KCF tracking and grabbing objects

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 frame the object to be tracked and clamped, and press the space bar to start tracking. The robot arm will adjust its posture so that the center of the tracked object 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, and the object is within the set clamping range at this time, the buzzer will sound, and then the robot arm will adjust its posture to clamp the object. After gripping, place it at the set position and then return to the initial posture of the robot arm.

2. Start and operation

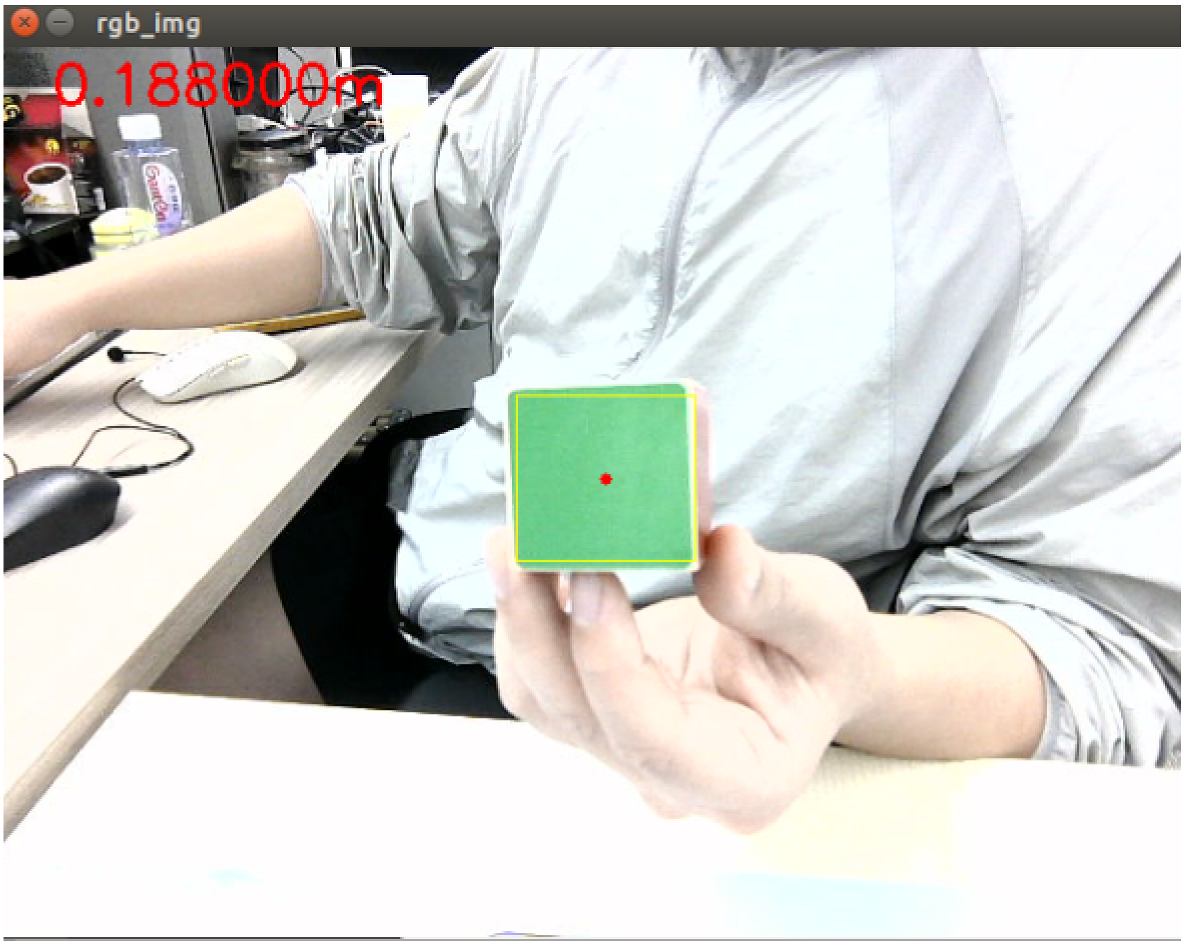
2.1. Start command

Enter the following command in the terminal to start.

```
#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 KCF program
rosrun dofbot_pro_KCF KCFTracker_node
#Start the tracking and grabbing program
rosrun dofbot_pro_KCF KCF_TrackAndGrap.py
#Start the calculation of the gripper width program
rosrun dofbot_pro_KCF compute_width.py
```

