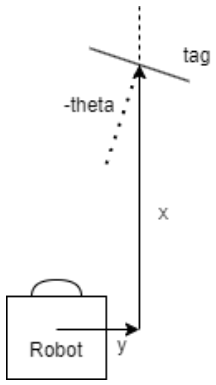
Auto Docking Code Documentation

Contents

General Approach

1. Spin until robot detects tag
2. Once tag is detected and the x, y and theta information are extracted



3. A point perpendicular from the tag z distance away is calculated with respect to the robot as origin.
4. Robot tries to move towards the point and face towards the tag using 3 maneuvers:
    a. Spin to face the point
    b. Move toward the point
    c. Spin to face the tag

    The first two maneuver is done by odometry and calculation of the point's distance and orientation from the robot. The last maneuver is done via closed loop feedback of the tag position/orientation detected.
5. Robot starts moving towards the tag and correcting its movement via PID closed loop control on the orientation and lateral distance error.

Code

Existing methods

Initialize robot

```
def __init__(self,x_offset=0,y_offset=0): #initialize robot with camera to
robot x,y offset
        self.x_offset=x_offset
        self.y_offset=y_offset
        self.xyt=XYT()
        self.odom=Odometry()
        self.tries=0
        self.pub=rospy.Publisher('/cmd_vel',Twist, queue_size=10)
        self.xytsub=rospy.Subscriber('/aruco_single/pose',PoseStamped,callback
=self.xyt_callback)
        self.odosub=rospy.Subscriber('/raw_odom',Odometry,callback=self.odo_ca
llback)
```

Initialize robot object and providing x,y camera offset. **Change subscriber/publisher name/topic here**

xyt_callback

```
 def xyt_callback(self,data:PoseStamped): #xyt subscriber callback
        x_offset=self.x_offset
        y_offset=self.y_offset
        self.xyt.x=data.pose.position.z+x_offset
        self.xyt.y=data.pose.position.x+y_offset
        orientation_q=data.pose.orientation
        orientation_list = [orientation_q.x, orientation_q.y, orientation_q.z,
orientation_q.w]
        (roll, pitch, yaw) = euler_from_quaternion (orientation_list)
        self.xyt.theta=-pitch
```

Used by ROS to store aruco tag detection data to global variable after some alteration to match robot position/orientation convention and camera offsets, do not call manually.

odo_callback

```
def odo_callback(self,data:Odometry):#odo subscriber callback
        #rospy.loginfo("receiving odom info")
        self.odom=data
        self.odom.twist.twist.linear.x=-data.twist.twist.linear.x
```

Used by ROS to store odometry data from robot after some alteration to match robot position/orientation convention, do not call manually.

`sign_function(x)`

```
def sign_function(x):
    if x > 0:
        return 1
    elif x == 0:
        return 0
    else:
        return -1
```

Python lacks this function, self-explanatory code

`turn(angle)`

```
def turn(self,angle):
        cmd=Twist()
        angle=angle%(2*math.pi)
        if angle>math.pi:
            angle=angle-2*math.pi
        t_est=0
        dt=1/100 #odom rate
        rate=rospy.Rate(100) #odom rate
        while abs(t_est-angle)>math.radians(1): #set angle threshold
            dtheta=self.odom.twist.twist.angular.z*1.17647#correction factor
for ang speed
            t_est+=dtheta*dt
            cmd.angular.z=0.3*self.sign_function(angle) #edit sign later
            self.pub.publish(cmd)
            rate.sleep()
        cmd.linear.x=0
        cmd.angular.z=0
        self.pub.publish(cmd)
```

Makes robot turn a certain amount of angle in radians via odometry.

Note: Angular speed measured by robot is off and is manually compensated here by 1/0.85

move(distance)

```
def move(self,distance):
        cmd=Twist()
        d_est=0
        dt=1/100 #odom rate
        rate=rospy.Rate(100) #odom rate
        while abs(abs(d_est)-abs(distance))>0.01: #set distance threshold
            dl=self.odom.twist.twist.linear.x*1#correction factor
            #rospy.loginfo(d_est)
            d_est+=dl*dt
            cmd.linear.x=-0.2*self.sign_function(distance) #linear speed
            #rospy.loginfo(cmd.linear.x)
            self.pub.publish(cmd)
            rate.sleep()
        cmd.angular.z=0
        cmd.linear.x=0
        self.pub.publish(cmd)
```

Makes robot move straight a certain amount forward in meters via odometry.

turn_to_tag(LoR)

```
    def turn_to_tag(self,LoR):
        cmd=Twist()
        while abs(self.xyt.y)>0.05:
            cmd.linear.x=0
            cmd.angular.z=-0.15*LoR
            self.pub.publish(cmd)
        cmd.linear.x=0
        cmd.angular.z=0
        self.pub.publish(cmd)
        self.reset_xyt()
        rospy.sleep(0.5)
        self.xyt=XYT()
        while abs(self.xyt.y)>0.01:
            cmd.linear.x=0
            cmd.angular.z=-0.04*self.sign_function(self.xyt.y)
            self.pub.publish(cmd)
        cmd.linear.x=0
        cmd.angular.z=0
        self.pub.publish(cmd)
```

Turn towards the tag given tag's left or right relative to the robot via camera closed loop control. Does it at 2 speeds to account for robot inertia slip after motor stops moving.

