AprilTag ID 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

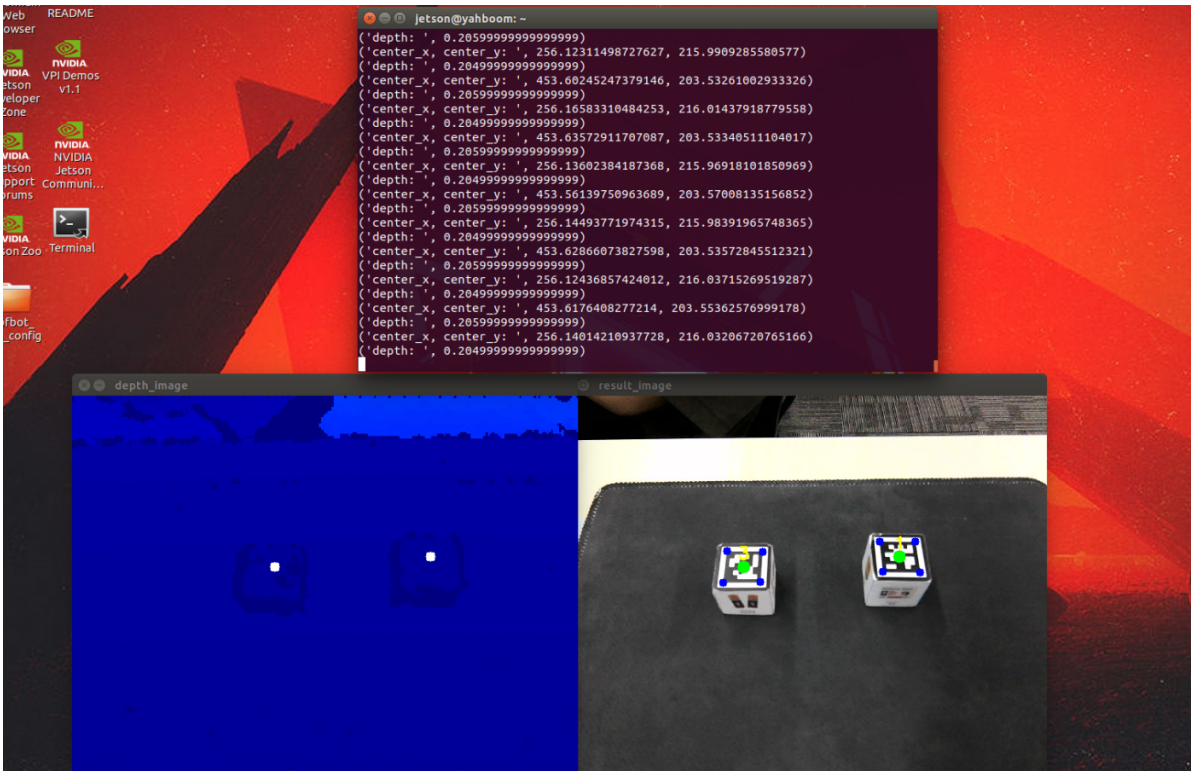
After the program starts, when the camera recognizes the AprilTag, it will frame the AprilTag in the image and print the corresponding ID value. Press the spacebar to start the sorting program. The robotic arm will grasp the recognized AprilTag block and place it at the specified position according to the recognized ID.

