# EECS 4421 LAB4

## Name: Mahfuz Rahman

## Student No.: 217847518

My aruco\_target.py file  
  
import math

import numpy as np

import rclpy

from rclpy.node import Node

from rclpy.parameter import Parameter

import cv2

from cv\_bridge import CvBridge, CvBridgeError

from sensor\_msgs.msg import Image

from sensor\_msgs.msg import CameraInfo

from packaging.version import parse

from std\_msgs.msg import Bool # Import Bool message type

if parse(cv2.\_\_version\_\_) >= parse('4.7.0'):

def local\_estimatePoseSingleMarkers(corners, marker\_size, mtx, distortion):

marker = np.array([[-marker\_size /2, marker\_size / 2, 0],

[marker\_size /2, marker\_size / 2, 0],

[marker\_size /2, -marker\_size / 2, 0],

[-marker\_size /2, -marker\_size / 2, 0]],

dtype = np.float32)

trash = []

rvecs = []

tvecs = []

for c in corners:

nada, R, t = cv2.solvePnP(marker, c, mtx, distortion, False, cv2.SOLVEPNP\_IPPE\_SQUARE)

rvecs.append(R)

tvecs.append(t)

trash.append(nada)

return rvecs, tvecs, trash

class ArucoTarget(Node):

\_DICTS = {

"4x4\_100" : cv2.aruco.DICT\_4X4\_100,

"4x4\_1000" : cv2.aruco.DICT\_4X4\_1000,

"4x4\_250" : cv2.aruco.DICT\_4X4\_250,

"4x4\_50" : cv2.aruco.DICT\_4X4\_50,

"5x5\_100" : cv2.aruco.DICT\_5X5\_100,

"5x5\_1000" : cv2.aruco.DICT\_5X5\_1000,

"5x5\_250" : cv2.aruco.DICT\_5X5\_250,

"5x5\_50" : cv2.aruco.DICT\_5X5\_50,

"6x6\_100" : cv2.aruco.DICT\_6X6\_100,

"6x6\_1000" : cv2.aruco.DICT\_6X6\_1000,

"6x6\_250" : cv2.aruco.DICT\_6X6\_250,

"6x6\_50" : cv2.aruco.DICT\_6X6\_50,

"7x7\_100" : cv2.aruco.DICT\_7X7\_100,

"7x7\_1000" : cv2.aruco.DICT\_7X7\_1000,

"7x7\_250": cv2.aruco.DICT\_7X7\_250,

"7x7\_50": cv2.aruco.DICT\_7X7\_50,

"apriltag\_16h5" : cv2.aruco.DICT\_APRILTAG\_16H5,

"apriltag\_25h9" : cv2.aruco.DICT\_APRILTAG\_25H9,

"apriltag\_36h10" : cv2.aruco.DICT\_APRILTAG\_36H10,

"apriltag\_36h11" : cv2.aruco.DICT\_APRILTAG\_36H11,

"aruco\_original" : cv2.aruco.DICT\_ARUCO\_ORIGINAL

}

def \_\_init\_\_(self, tag\_set="apriltag\_36h10", target\_width=0.20):

super().\_\_init\_\_('aruco\_target')

self.get\_logger().info(f'{self.get\_name()} created')

self.declare\_parameter('image', "/mycamera/image\_raw")

self.declare\_parameter('info', "/mycamera/camera\_info")

self.\_image\_topic = self.get\_parameter('image').get\_parameter\_value().string\_value

self.\_info\_topic = self.get\_parameter('info').get\_parameter\_value().string\_value

self.create\_subscription(Image, self.\_image\_topic, self.\_image\_callback, 1)

self.create\_subscription(CameraInfo, self.\_info\_topic, self.\_info\_callback, 1)

self.\_bridge = CvBridge()

dict = ArucoTarget.\_DICTS.get(tag\_set.lower(), None)

if dict is None:

self.get\_logger().error(f'ARUCO tag set {tag\_set} not found')

else:

if parse(cv2.\_\_version\_\_) < parse('4.7.0'):

self.\_aruco\_dict = cv2.aruco.Dictionary\_get(dict)

self.\_aruco\_param = cv2.aruco.DetectorParameters\_create()

else:

self.\_aruco\_dict = cv2.aruco.getPredefinedDictionary(dict)

self.\_aruco\_param = cv2.aruco.DetectorParameters()

self.\_aruco\_detector = cv2.aruco.ArucoDetector(self.\_aruco\_dict, self.\_aruco\_param)

self.\_target\_width = target\_width

self.\_image = None

self.\_cameraMatrix = None

self.get\_logger().info(f"using dictionary {tag\_set}")

