #If abnormal height color block is found and array index is not
None, then read the center point coordinates and center depth value of this color
block
            if self.detect_flag == True and self.index != None:
                cx = int(self.CX_list[self.index])
                cy = int(self.CY_list[self.index])
                vx = self.corners[0][0][0] - self.corners[1][0][0]
                vy = self.corners[0][0][1] - self.corners[1][0][1]
                target_joint5 = compute_joint5(vx,vy)
                dist = depth_image_info[int(cy),int(cx)]/1000
                cv.putText(result_frame, heigh, (int(cx)+10, int(cy)-25),
cv.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
                if dist!=0 and self.pub_pos_flag == True:
                    pos = AprilTagInfo()
                    pos.id = self.target_color
                    pos.x = float(cx)
                    pos.y = float(cy)
                    pos.z = float(dist)
                    if self.pub_pos_flag == True:
                        self.joint5.data = int(target_joint5)
                        self.TargetJoint5_pub.publish(self.joint5)
                        self.index = None
                        self.pub_pos_flag = False
                        self.done_flag = False
```

```
                        #Publish position information topic data of abnormal
height color block
                        self.pos_info_pub.publish(pos)
                        print("Publish the position.")
```

2.4. Grip Other Objects

2.4.1. Startup

Input in the terminal that started text_chat:

```
Grip the small yellow duck on the desktop
```

The program will first take a photo to determine the position of the small yellow duck, then open two terminals, starting the KCF tracking positioning program and object gripping program respectively. After calculation, the robotic arm will descend to grip the small yellow duck and finally return to the initial posture.

2.4.2. Task Planning

1. Call seehat() to check the position of the small yellow duck;
2. Call grasp_obj(x1,y1,x2,y2) gripping function, where x1,y1,x2,y2 represent the top-left and bottom-right corner coordinates of the small yellow duck's outer bounding box.

2.4.3. Core Code Analysis

The seehat() function was analyzed in the previous tutorial [2. Multimodal Visual Understanding] and will not be repeated here. Let's mainly look at the grasp_obj function, source code located at: `LargeModel_ws/src/largemodel/largemodel/action_service.py`

```
def grasp_obj(self, x1, y1, x2, y2):
    """
    Grip object
    x1,y1,x2,y2: Object outer bounding box coordinates
    """
    #Arm.Arm_serial_servo_write6(90,120,12,20,90,30,1000)
    #self.check_close_grasp_obj()
    #Object gripping program
    cmd1 = "ros2 run largemodel_arm KCF_Grap_Move"
    #KCF tracking positioning program
    cmd2 = "ros2 run largemodel_arm ALM_KCF_Tracker"
    # cmd3 = "ros2 run --prefix 'gdb -ex run --args' M3Pro_KCF
ALM_KCF_Tracker_Node"
    subprocess.Popen(
        [
            "gnome-terminal",
            "--title=ALM_KCF_Tracker",
            "--",
            "bash",
            "-c",
            f"{cmd2}; exec bash",
        ]
```

```
    )
    time.sleep(5.0) #Wait for ALM_KCF_Tracker to start up
    subprocess.Popen(
        [
            "gnome-terminal",
            "--title=grasp_desktop",
            "--",
            "bash",
            "-c",
            f"{cmd1}; exec bash",
        ]
    )
    time.sleep(2.0)
    if self.stack_flag == True:
        self.get_logger().info('Publish the stack_step topic...')
        step_ = Int16()
        step_.data = self.step
        self.step_pub.publish(step_)
        self.step = self.step + 1
    #Publish outer bounding box coordinate information topic data,
ALM_KCF_Tracker node will subscribe to this topic
    x1 = int(x1)
    y1 = int(y1)
    x2 = int(x2)
    y2 = int(y2)
    self.object_position_pub.publish(Int16MultiArray(data=[x1, y1, x2, y2]))
```

KCF_Grap_Move program, source code location at:
`LargeModel_ws/src/largemodel_arm/largemodel_arm/KCF_Grap_Move.py`

```
#Import library, this library's path is
LargeModel_ws/src/largemodel_arm/largemodel_arm/Dofbot_Track.py
from largemodel_arm.Dofbot_Track import *

#Create robotic arm tracking gripping object
self.dofbot_tracker = DofbotTrack()

#Create subscriber, subscribe to /pos_xyz topic
self.pos_sub = Subscriber(self,Position,'/pos_xyz')