2. Startup and Operation

2.1. Startup Commands

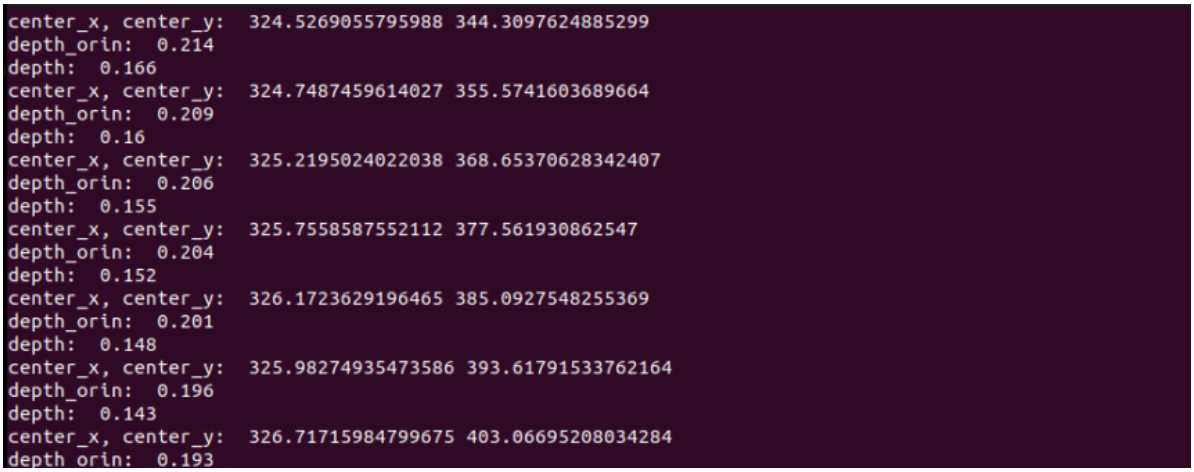
After opening the terminal, enter the following commands:

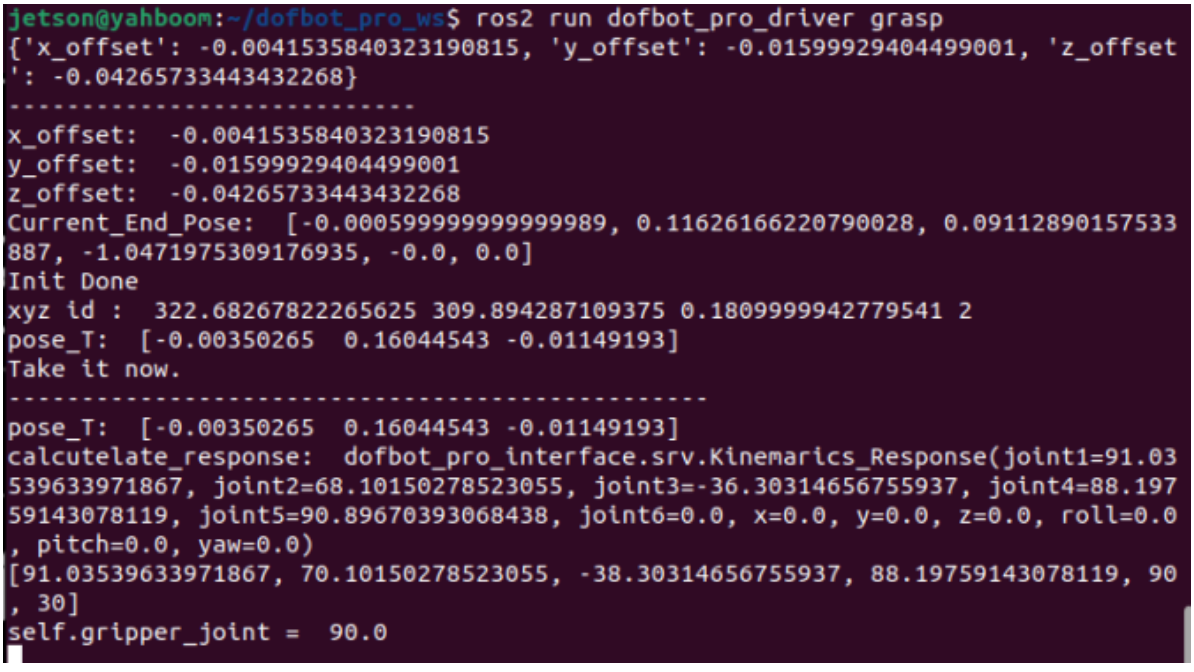
```
#Start camera
ros2 launch orbbec_camera dabai_dcw2.launch.py
#Start inverse kinematics program
ros2 run dofbot_pro_info kinemarics_dofbot
#Start underlying control
ros2 run dofbot_pro_driver arm_driver
#AprilTag recognition
ros2 run dofbot_pro_driver apriltag_detect
#Grasping
ros2 run dofbot_pro_driver grasp
```