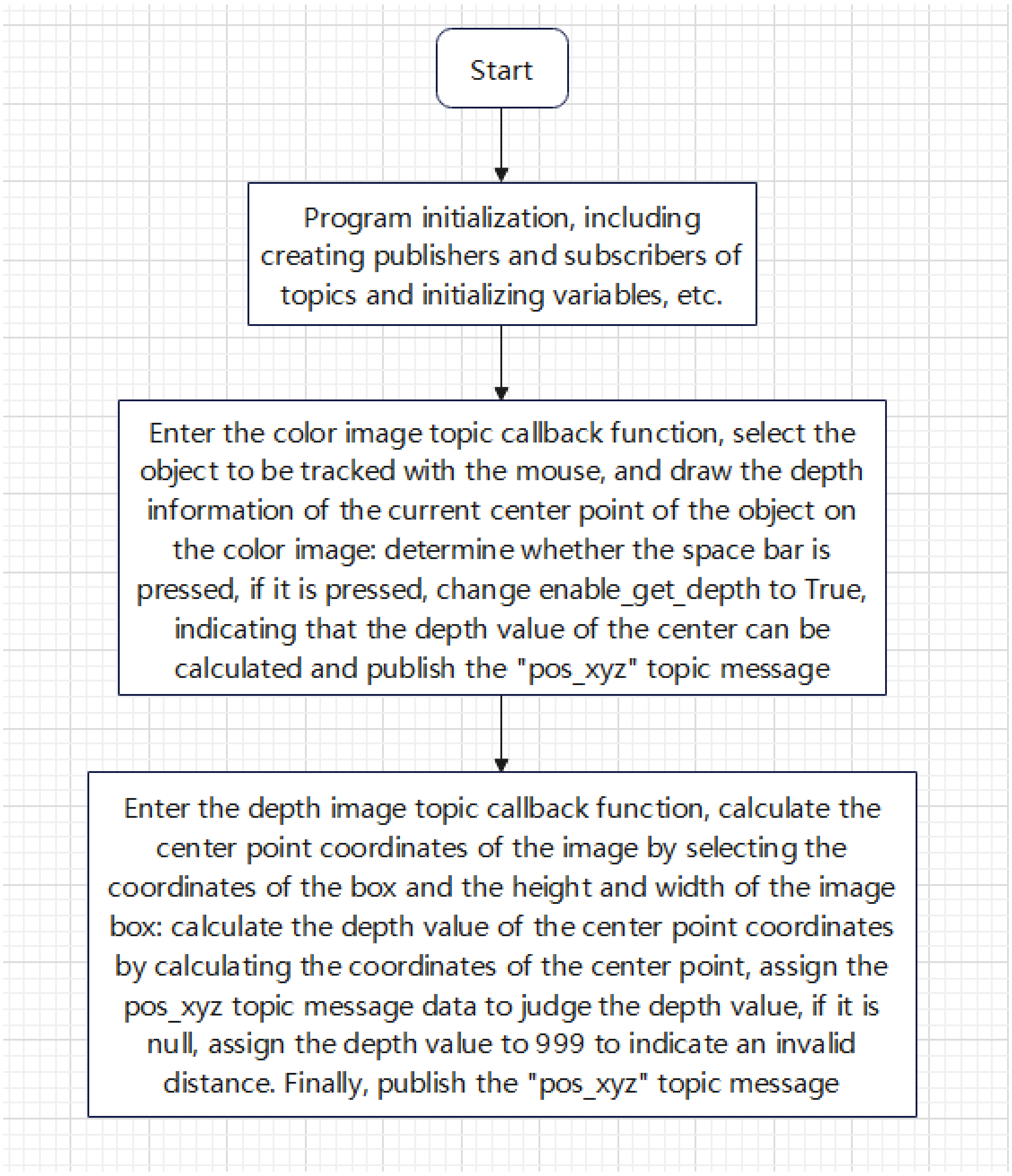
2.2, Operation

After the program is started, hold an object, click the mouse, and select the object in the image frame. The selection requirement is that the width of the selection frame is the same as the width of the object in the image. After releasing the mouse, the object will be framed in the image, and press the space bar to start tracking. Slowly move the object, and the robot will adjust its posture so that the center of the object always coincides with the center of the image. After the tracking action stops, wait for 2-3 seconds. If the depth distance is valid (not 0) and the object is within the set gripping range (greater than 18 cm and less than 30 cm), the buzzer will sound once, and then the robot will adjust its posture to grip the object. After gripping, place it at the set position, and then return to the initial posture of the robot.

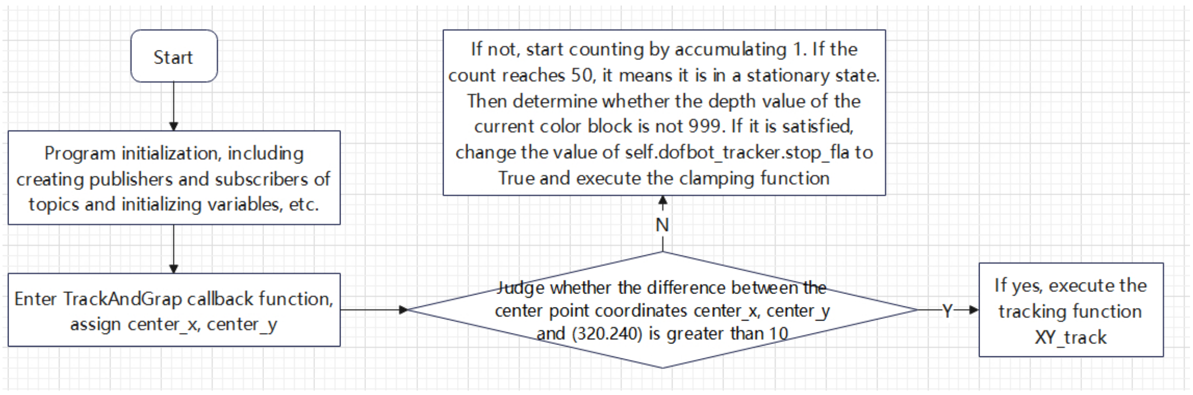


3. Program flow chart

KCF_Tracker.cpp


Start
Program initialization, including creating publishers and subscribers of topics and initializing variables, etc.
Enter the color image topic callback function, select the object to be tracked with the mouse, and draw the depth information of the current center point of the object on the color image: determine whether the space bar is pressed, if it is pressed, change enable_get_depth to True, indicating that the depth value of the center can be calculated and publish the "pos_xyz" topic message
Enter the depth image topic callback function, calculate the center point coordinates of the image by selecting the coordinates of the box and the height and width of the image box: calculate the depth value of the center point coordinates by calculating the coordinates of the center point, assign the pos_xyz topic message data to judge the depth value, if it is null, assign the depth value to 999 to indicate an invalid distance. Finally, publish the "pos_xyz" topic message

KCF_TrackAndGrap.py


Start
Program initialization, including creating publishers and subscribers of topics and initializing variables, etc.
Enter TrackAndGrap callback function, assign center_x, center_y
Judge whether the difference between the center point coordinates center_x, center_y and (320.240) is greater than 10
N
If not, start counting by accumulating 1. If the count reaches 50, it means it is in a stationary state. Then determine whether the depth value of the current color block is not 999. If it is satisfied, change the value of self.dofbot_tracker.stop_fla to True and execute the clamping function
Y
If yes, execute the tracking function XY_track