#Callback function
def TrackAndGrap(self,position):
    center_x, center_y = position.x,position.y
    self.cur_distance = position.z
    if self.done == True:
        self.get_current_end_pos()
        self.get_logger().info('-*-*-*-*-*-*-*-*-*-*-*')
        print("self.CurEndPos: ",self.CurEndPos)
        #Call external library function Clamping

 self.dofbot_tracker.Clamping(center_x,center_y,self.cur_distance,self.CurEndPos
)
        self.done = False

#View Clamping function in Dofbot_Track.py
def Clamping(self,cx,cy,cz,CurEndPos):
    self.CurEndPos = CurEndPos
```

```
    #Start coordinate system transformation
    camera_location = self.pixel_to_camera_depth((cx,cy),cz)
    print("cz: ",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)
    #Convert rotation transformation matrix to xyz and Euler angles
    pose_T, pose_R = self.mat_to_xyz_euler(WorldPose)
    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)
    #Create inverse kinematics service object and assign the calculated pose_T
representing target position to it
    request = Kinemarics.Request()
    if pose_T[0]>0:
        request.tar_x = pose_T[0] +0.01
    else:
        request.tar_x = pose_T[0] - 0.01
    request.tar_z = pose_T[2]
    request.tar_y = pose_T[1] + request.tar_z*0.2
    request.kin_name = "ik"
    request.roll =  -1.0
    print("calcutelate_request: ",request)

    try:
        #Call inverse kinematics service, calculate to get servo values 1-4
        future = self.client.call_async(request)
        rclpy.spin_until_future_complete(self, future, timeout_sec=10.0)
        response = future.result()
        joints = [0.0, 0.0, 0.0, 0.0, 0.0,0.0]
        joints[0] = response.joint1 #response.joint1
        joints[1] = response.joint2
        joints[2] = response.joint3
        if response.joint4>90:
            joints[3] = 90.0
        else:
            joints[3] = response.joint4
        joints[4] = 90.0
        joints[5] = 20.0
        print("compute_joints: ",joints)
```

ALM_KCF_Tracker program, source code location at:
`LargeModel_ws/src/largemodel_arm/largemodel_arm/ALM_KCF_Tracker.py`

```
#Create publisher, publish object position information
self.pub_pos = self.create_publisher(Position, "/pos_xyz", 10)
#Create subscriber, subscribe to object outer bounding box information
self.xy_subscription =
self.create_subscription(Int16MultiArray,'corner_xy',self.GetXYCallback,qos_prof
ile=1)

#Callback function
def GetXYCallback(self,msg):
    print("msg: ",msg.data)
    print(msg.data[0])
```

```
    print(msg.data[1])
    print(msg.data[2])
    print(msg.data[3])
    #Assign self.Roi_init value to topic data message data
    self.Roi_init = (msg.data[0],msg.data[1],msg.data[2],msg.data[3])
    self.Track_state = 'identify'
    self.get_xy = True
    self.rect = [msg.data[0],msg.data[1],msg.data[2],msg.data[3]]

#In process function, initialize KCF based on self.Roi_init value
if self.gTracker_state == True:
    Roi = (self.Roi_init[0], self.Roi_init[1], self.Roi_init[2] -
self.Roi_init[0], self.Roi_init[3] - self.Roi_init[1])
    self.compute_roi = True
    print("Roi: ",Roi)
    self.gTracker = Tracker(tracker_type=self.tracker_type)
    self.gTracker.initWorking(rgb_img, Roi)
    self.gTracker_state = False
#KCF tracking, get object center point information
rgb_img, (targBegin_x, targBegin_y), (targEnd_x, targEnd_y) =
self.gTracker.track(rgb_img)
center_x = targEnd_x / 2 + targBegin_x / 2
center_y = targEnd_y / 2 + targBegin_y / 2
self.cx = center_x
self.cy = center_y
#In kcfTrack image callback function
#Assign calculated center point coordinates to center_x, center_y
center_x, center_y = self.cx,self.cy
cv2.circle(depth_to_color_image,(int(center_x),int(center_y)),1,
(255,255,255),10)
#Calculate center point depth distance information
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'
cv.putText(result_frame, dist,  (30, 30), cv.FONT_HERSHEY_SIMPLEX, 1.0, (255, 0,
0), 2)
pos = Position()
pos.x = center_x
pos.y = center_y
pos.z = self.cur_dis  tance
Track_pos = Position()
Track_pos.x = center_x
Track_pos.y = center_y
#Publish position information topic data
self.pub_track_pos.publish(Track_pos)
```