2.2. Operation

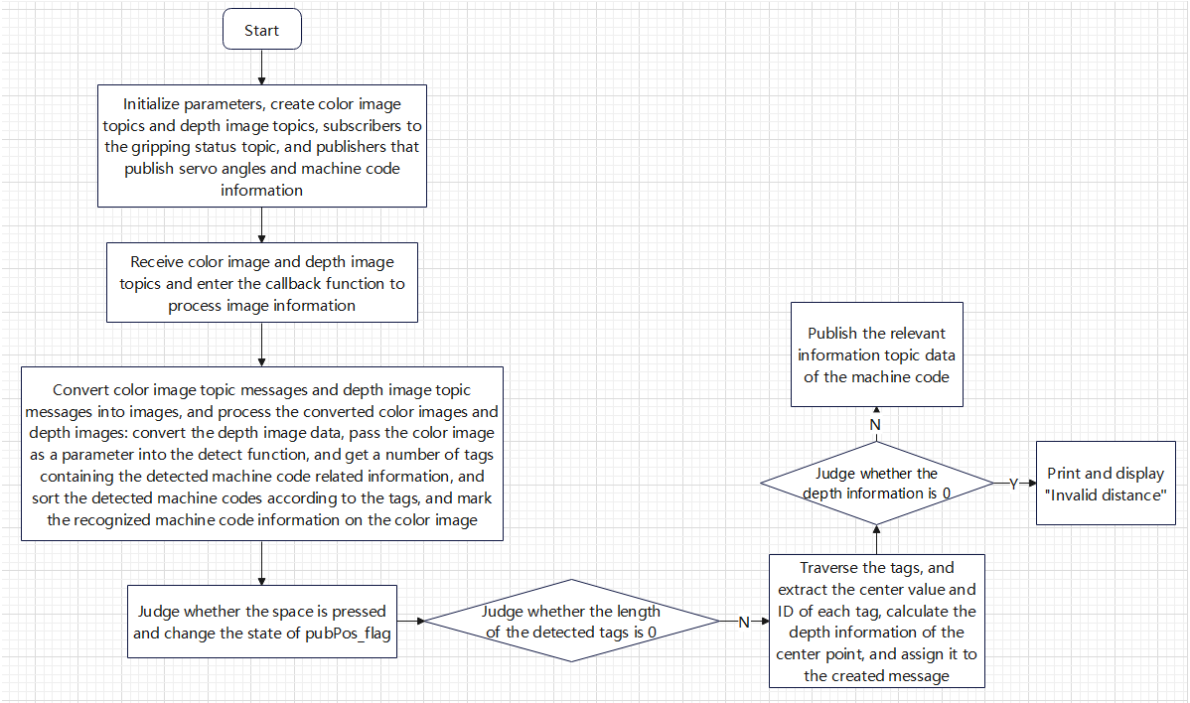
Use **3*3 square wooden blocks**. Click on the image frame with the mouse, then press the spacebar on the keyboard. The robotic arm will perform a series of calculations based on the center point coordinates of the recognized AprilTag block and the depth value of the center point's xy coordinates. Then, according to the calculation results, it will lower the gripper to grasp the recognized AprilTag block. After grasping is completed, it will place the AprilTag at the set position according to the ID value of the grasped AprilTag. After placement is completed, it will return to the recognition pose to recognize the next AprilTag. Two terminals will print the center value of the recognized AprilTag and the depth information of this center value, while another will print the real-time calculation process.



