AprilTag Abnormal Height Sorting

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

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

1. Function Description

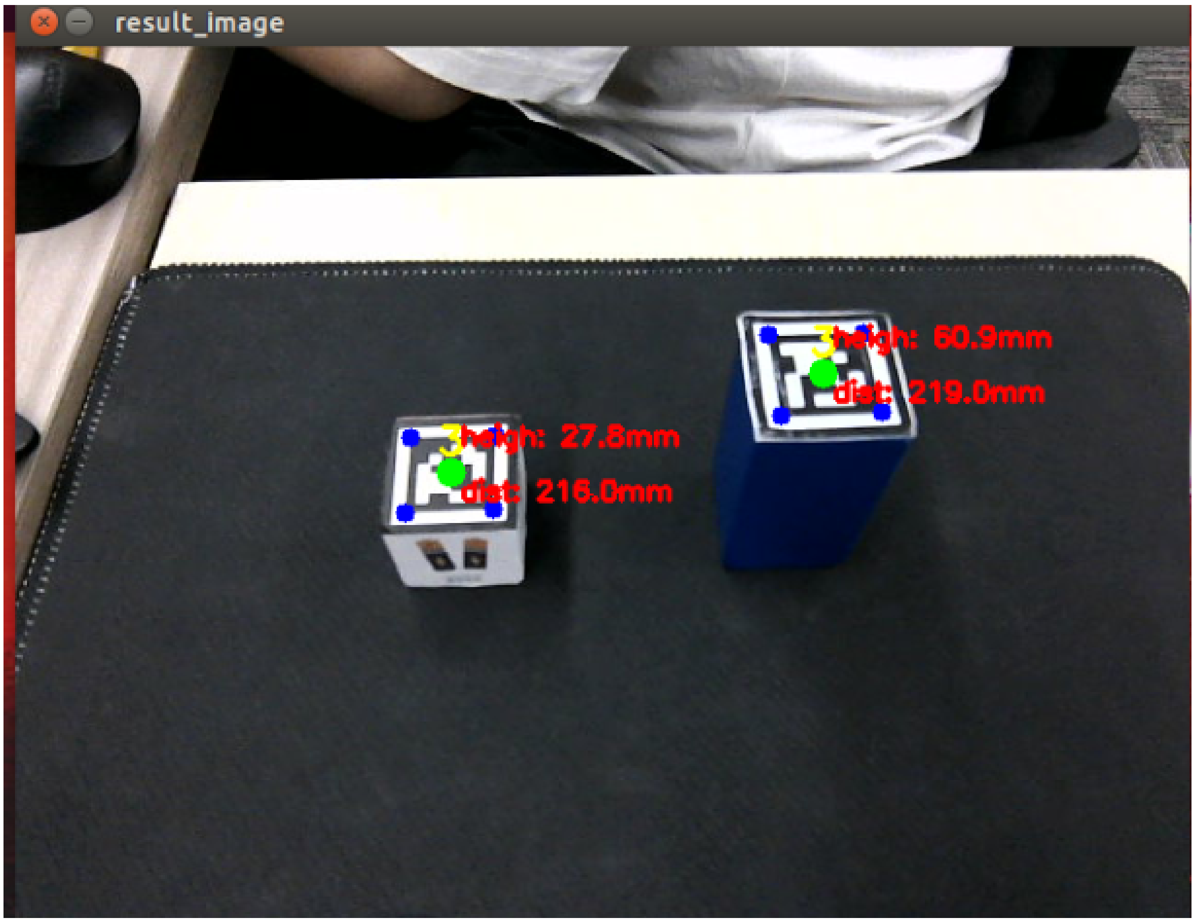
After the program starts, when the camera recognizes the AprilTag, it will grasp AprilTags with height higher than the set height and place them at the designated position.