4. Core code analysis

4.1. KCF_Tracker.cpp

Code path: `/home/jetson/dofbot_pro_ws/src/dofbot_pro_KCF/src/KCF_Tracker.cpp`

Constructor, mainly to create publishers and subscribers,

```
ImageConverter::ImageConverter(ros::NodeHandle &n) {
    KCFTracker tracker(HOG, FIXEDWINDOW, MULTISCALE, LAB);
    //Subscribers to color image messages
    image_sub_ = n.subscribe("/camera/color/image_raw", 1,
&ImageConverter::imageCb, this);
    //Subscriber to the depth image message
    depth_sub_ = n.subscribe("/camera/depth/image_raw", 1,
&ImageConverter::depthCb, this);
    //Subscribe to the subscriber of the clip result message
    Move_sub_ = n.subscribe("/grab", 1, &ImageConverter::MoveCb, this);
    //Publisher of KCF images
    image_pub_ = n.advertise<sensor_msgs::Image>("/KCF_image", 1);
    //Publisher of object center coordinates and depth messages
    pub_pos = n.advertise<dofbot_pro_info::Position>("/pos_xyz", 1);
    server.setCallback(f);
    namedWindow(RGB_WINDOW);
}
```

Color image topic callback function,

```
void ImageConverter::imageCb(const sensor_msgs::ImageConstPtr &msg) {
    cv_bridge::CvImagePtr cv_ptr;
    try {
        cv_ptr = cv_bridge::toCvCopy(msg, sensor_msgs::image_encodings::BGR8);
    }
    catch (cv_bridge::Exception &e) {
        ROS_ERROR("cv_bridge exception: %s", e.what());
        return;
    }
    cv_ptr->image.copyTo(rgbimage);
    setMouseCallback(RGB_WINDOW, onMouse, 0);
    if (bRenewROI) {
         if (selectRect.width <= 0 || selectRect.height <= 0)
         {
             bRenewROI = false;
             return;
         }
        tracker.init(selectRect, rgbimage);
        bBeginKCF = true;
        bRenewROI = false;
        enable_get_depth = false;
    }
    if (bBeginKCF) {
        result = tracker.update(rgbimage);
        rectangle(rgbimage, result, Scalar(0, 255, 255), 1, 8);

        circle(rgbimage, Point(result.x + result.width / 2, result.y +
result.height / 2), 3, Scalar(0, 0, 255),-1);
    } else rectangle(rgbimage, selectRect, Scalar(255, 0, 0), 2, 8, 0);
    sensor_msgs::ImagePtr kcf_imagemsg = cv_bridge::CvImage(std_msgs::Header(), 
"bgr8", rgbimage).toImageMsg();
    image_pub_.publish(kcf_imagemsg);
    std::string text = std::to_string(get_depth);
    std::string units= "m";
    text  = text + units;
    cv::Point org(30, 30);
    int fontFace = cv::FONT_HERSHEY_SIMPLEX;
    double fontScale = 1.0;
    int thickness = 2;
    cv::Scalar color(0, 0, 255);  // Text color: red
    // Draw the depth value of the object center in the color image
    putText(rgbimage, text, org, fontFace, fontScale, color, thickness);
    imshow(RGB_WINDOW, rgbimage);
    int action = waitKey(1) & 0xFF;
    if (action == 'q' || action == ACTION_ESC) this->Cancel();
    else if (action == 'r'||action == 'R')  this->Reset();
    else if (action == ACTION_SPACE) enable_get_depth = true;
}
```