approach_tag()

```
def approach_tag(self): #change PID constants here
        cmd=Twist()
        #lateral errors
        KP_y=0.2
        KI_y=0.6
        KD_y=0.05
        pe_y=0
        sum_e_y=0
        #angle/orientation errors
        KP_t=0.9
        KI_t=0
        KD_t=0.05
        pe_t=0
        sum_e_t=0

        cmd.linear.x=-0.1 #linear speed
        dt=1/100 #odom rate
        rate=rospy.Rate(100) #odom rate
        prev_x=0
        tag_error=0
        while self.xyt.x>0.75: #set thershold distance here
            e_y=-self.xyt.y #y error
            e_t=self.xyt.theta #angle error
            cmd.angular.z=(KP_y*e_y+(e_y-pe_y)/dt*KD_y+sum_e_y*KI_y +
KP_t*e_t+(e_t-pe_t)/dt*KD_t+sum_e_t*KI_t)
            self.pub.publish(cmd)

            pe_t=e_t
            sum_e_t+=e_t*dt
            pe_y=e_y
            sum_e_y+=e_y*dt
            if self.xyt.x==prev_x:
                tag_error+=1
                if tag_error>=50:
                    rospy.loginfo("aruco tag out of view")
                    cmd.angular.z=0
                    cmd.linear.x=0
                    self.pub.publish(cmd)
                    rospy.loginfo(self.xyt)
                    break
            else:
                tag_error=0
            prev_x=self.xyt.x
            rate.sleep()
        cmd.angular.z=0
```

```
        cmd.linear.x=0
        self.pub.publish(cmd)
        rospy.loginfo("docking complete")
        rospy.loginfo(self.xyt)
```

Use 2 sets of PID control on lateral and orientation error to make robot approach the tag at fixed speed while correcting any movement error until it reaches a certain threshold distance and stops.
Set the threshold/stopping distance from the tag here

Change PID constants here(Beware, can lead to performance instability)

Start_Docking

```
def Start_Docking(self):
        cmd=Twist()
        has_init=1
        rospy.sleep(1)
        while self.xyt.x==0:
            if has_init:
                rospy.loginfo("spinning...")
                has_init=0
            cmd.angular.z=0.15
            cmd.linear.x=0
            self.pub.publish(cmd)

        cmd.angular.z=0
        cmd.linear.x=0
        self.pub.publish(cmd)
        rospy.sleep(3)

        set_distance=2
        self.tries=0
        while not (abs(self.xyt.y)<0.3 and
abs(self.xyt.theta)<=math.radians(4)): #change acceptable tolerance here
            self.tries+=1
            targetx=self.xyt.y+set_distance*math.cos(self.xyt.theta+3*math.pi/
2)
            targety=self.xyt.x+set_distance*math.sin(self.xyt.theta+3*math.pi/
2)
            target_theta=(math.atan2(targety,targetx)-math.pi/2)
            rospy.loginfo("Aruco tag x,y,theta:\n%s ",self.xyt)
            rospy.loginfo("tx,ty,tt: %s,%s,%s",targetx,targety,target_theta)
            #rospy.loginfo("LoR: %s",LoR)
            center_theta=self.modpi(target_theta)
            center_dist=math.sqrt(targetx**2+targety**2)
            rospy.loginfo("dist,theta: %s,%s",center_dist,center_theta)
            self.turn(center_theta)
            rospy.sleep(1)
            self.move(center_dist)
            rospy.sleep(1)
            LoR=self.sign_function(center_theta)
```

```
            self.turn_to_tag(LoR)
            rospy.sleep(1)
            set_distance-=0.1


        rospy.loginfo("start approaching tag")
        self.approach_tag()
```

Uses previous methods to dock the robot to the aruco tag, **change acceptable tolerance for the docking process here**. Higher tolerance leads to lower docking time and higher stability but loses accuracy.

If the alignment process fails, set distance is reduced by 10cm and the robot tries again. After 5 tries, the Robot is forced into approach tag no matter the alignment.

Code Usage Example

```
if __name__=='__main__':
    rospy.init_node('docking_controller')
    Initialize ROS node
    TARS4=Dock_Robot(0.22,0) #x_offset,y_offset
    Create robot object(TARS4) with camera offsets as input
    TARS4.Start_Docking()
    Run Start_Docking method to dock robot
```

The turn and move methods can be used to for other applications to move the robot a certain amount based on odometry.

Note:

- **Start_Docking requires Aruco tag detection code to be running to work**
- **Code in this document is most likely not up to date and only serves as an example/baseline to work with.**
- **Dock_Robot_Sim class works the same as Dock_Robot but parameters and adjusted to be used for gazebo simulation-waffle turtlebot.**

Math/Equations used

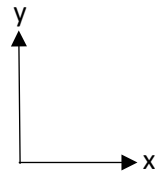
Calculating target point

$$Targetx = y_{tag} + set\ distance * cos(theta_{tag} + \frac{3\pi}{2})$$

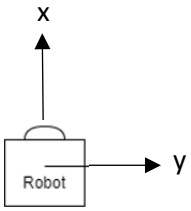
$$Targety = x_{tag} + set\ distance * sin(theta_{tag} + \frac{3\pi}{2})$$

$$Target\ theta = \tan^{-1}(\frac{targety}{targetx}) - \frac{\pi}{2}$$

- The Robot uses different frame of coordinates than the world frame due to conflicting convention between 2D and 3D world.



World Frame



Robot Frame

Note: In the real robot, the camera is placed on the back of the robot and all linear velocity needs to be multiplied by -1

- The tag angles are increased by 3pi/2 to shift the 0-angle point to 270deg relative to the standard convention. Same for the target theta.

Known issues

- Turn to tag method sometimes misses the aruco tag and turns a full circle which takes quite a while.
- The floor surface greatly affects the performance of the robot.
- Bad lighting condition can cause docking to take longer
- The auto-docking process is unreliable when the tag is 2m away from the robot

Possible Improvements

- Use more complex controllers to correct orientation and lateral errors in the final step.
- Use higher framerate camera/optimize the image post processing to allow the detection software to detect the tag reliably from 2m or more away