2. Startup and Operation

2.1. Startup Commands

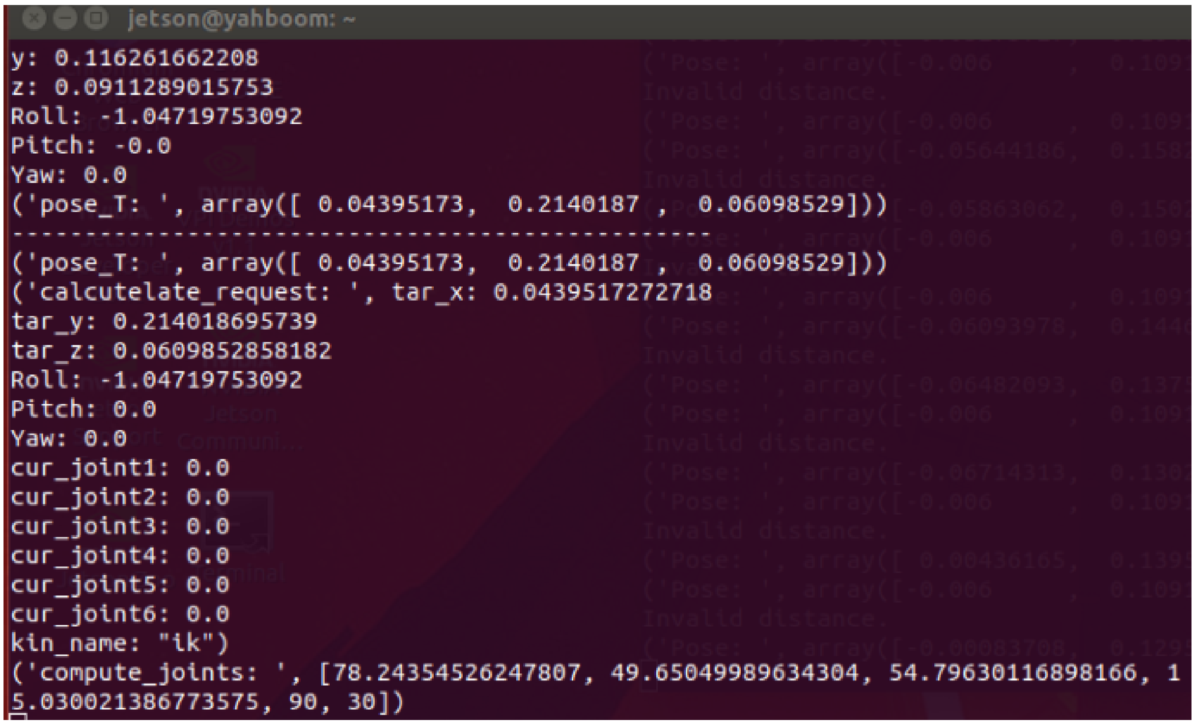
Enter in the terminal:

```
#Start camera
ros2 launch orbbec_camera dabai_dcw2.launch.py
#Start inverse kinematics program
ros2 run dofbot_pro_info kinemarics_dofbot
#Start underlying control
ros2 run dofbot_pro_driver arm_driver
#Start AprilTag detection and recognition program
ros2 run dofbot_pro_driver apriltag_list
#Start robotic arm AprilTag grasping program
ros2 run dofbot_pro_driver apriltag_remove_higher
```