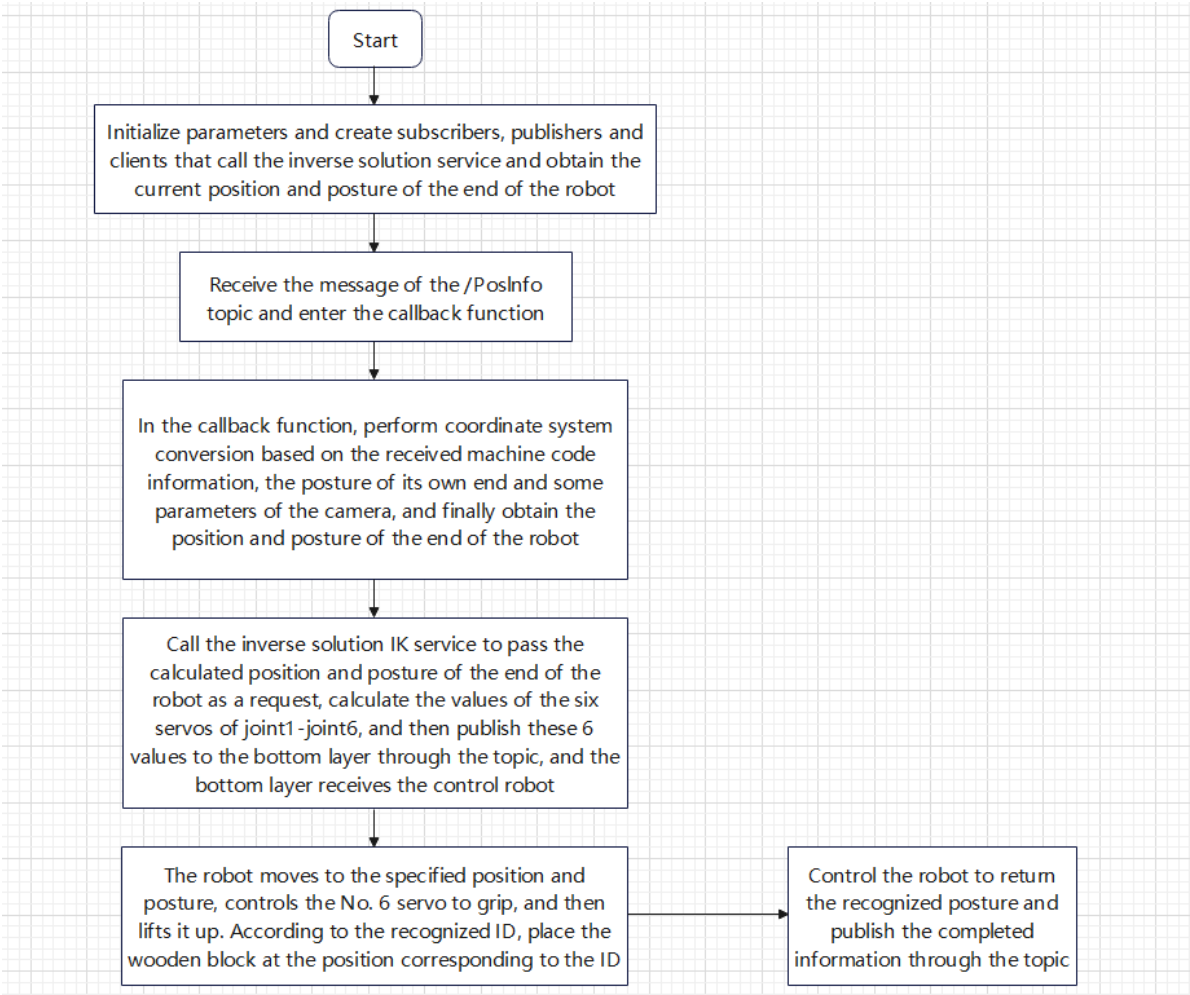


3. Program Flowchart

apriltag_detect.py



grasp.py



4. Core Code Analysis

4.1. apriltag_detect.py

Code path:

```
/home/jetson/dofbot_pro_ws/src/dofbot_pro_driver/dofbot_pro_driver/apriltag_dete
ct.py
```

Import necessary library files

```
import rclpy
from rclpy.node import Node
import cv2
import numpy as np
from sensor_msgs.msg import Image
from message_filters import ApproximateTimeSynchronizer, Subscriber
from std_msgs.msg import Float32, Bool
from cv_bridge import CvBridge
import cv2 as cv
from dt_apriltags import Detector
import threading
#Import AprilTag drawing library
from dofbot_pro_driver.vutils import draw_tags
#Import custom service data type
from dofbot_pro_interface.srv import Kinemarics
#Import custom message data type
from dofbot_pro_interface.msg import *
import pyzbar.pyzbar as pyzbar
from std_msgs.msg import Float32,Bool
import time
import queue
```