# Create a publisher for target visibility

self.\_visibility\_publisher = self.create\_publisher(Bool, '/target\_visible', 1)

def \_info\_callback(self, msg):

if msg.distortion\_model != "plumb\_bob":

self.get\_logger().error(f"We can only deal with plumb\_bob distortion {msg.distortion\_model}")

self.\_distortion = np.reshape(msg.d, (1,5))

self.\_cameraMatrix = np.reshape(msg.k, (3,3))

def \_image\_callback(self, msg):

self.\_image = self.\_bridge.imgmsg\_to\_cv2(msg, "bgr8")

grey = cv2.cvtColor(self.\_image, cv2.COLOR\_BGR2GRAY)

if parse(cv2.\_\_version\_\_) < parse('4.7.0'):

corners, ids, rejectedImgPoints = cv2.aruco.detectMarkers(grey, self.\_aruco\_dict)

else:

corners, ids, rejectedImgPoints = self.\_aruco\_detector.detectMarkers(grey)

frame = cv2.aruco.drawDetectedMarkers(self.\_image, corners, ids)

if ids is None:

self.get\_logger().info(f"No targets found!")

self.set\_target\_visible(False) # Set visibility to False

return

if self.\_cameraMatrix is None:

self.get\_logger().info(f"We have not yet received a camera\_info message")

return

# Set visibility to True when a target is found

self.set\_target\_visible(True)

if parse(cv2.\_\_version\_\_) < parse('4.7.0'):

rvec, tvec, \_objPoints = cv2.aruco.estimatePoseSingleMarkers(corners, self.\_target\_width, self.\_cameraMatrix, self.\_distortion)

else:

rvec, tvec, \_objPoints = local\_estimatePoseSingleMarkers(corners, self.\_target\_width, self.\_cameraMatrix, self.\_distortion)

result = self.\_image.copy()

for r,t in zip(rvec,tvec):

self.get\_logger().info(f"Found a target at {t} rotation {r}")

if parse(cv2.\_\_version\_\_) < parse('4.7.0'):

result = cv2.aruco.drawAxis(result, self.\_cameraMatrix, self.\_distortion, r, t, self.\_target\_width)

else:

result = cv2.drawFrameAxes(result, self.\_cameraMatrix, self.\_distortion, r, t, self.\_target\_width)

cv2.imshow('window', result)

cv2.waitKey(3)

def set\_target\_visible(self, visible):

""" Update the visibility of the target and publish the status. """

msg = Bool()

msg.data = visible

self.\_visibility\_publisher.publish(msg)

self.get\_logger().info(f"Target visibility set to {visible}")

def main(args=None):

rclpy.init(args=args)

node = ArucoTarget()

try:

rclpy.spin(node)

rclpy.shutdown()

except KeyboardInterrupt:

pass

if \_\_name\_\_ == '\_\_main\_\_':

main()

The drive\_to\_goal file:

import math

import numpy as np

import rclpy

from rclpy.node import Node

from rclpy.parameter import Parameter

from nav\_msgs.msg import Odometry

from geometry\_msgs.msg import Twist, Pose, Point, Quaternion

from std\_msgs.msg import Bool

from geometry\_msgs.msg import TransformStamped

def euler\_from\_quaternion(quaternion):

"""

Converts quaternion (w in last place) to euler roll, pitch, yaw

quaternion = [x, y, z, w]

"""

x = quaternion.x

y = quaternion.y

z = quaternion.z

w = quaternion.w

sinr\_cosp = 2 \* (w \* x + y \* z)

cosr\_cosp = 1 - 2 \* (x \* x + y \* y)

roll = np.arctan2(sinr\_cosp, cosr\_cosp)

sinp = 2 \* (w \* y - z \* x)

pitch = np.arcsin(sinp)

siny\_cosp = 2 \* (w \* z + x \* y)

cosy\_cosp = 1 - 2 \* (y \* y + z \* z)

yaw = np.arctan2(siny\_cosp, cosy\_cosp)

return roll, pitch, yaw

class move\_to\_goal(Node):

def \_\_init\_\_(self):

super().\_\_init\_\_('aruco\_robot\_controller')