2.2. Operation

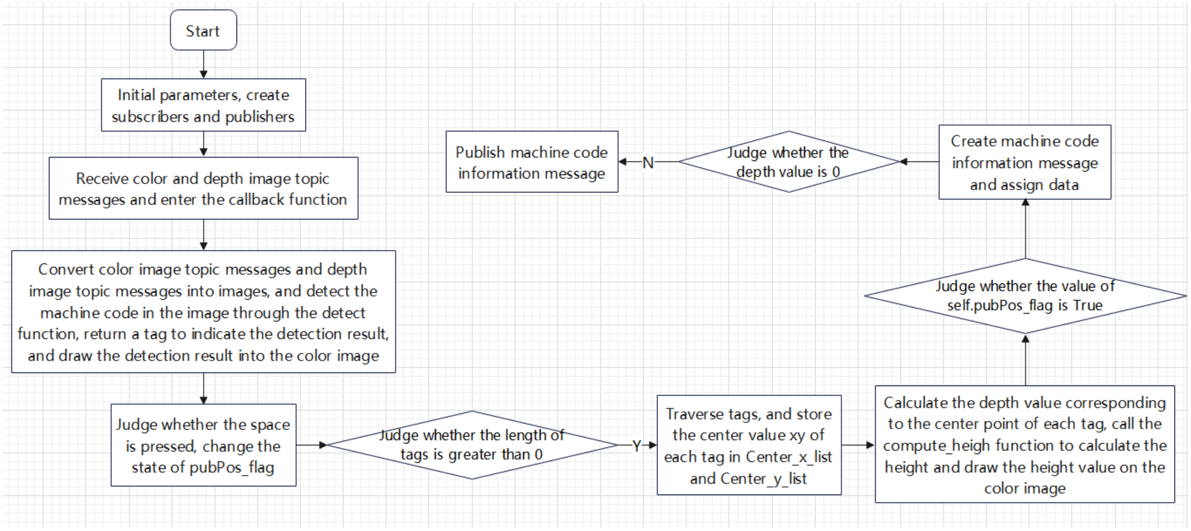
Click on the image frame with the mouse, then press the spacebar on the keyboard. The robotic arm will grasp AprilTag blocks higher than the set height and place them at the designated position. After placement is completed, it will return to the recognition pose and continue to recognize AprilTags and detect if there are AprilTag blocks with abnormal height. The second time an abnormal height block is recognized, there is no need to press the spacebar for grasping. The terminal will print grasping information.



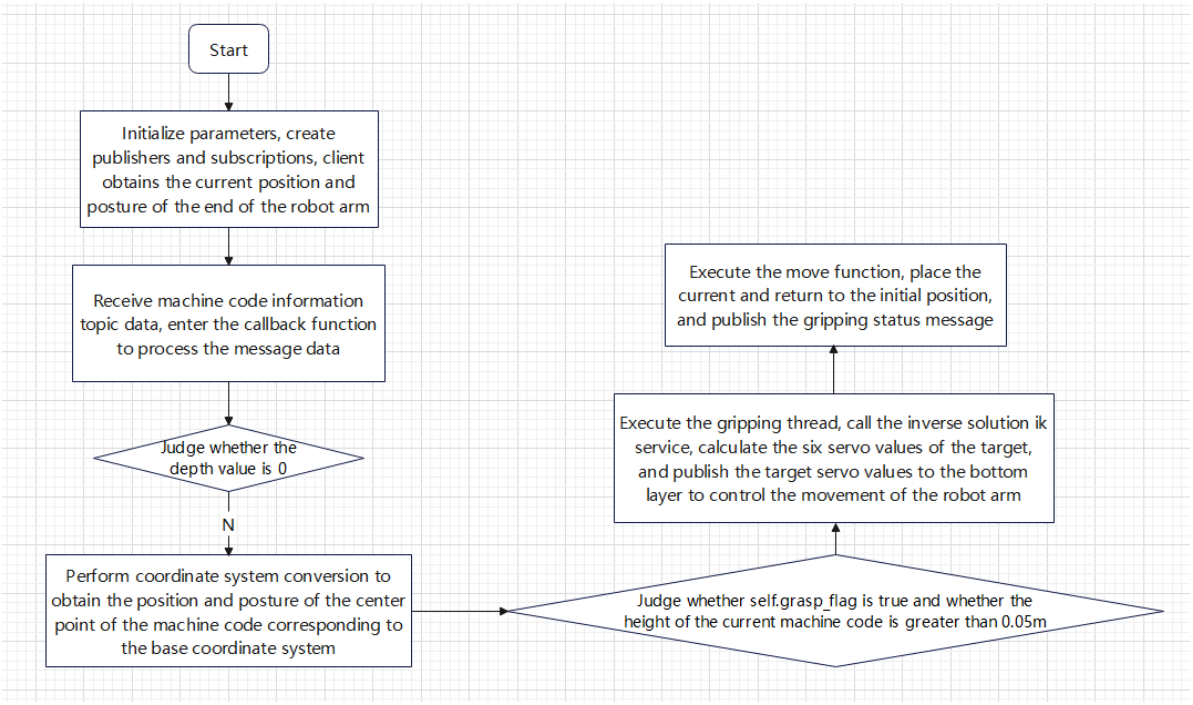
```
y: 0.116261662208
z: 0.0911289015753
Roll: -1.04719753092
Pitch: -0.0
Yaw: 0.0
('pose_T: ', array([ 0.04395173,  0.2140187 ,  0.06098529]))
------------------------------------------------
('pose_T: ', array([ 0.04395173,  0.2140187 ,  0.06098529]))
('calcutelate_request: ', tar_x: 0.0439517272718
tar_y: 0.214018695739
tar_z: 0.0609852858182
Roll: -1.04719753092
Pitch: 0.0
Yaw: 0.0
cur_joint1: 0.0
cur_joint2: 0.0
cur_joint3: 0.0
cur_joint4: 0.0
cur_joint5: 0.0
cur_joint6: 0.0
kin_name: "ik")
('compute_joints: ', [78.24354526247807, 49.65049989634304, 54.79630116898166, 1
5.030021386773575, 90, 30])
```