Depth image topic callback function,

```
void ImageConverter::depthCb(const sensor_msgs::ImageConstPtr &msg) {
    cv_bridge::CvImagePtr cv_ptr;
    try {
        cv_ptr = cv_bridge::toCvCopy(msg, 
sensor_msgs::image_encodings::TYPE_32FC1);
        cv_ptr->image.copyTo(depthimage);
    }
    catch (cv_bridge::Exception &e) {
        ROS_ERROR("Could not convert from '%s' to 'TYPE_32FC1'.", msg-
>encoding.c_str());
    }
    if (enable_get_depth) {
        // Calculate the center coordinates based on the coordinates of the 
selected object image and the image width and image height
        int center_x = (int)(result.x + result.width / 2);
        int center_y = (int)(result.y + result.height / 2);
        // Calculate the depth value of the object's center coordinates
        dist_val[0] = depthimage.at<float>(center_y - 5, center_x - 5)/1000;
        dist_val[1] = depthimage.at<float>(center_y - 5, center_x + 5)/1000;
        dist_val[2] = depthimage.at<float>(center_y + 5, center_x + 5)/1000;
        dist_val[3] = depthimage.at<float>(center_y + 5, center_x - 5)/1000;
        dist_val[4] = depthimage.at<float>(center_y, center_x)/1000;
        float distance = 0;
        int num_depth_points = 5;
        for (int i = 0; i < 5; i++) {
            if (dist_val[i] !=0 ) distance += dist_val[i];
            else num_depth_points--;
        }
        distance /= num_depth_points;
        //Create a pos message object
        dofbot_pro_info::Position pos;
        //Assign values ••to the data in the pos message object
        pos.x = center_x;
        pos.y = center_y;
        //Judge whether the distance value of the center point is valid. If it
is an invalid distance, assign 999 to indicate an invalid distance
        if (std::isnan(distance))
        {
            ROS_INFO("distance error!");
            distance = 999;
        }
        pos.z = distance;
        get_depth = distance;
        ROS_INFO("center_x =  %d, center_y =  %d,distance = %.3f",
center_x,center_y,distance);
        // Publish /pos_xyz topic message
        pub_pos.publish(pos);
    }
    waitKey(1);
}
```

4.2、KCF_TrackAndGrap.py

Code path: `/home/jetson/dofbot_pro_ws/src/dofbot_pro_KCF/scripts/KCF_TrackAndGrap.py`

Import necessary libraries,

```
import rospy
import numpy as np
import message_filters
from Dofbot_Track import *
from dofbot_pro_info.msg import *
```

Initialize program parameters, create publishers, subscribers, etc.

```
def __init__(self):
    rospy.init_node('KCF_tracking')
    #Create a tracking gripper object
    self.dofbot_tracker = DofbotTrack()
    #Create a subscriber to subscribe to the center position information of the
object
    self.pos_sub = message_filters.Subscriber('/pos_xyz',Position)
    self.TimeSynchronizer =
message_filters.ApproximateTimeSynchronizer([self.pos_sub],10,0.5,allow_headerle
ss=True)
    self.TimeSynchronizer.registerCallback(self.TrackAndGrap)
    #The depth value of the current object center coordinate
    self.cur_distance = 0.0
    self.cnt = 0
    print("Init done!")
```

Mainly look at the callback function,

```
def TrackAndGrap(self,position):

    if position.z != 0 :
        center_x, center_y = position.x,position.y
        self.cur_distance = position.z
        #If the coordinates of the center point of the object 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 object within the acceptable
range
        if abs(center_x-320) >10 or abs(center_y-240)>10 :
            #Execute the tracking program, input the center value of the current
object
            self.dofbot_tracker.XY_track(center_x,center_y)
        else:
            #If the coordinates of the center point of the machine code and the
coordinates of the center point of the image (320, 240) are less than 10, it can
be regarded that the center value of the object at this time is in the middle of
the image, and self.cnt is accumulated
            self.cnt = self.cnt + 1
            #When the cumulative number reaches 10, it means that the center
value of the machine code can be stationary in the middle of the image.
            if self.cnt == 10 :
                # Clear the count of self.cnt
                self.cnt = 0
                #Judge whether the center value of the object is not 999, 999
indicates an invalid distance
                if self.cur_distance!= 999:
                    self.dofbot_tracker.stop_flag = True

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