Program parameter initialization, create publishers and subscribers

```
def __init__(self):
    rospy.init_node('apriltag_detect')
    #self.init_joints = [90.0, 120, 0, 0.0, 90, 90]
    #Publish initial pose of robotic arm, also the recognition pose
    self.init_joints = [90.0, 120, 0.0, 0.0, 90, 30]
    #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)
    #Create publisher for robotic arm target angles
    self.pubPoint = self.create_publisher(ArmJoint, "TargetAngle",
qos_profile=1)
    #Synchronize color and depth image subscription messages by time
    self.TimeSynchronizer =
message_filters.ApproximateTimeSynchronizer([self.rgb_image_sub,self.depth_image
_sub],1,0.5)
    #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)
```

Main focus on the TagDetect callback function:

```
def TagDetect(self,color_frame,depth_frame):
    #rgb_image
    #Receive color image topic message and convert message data to image data
    rgb_image = self.rgb_bridge.imgmsg_to_cv2(color_frame,'rgb8')
    result_image = np.copy(rgb_image)
    #depth_image
    #Receive depth image topic message and convert message data to image data
    depth_image = self.depth_bridge.imgmsg_to_cv2(depth_frame, encoding[1])
    #Convert depth image to pseudo-color image
    depth_to_color_image = cv.applyColorMap(cv.convertScaleAbs(depth_image,
alpha=0.03), cv.COLORMAP_JET)
    frame = cv.resize(depth_image, (640, 480))
    depth_image_info = frame.astype(np.float32)
    #Call detect function with parameters,
    '''
    cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY): Convert RGB image to grayscale
image for tag detection.
    False: Indicates not to estimate tag pose.
    None: Indicates no camera parameters provided, may only perform simple
detection.
```

```
    0.025: May be the set tag size (unit is usually meters), used to help
detection algorithm determine tag size
    Returns a detection result, including position, ID and bounding box
information of each tag.
    '''
    tags = self.at_detector.detect(cv2.cvtColor(rgb_image, cv2.COLOR_RGB2GRAY),
False, None, 0.025)
    #Sort tags in tags, non-essential step
    tags = sorted(tags, key=lambda tag: tag.tag_id)
    #Call draw_tags function, which draws recognized AprilTag related information
on the color image, including corner points, center point and id value
    draw_tags(result_image, tags, corners_color=(0, 0, 255), center_color=(0,
255, 0))
    #Wait for keyboard input, 32 means spacebar pressed, after pressing change
self.pubPos_flag value, indicating AprilTag related information can be published
    key = cv2.waitKey(10)
    if key == 32:
        self.pubPos_flag = True
    #Check if tags length is greater than 0, indicating AprilTag detected
    if len(tags) > 0 :
        #Traverse AprilTags
        for i in range(len(tags)):
            if self.pubPos_flag == True:
                #Get center value of recognized AprilTag
                center_x, center_y = tags[i].center
                #Mark center point of AprilTag block on pseudo-color image
                cv.circle(result_image, (int(center_x),int(center_y)), 10,
(0,210,255), thickness=-1)
                #Create message data for AprilTag information
                tag = AprilTagInfo()
                #Assign values to message data, id is AprilTag id, x and y are
AprilTag center values, z is depth value of center point, here scaled down by
1000 times, unit is meters
                tag.id = tags[i].tag_id
                tag.x = center_x
                tag.y = center_y
                tag.z = depth_image_info[int(center_y),int(center_x)]/1000
                #Print recognized AprilTag information
                print("tag_id: ",tags[i].tag_id)
                print("center_x, center_y: ",center_x, center_y)
                print("depth:
",depth_image_info[int(center_y),int(center_x)]/1000)
                #Check if AprilTag distance is greater than 0, indicating 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 completion before changing state
                    self.pubPos_flag = False
                else:
                    print("Invalid distance.")
    #Convert color image color space, RGB to BGR
    result_image = cv2.cvtColor(result_image, cv2.COLOR_RGB2BGR)
    #Display image
    cv2.imshow("result_image", result_image)
    cv2.imshow("depth_image", depth_to_color_image)
    key = cv2.waitKey(1)
```

Grasping status callback function

```
def GraspStatusCallback(self,msg):
    #If received message data is True, indicating grasping completed, change
self.pubPos_flag, indicating next frame message can be published
    if msg.data == True:
        self.pubPos_flag = True
```

4.2. grasp.py

Code path:

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

Import necessary library files

```
import rclpy
from rclpy.node import Node
import numpy as np
from std_msgs.msg import Float32, Bool
import time
import math
from dofbot_pro_interface.msg import *
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         # ROS2 uses tf_transformations
import threading
from ament_index_python import get_package_share_directory
import yaml
import os
from Arm_Lib import Arm_Device
```

Open offset parameter table,

```
pkg_path = get_package_share_directory('dofbot_pro_driver')
offset_file = os.path.join(pkg_path,'config', 'offset_value.yaml')

with open(offset_file, 'r') as file:
    offset_config = yaml.safe_load(file)
print(offset_config)
print("----------------------------")
print("x_offset: ",offset_config.get('x_offset'))
print("y_offset: ",offset_config.get('y_offset'))
print("z_offset: ",offset_config.get('z_offset'))
```

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 robotic arm servo
control messages
    self.pub_point = self.create_publisher(ArmJoint, 'TargetAngle',1)
    #Create publisher for grasping results topic, publish grasping result
messages
    self.pubGraspStatus = self.create_publisher(Bool, 'grasp_done',1)
    #Create client for inverse kinematics service request, 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
    #Robotic arm initialization pose
    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 and 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
    #Read offset parameter table content, assign to offset parameters
    self.x_offset = offset_config.get('x_offset')
    self.y_offset = offset_config.get('y_offset')
    self.z_offset = offset_config.get('z_offset')
    #Print current robotic arm end position and pose
    print("Current_End_Pose: ",self.CurEndPos)
    print("Init Done")
```

AprilTag information callback function tag_info_callback,

```
def tag_info_callback(self,msg):
    pos_x = msg.x
    pos_y = msg.y
    pos_z = msg.z
    self.cur_tagId = msg.id
```

```
    #Check if received center point depth information is not 0, indicating valid
data
    if pos_z!=0.0:
        print("xyz id : ",pos_x,pos_y,pos_z,self.cur_tagId)
        #First coordinate system transformation, 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 transformation, 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)
        EndPointMat = self.get_end_point_mat()
        #Third coordinate system transformation, 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)
        #Add offset parameters to compensate for deviations caused by servo
value differences
        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)
        #Check grasping flag, True means lower gripper to grasp
    if self.grasp_flag == True :
        print("Take it now.")
        #Change grasping flag to prevent re-execution of grasping when
recognized during grasping process
        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,))
        #Execute thread
        grasp.start()
        grasp.join()
```

Robotic arm lower gripper 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
    request.tar_x = pose_T[0] -0.01
    #Target y value of robotic arm end, unit is m
    request.tar_y = pose_T[1]
```

```
    #Target z value of robotic arm end, unit is m, 0.2 is scaling factor, make
slight adjustments according to actual situation
    request.tar_z = pose_T[2] +
(math.sqrt(request.tar_y**2+request.tar_x**2)-0.181)*0.2
    #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:

        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]
        #Assign returned joint1-joint6 values from service call 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 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 pubTargetArm(self, joints, id=6, angle=180.0, runtime=2000):
        print(joints)
 self.Arm.Arm_serial_servo_write6(joints[0],joints[1],joints[2],joints[3],joints
[4],joints[5],2000)
```

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, 140, 2000)
    time.sleep(2.5)
    self.pubArm([],2, 120, 2000)
    time.sleep(2.5)
```

```
    #According to id value, change self.down_joint value, this value represents
the placement position of AprilTag block
    if self.cur_tagId == 1:
        self.down_joint = [150.0, 30, 70, 5, 90.0,140]
    elif self.cur_tagId == 2:
        self.down_joint = [180.0, 35, 60, 0, 90.0,140]
    elif self.cur_tagId == 3:
        self.down_joint = [28.0, 30, 70, 2, 90.0,140]
    elif self.cur_tagId == 4:
        self.down_joint = [0.0, 43, 48, 6, 90.0,140]
    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 completed, return to initial position
    self.pubArm(self.init_joints)
    time.sleep(5)
    #Wait for robotic arm to return to initial position, then publish grasping
completion message, change grasping flag value to allow next grasping when
conditions are met
    self.grasp_flag = True
    grasp_done = Bool()
    grasp_done.data = True
    self.pubGraspStatus.publish(grasp_done)
```