3. Program Flowchart

3.1. apriltag_list.py


Start
Initial parameters, create subscribers and publishers
Receive color and depth image topic messages and enter the callback function
Convert color image topic messages and depth image topic messages into images, and detect the machine code in the image through the detect function, return a tag to indicate the detection result, and draw the detection result into the color image
Judge whether the space is pressed, change the state of pubPos_flag
Judge whether the length of tags is greater than 0
Y
Traverse tags, and store the center value xy of each tag in Center_x_list and Center_y_list
Calculate the depth value corresponding to the center point of each tag, call the compute_heigh function to calculate the height and draw the height value on the color image
Judge whether the value of self.pubPos_flag is True
Create machine code information message and assign data
Judge whether the depth value is 0
N
Publish machine code information message

3.2. apriltag_remove_higher.py


Start
Initialize parameters, create publishers and subscriptions, client obtains the current position and posture of the end of the robot arm
Receive machine code information topic data, enter the callback function to process the message data
Judge whether the depth value is 0
N
Perform coordinate system conversion to obtain the position and posture of the center point of the machine code corresponding to the base coordinate system
Judge whether self.grasp_flag is true and whether the height of the current machine code is greater than 0.05m
Execute the gripping thread, call the inverse solution ik service, calculate the six servo values of the target, and publish the target servo values to the bottom layer to control the movement of the robot arm
Execute the move function, place the current and return to the initial position, and publish the gripping status message

4. Core Code Analysis

4.1. apriltag_list.py

Code path:

```
/home/jetson/dofbot_pro_ws/src/dofbot_pro_driver/dofbot_pro_driver/apriltag_list
.py
```

Import necessary library files

```
import rclpy
from rclpy.node import Node
import numpy as np
from sensor_msgs.msg import Image
from message_filters import ApproximateTimeSynchronizer, Subscriber
#Import AprilTag drawing library
from dofbot_pro_driver.vutils import draw_tags
#Import AprilTag library
from dt_apriltags import Detector
from cv_bridge import CvBridge
import cv2 as cv
#Import custom service data type
from dofbot_pro_interface.srv import Kinemarics
#Import custom message data type
from dofbot_pro_interface.msg import *
from std_msgs.msg import Float32,Bool
encoding = ['16UC1', '32FC1']
import time
#Import transforms3d library for handling transformations in 3D space, performing
conversions between quaternions, rotation matrices and Euler angles, supporting
3D geometric operations and coordinate transformations
import transforms3d as tfs
#Import transformations for handling and calculating transformations in 3D space,
including conversions between quaternions and Euler angles
import tf_transformations as tf
import math
```