# Publisher for cmd\_vel to control robot velocity

self.cmd\_vel\_pub = self.create\_publisher(Twist, '/cmd\_vel', 10)

# Subscriber to know if the target is visible

self.create\_subscription(Bool, '/target\_visible', self.target\_visible\_callback, 10)

# Subscriber to get the position and orientation of the target

self.create\_subscription(TransformStamped, '/target\_pose', self.target\_pose\_callback, 10)

# Subscriber to get the odometry data

self.create\_subscription(Odometry, '/odom', self.odom\_callback, 10)

self.target\_visible = False # Whether the target is visible or not

self.target\_distance = None # Distance to the target (None if not visible)

self.stop\_distance = 0.01 # Stop 1 cm away from the target

self.robot\_pose = None # Robot's current pose

self.robot\_yaw = None # Robot's current yaw angle

self.target\_position = None # Target's position (x, y)

self.timer = self.create\_timer(0.1, self.control\_loop) # Timer for control loop

def target\_visible\_callback(self, msg):

self.target\_visible = msg.data

def target\_pose\_callback(self, msg):

# Assuming the target's pose is relative to the robot's coordinate frame

# Translation (x, y, z) gives the position of the target

self.target\_position = (msg.transform.translation.x, msg.transform.translation.y)

self.target\_distance = math.sqrt(msg.transform.translation.x \*\* 2 + msg.transform.translation.y \*\* 2)

def odom\_callback(self, msg):

# Extract position and orientation (yaw) from odometry data

self.robot\_pose = msg.pose.pose.position

\_, \_, self.robot\_yaw = euler\_from\_quaternion(msg.pose.pose.orientation)

def control\_loop(self):

if self.robot\_pose is None or self.robot\_yaw is None:

self.get\_logger().info('Waiting for robot odometry...')

return

twist = Twist()

if not self.target\_visible:

# Target is not visible, spin the robot

twist.angular.z = 0.5 # Rotate at 0.5 rad/s

self.get\_logger().info('Spinning to search for target...')

elif self.target\_visible and self.target\_position is not None:

target\_x, target\_y = self.target\_position

self.get\_logger().info(f'Target visible at distance {self.target\_distance:.2f}m')

# Calculate angle to the target

angle\_to\_target = math.atan2(target\_y, target\_x)

# Adjust the angle based on the robot's current orientation (yaw)

angle\_difference = angle\_to\_target - self.robot\_yaw

# Normalize the angle difference to the range [-pi, pi]

angle\_difference = (angle\_difference + np.pi) % (2 \* np.pi) - np.pi

if self.target\_distance > self.stop\_distance:

# Move forward and adjust angular velocity to turn towards the target

twist.linear.x = 0.2 # Move forward at 0.2 m/s

twist.angular.z = 0.5 \* angle\_difference # Proportional control to turn towards target

self.get\_logger().info('Approaching the target...')

else:

# Stop the robot when within 1 cm distance

twist.linear.x = 0.0 # Stop moving forward

twist.angular.z = 0.0 # Stop rotating

self.get\_logger().info('Stopping the robot 1cm in front of the target.')

# Publish the velocity command

self.cmd\_vel\_pub.publish(twist)

def main(args=None):

rclpy.init(args=args)

node = move\_to\_goal()

try:

rclpy.spin(node)

except KeyboardInterrupt:

pass

finally:

node.destroy\_node()

rclpy.shutdown()

if \_\_name\_\_ == '\_\_main\_\_':

main()

Screen dump:

A screenshot of a computer

Description automatically generatedA computer screen shot of a blue and red object

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

To execute a long-duration mission, I structured the code by creating modular functions for distinct tasks such as target detection, position tracking, and robot odometry updates. I implemented a continuous control loop that checks if the target is visible, calculates the direction and distance to it, and adjusts the robot's movement accordingly. A timer was used to ensure the control loop runs regularly (e.g., every 0.1 seconds), allowing for quick responses to environmental changes. Additionally, I set stopping conditions to halt the robot when it approaches within 1 m of the target, preventing it from spinning indefinitely. This organized approach allows the robot to effectively handle the mission over an extended period.