Program parameter initialization, create publishers, subscribers and clients

```
def __init__(self):
    super().__init__('apriltag_detect')
    #Publish initial pose of robotic arm, also the recognition pose
    self.init_joints = [90.0, 120, 0, 0.0, 90, 90]
    #Create two subscribers, subscribe to color image topic and depth image
topic
    self.depth_image_sub = Subscriber(self, Image, '/camera/depth/image_raw')
    self.rgb_image_sub = Subscriber(self, Image, '/camera/color/image_raw')
    #Create publisher for AprilTag information
    self.pos_info_pub = self.create_publisher(AprilTagInfo, "PosInfo",
qos_profile=10)

    #Synchronize color and depth image subscription messages by time
    self.ts = ApproximateTimeSynchronizer([self.rgb_image_sub,
self.depth_image_sub],queue_size=10,slop=0.5)
    #Create subscriber for grasping results
    self.subscription =
self.create_subscription(Bool,'grasp_done',self.GraspStatusCallback,qos_profile=
1)
    #Connect the callback function TagDetect for processing synchronized messages
with the subscribed messages, so this function is automatically called when new
messages are received
    self.ts.registerCallback(self.TagDetect)
    #Create bridge for converting color and depth image topic message data to
image data
    self.rgb_bridge = CvBridge()
    self.depth_bridge = CvBridge()
    #Flag for publishing AprilTag information, when True, publish /TagInfo topic
data
    self.pubPos_flag = False
    #Create AprilTag object, set some parameters as follows,
    '''
    searchpath: Specify the path to find tag models.
    families: Set the tag family to use, e.g., 'tag36h11'.
    nthreads: Number of parallel processing threads to improve detection speed.
    quad_decimate: Reduce input image resolution to reduce computation.
    quad_sigma: Standard deviation of Gaussian blur, affecting image
preprocessing.
    refine_edges: Whether to refine edges to improve detection accuracy.
    decode_sharpening: Sharpening parameter during decoding to enhance tag
contrast.
    debug: Debug mode switch, convenient for viewing information during detection
process
    '''
    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)
    #List to store AprilTag center x values
    self.Center_x_list = []
    #List to store AprilTag center y values
    self.Center_y_list = []
    self.heigh = 0.0
    #Position and pose of robotic arm end
    self.CurEndPos = [-0.006,0.116261662208,0.0911289015753,-1.04719,-0.0,0.0]
    #Depth camera intrinsic parameters
    self.camera_info_K = [477.57421875, 0.0, 319.3820495605469, 0.0,
477.55718994140625, 238.64108276367188, 0.0, 0.0, 1.0]
    #Rotation transformation matrix between robotic arm end and camera,
describing relative position and pose between them
    self.EndToCamMat =
np.array([[1.00000000e+00,0.00000000e+00,0.00000000e+00,0.00000000e+00],
    [0.00000000e+00,7.96326711e-04,9.99999683e-01,-9.90000000e-02],
    [0.00000000e+00,-9.99999683e-01,7.96326711e-04,4.90000000e-02],
    [0.00000000e+00,0.00000000e+00,0.00000000e+00,1.00000000e+00]])
```

AprilTag information callback function tag_info_callback,

```
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])
    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) # Seems to be sorted in
ascending order by default, no need for manual sorting
    draw_tags(result_image, tags, corners_color=(0, 0, 255), center_color=(0,
255, 0))
    key = cv2.waitKey(10)
    self.Center_x_list = list(range(len(tags)))
    self.Center_y_list = list(range(len(tags)))
    if key == 32:
        self.pubPos_flag = True
    if len(tags) > 0 :
        for i in range(len(tags)):
            center_x, center_y = tags[i].center
            #Store AprilTag center xy values in Center_x_list and Center_y_list
            self.Center_x_list[i] = center_x
            self.Center_y_list[i] = center_y
            cx = center_x
            cy = center_y
            #Calculate depth value of center coordinates
            cz = depth_image_info[int(cy),int(cx)]/1000
            #Call compute function to calculate AprilTag height, parameters are 
AprilTag center coordinates and center point depth value, returns a position 
list, pose[2] represents z value, which is height value
            pose = self.compute_heigh(cx,cy,cz)
            #Perform scaling operation on height value, convert unit to 
millimeters
            heigh = round(pose[2],4)*1000
            heigh = 'heigh: ' + str(heigh) + 'mm'
            #Calculate distance value of AprilTag from base coordinate system, 
perform scaling operation on this value, convert unit to millimeters
            dist_detect = math.sqrt(pose[1] ** 2 + pose[0]** 2)
            dist_detect = round(dist_detect,3)*1000
            dist = 'dist: ' + str(dist_detect) + 'mm'
            #Draw height and distance values on color image using opencv
            cv.putText(result_image, heigh, (int(cx)+5, int(cy)-15), 
cv.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
            cv.putText(result_image, dist, (int(cx)+5, int(cy)+15), 
cv.FONT_HERSHEY_SIMPLEX, 0.5, (255, 0, 0), 2)
            print("Pose: ",pose)
        for i in range(len(tags)):
            if self.pubPos_flag == True:
                tag = AprilTagInfo()
                #Assign values to message data, id value is AprilTag id, x and y 
are AprilTag center values, z is center point depth value, here scaled down by 
1000 times, unit is meters
                tag.id = tags[i].tag_id
                tag.x = self.Center_x_list[i]
                tag.y = self.Center_y_list[i]
                tag.z = depth_image_info[int(tag.y),int(tag.x)]/1000
                #If AprilTag distance is not equal to 0, it means it's valid 
data, then publish AprilTag information message
                if tag.z!=0:
                    self.tag_info_pub.publish(tag)
                #Change self.pubPos_flag state to prevent multiple message 
publishing, wait for grasping to complete before changing state
                    self.pubPos_flag = False
                else:
                    print("Invalid distance.")
    #Convert color image color space, convert RGB to BGR
    result_image = cv2.cvtColor(result_image, cv2.COLOR_RGB2BGR)
    #Display image
    cv2.imshow("result_image", result_image)
    key = cv2.waitKey(1)
```

Calculate height value function compute_heigh

```
def compute_heigh(self,x,y,z):
    #First coordinate system conversion, from pixel coordinate system to camera
coordinate system
    camera_location = self.pixel_to_camera_depth((x,y),z)
    #print("camera_location: ",camera_location)
    #Second coordinate system conversion, from camera coordinate system to
robotic arm end coordinate system
    PoseEndMat = np.matmul(self.EndToCamMat,
self.xyz_euler_to_mat(camera_location, (0, 0, 0)))
    #PoseEndMat = np.matmul(self.xyz_euler_to_mat(camera_location, (0, 0,
0)),self.EndToCamMat)
    #Get current robotic arm end position and pose
    EndPointMat = self.get_end_point_mat()
    #Third coordinate system conversion, from robotic arm end coordinate system
to base coordinate system, the resulting worldPose (rotation transformation
matrix) is the position and pose of the AprilTag center relative to the robotic
arm base coordinate system
    WorldPose = np.matmul(EndPointMat, PoseEndMat)
    #WorldPose = np.matmul(PoseEndMat,EndPointMat)
    #Convert rotation transformation matrix to xyz and Euler angles
    pose_T, pose_R = self.mat_to_xyz_euler(WorldPose)
    #Return pose_T representing position
    return pose_T
```

4.2. apriltag_remove_higher.py

Code path:

```
/home/jetson/dofbot_pro_ws/src/dofbot_pro_driver/dofbot_pro_driver/apriltag_remo
ve_higher.py
```

Import necessary library files

```
import math
import rclpy
from rclpy.node import Node
import numpy as np
from std_msgs.msg import Float32,Bool
import time
from dofbot_pro_interface.msg import *       # Need to confirm ROS2 message
package name consistency
from dofbot_pro_interface.srv import Kinemarics
#Import transforms3d library for handling transformations in 3D space, performing
conversions between quaternions, rotation matrices and Euler angles, supporting
3D geometric operations and coordinate transformations
import transforms3d as tfs
#Import transformations for handling and calculating transformations in 3D space,
including conversions between quaternions and Euler angles
import tf_transformations as tf
import threading
```

Program parameter initialization, create publishers, subscribers and clients

```
def __init__(self):
     super().__init__('color_grap')

    #Create subscriber for TagInfo topic, subscribe to AprilTag information 
messages
    self.sub = 
self.create_subscription(AprilTagInfo,'PosInfo',self.pos_callback,1)
    #Create publisher for servo target angle topic, publish messages to control 
robotic arm servos
    self.pub_point = self.create_publisher(ArmJoint, 'TargetAngle',1)
    #Create publisher for grasping result topic, publish grasping result 
messages
    self.pubGraspStatus = self.create_publisher(Bool, 'grasp_done',1)
    #Create client for requesting inverse kinematics service, used to calculate 
current robotic arm end position and pose and solve target servo values
    self.client = self.create_client(Kinemarics, 'dofbot_kinemarics')
    #Initial grasping flag, True means can grasp, False means cannot grasp
    self.grasp_flag = True
    self.init_joints = [90.0, 120, 0.0, 0.0, 90, 90]
    self.down_joint = [130.0, 55.0, 34.0, 16.0, 90.0,125]
    self.gripper_joint = 90
    #Initialize current position pose, corresponding to x, y, z, roll, pitch and 
yaw
    self.CurEndPos = [0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
    #Depth camera intrinsic parameters
    self.camera_info_K = [477.57421875, 0.0, 319.3820495605469, 0.0, 
477.55718994140625, 238.64108276367188, 0.0, 0.0, 1.0]
    #Rotation transformation matrix between robotic arm end and camera, 
describing relative position and pose between them
    self.EndToCamMat = 
np.array([[1.00000000e+00,0.00000000e+00,0.00000000e+00,0.00000000e+00],
                                 [0.00000000e+00,7.96326711e-04,9.99999683e-
01,-9.90000000e-02],
                                 [0.00000000e+00,-9.99999683e-01,7.96326711e-
04,4.90000000e-02],
                                 
[0.00000000e+00,0.00000000e+00,0.00000000e+00,1.00000000e+00]])
    #Get current robotic arm end position and pose, will change self.CurEndPos 
value
    self.get_current_end_pos()
    #Define current id value, later use this value to place AprilTag at 
corresponding position
    self.cur_tagId = 0
    #Print current robotic arm end position pose
    print("Current_End_Pose: ",self.CurEndPos)
    print("Init Done")
```

AprilTag information callback function tag_info_callback,

```
def tag_info_callback(self,msg):
    #print("msg: ",msg)
    pos_x = msg.x
    pos_y = msg.y
    pos_z = msg.z
    self.cur_tagId = msg.id
    #If the received center point depth information is not >0, it means it's 
valid data
```

```
    if pos_z!=0.0:
        print("xyz id : ",pos_x,pos_y,pos_z,self.cur_tagId)
        #Get current robotic arm end position and pose
        self.get_current_end_pos()
        #First coordinate system conversion, from pixel coordinate system to
camera coordinate system
        camera_location = self.pixel_to_camera_depth((pos_x,pos_y),pos_z)
        #print("camera_location: ",camera_location)
        #Second coordinate system conversion, from camera coordinate system to
robotic arm end coordinate system
        PoseEndMat = np.matmul(self.EndToCamMat,
self.xyz_euler_to_mat(camera_location, (0, 0, 0)))
        EndPointMat = self.get_end_point_mat()
        #Third coordinate system conversion, from robotic arm end coordinate
system to base coordinate system, the resulting worldPose (rotation
transformation matrix) is the position and pose of the AprilTag center relative
to the robotic arm base coordinate system
        WorldPose = np.matmul(EndPointMat, PoseEndMat)
        #Convert rotation transformation matrix to xyz and Euler angles
        pose_T, pose_R = self.mat_to_xyz_euler(WorldPose)
        print("pose_T: ",pose_T)
        #Check grasping flag, if True and height greater than 0.05, i.e., higher
than 0.05m, execute grasping
        if self.grasp_flag == True and (pose_T[2])>0.05:
            self.grasp_flag = False
            #Start a thread, thread executes grasp program, parameter is the
calculated pose_T, which is xyz value, representing the target position of
robotic arm end
            grasp = threading.Thread(target=self.grasp, args=(pose_T,))
            grasp.start()
            grasp.join()
```

Robotic arm grasping function grasp

```
def grasp(self,pose_T):
    print("------------------------------------------------")
    print("pose_T: ",pose_T)
    #Call inverse kinematics service, calling ik service content, assign required
request parameters
    request = kinemaricsRequest()
    #Target x value of robotic arm end, unit is m, 0.01 is x-axis direction
(left-right) offset parameter, due to minor differences in servos, cannot
guarantee calculated position matches actual position, make minor adjustments
according to actual situation
    request.tar_x = pose_T[0]  - 0.01
    #Target y value of robotic arm end, unit is m, 0.015 is y-axis direction
(front-back) offset parameter, due to minor differences in servos, cannot
guarantee calculated position matches actual position, make minor adjustments
according to actual situation
    request.tar_y = pose_T[1]  + 0.015
    #Target z value of robotic arm end, unit is m, 0.02 is z-axis direction (up-
down) offset parameter, due to minor differences in servos, cannot guarantee
calculated position matches actual position, make minor adjustments according to
actual situation
    request.tar_z = pose_T[2] + request.tar_y* 0.02
    #Specify service content as ik
    request.kin_name = "ik"
    #Target Roll value of robotic arm end, unit is radians, this value is the
current robotic arm end roll value
    request.Roll = self.CurEndPos[3]
    print("calcutelate_request: ",request)
    try:
        response = self.client.call(request)
        #print("calcutelate_response: ",response)
        joints = [0.0, 0.0, 0.0, 0.0, 0.0,0.0]
        #Assign returned joint1-joint6 values from 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)
        #Execute pubTargetArm function, pass calculated joints values as
parameters
        self.pubTargetArm(joints)
        time.sleep(3.5)
        #Execute move function, grasp wood block and place at set position
according to AprilTag id value
        self.move()
    except Exception:
       rospy.loginfo("run error")
```

Publish robotic arm target angle function pubTargetArm

```
def pubArm(self, joints, id=1, angle=90, run_time=2000):
    armjoint = ArmJoint()
    armjoint.run_time = run_time
    if len(joints) != 0: armjoint.joints = joints
    else:
        armjoint.id = id
        armjoint.angle = angle
        self.pubPoint.publish(armjoint)
```

Grasp and place function move

```
def move(self):
    print("self.gripper_joint = ",self.gripper_joint)
    self.pubArm([],5, self.gripper_joint, 2000)
    time.sleep(2.5)
    self.pubArm([],6, 135, 2000)
    time.sleep(2.5)
    self.pubArm([],2, 135, 2000)
    time.sleep(2.5)
    #Change self.down_joint value according to id value, this value represents
the position where AprilTag wood block is placed
    if self.cur_tagId == 1:
        self.down_joint = [130.0, 55.0, 34.0, 16.0, 90.0,135]
    elif self.cur_tagId == 2:
        self.down_joint = [170.0, 55.0, 34.0, 16.0, 90.0,135]
    elif self.cur_tagId == 3:
        self.down_joint = [50.0, 55.0, 34.0, 16.0, 90.0,135]
    elif self.cur_tagId == 4:
        self.down_joint = [10.0, 55.0, 34.0, 16.0, 90.0,135]
    self.pubArm(self.down_joint)
    time.sleep(2.5)
    self.pubArm([],6, 90, 2000)
    time.sleep(2.5)
    self.pubArm([],2, 90, 2000)
    time.sleep(2.5)
    #After placement is complete, return to initial position
    self.pubArm(self.init_joints)
    time.sleep(5)
    #After waiting for robotic arm to return to initial position, publish
grasping completion message, change grasping flag value for next grasping when
conditions are met
    self.grasp_flag = True
    grasp_done = Bool()
    grasp_done.data = True
    self.pubGraspStatus.publish(grasp_done)
```