CMSC 477 Project 1: Plan Your Way Out Using Landmarks

(USING 1 LATE DAY)

Shanay Kadakia
Eric Kim
Trevor Moore

Introduction

This project aimed to plan and follow the shortest path through a known maze, while utilizing landmarks for localization. The walls consist of blocks (side length 0.266 m) placed within a grid. Unique AprilTags with known IDs and locations are periodically placed throughout the maze. Using the camera onboard the Robomaster EP Core, these AprilTags can be scanned and an inbuilt function can identify the camera's position relative to the tag.

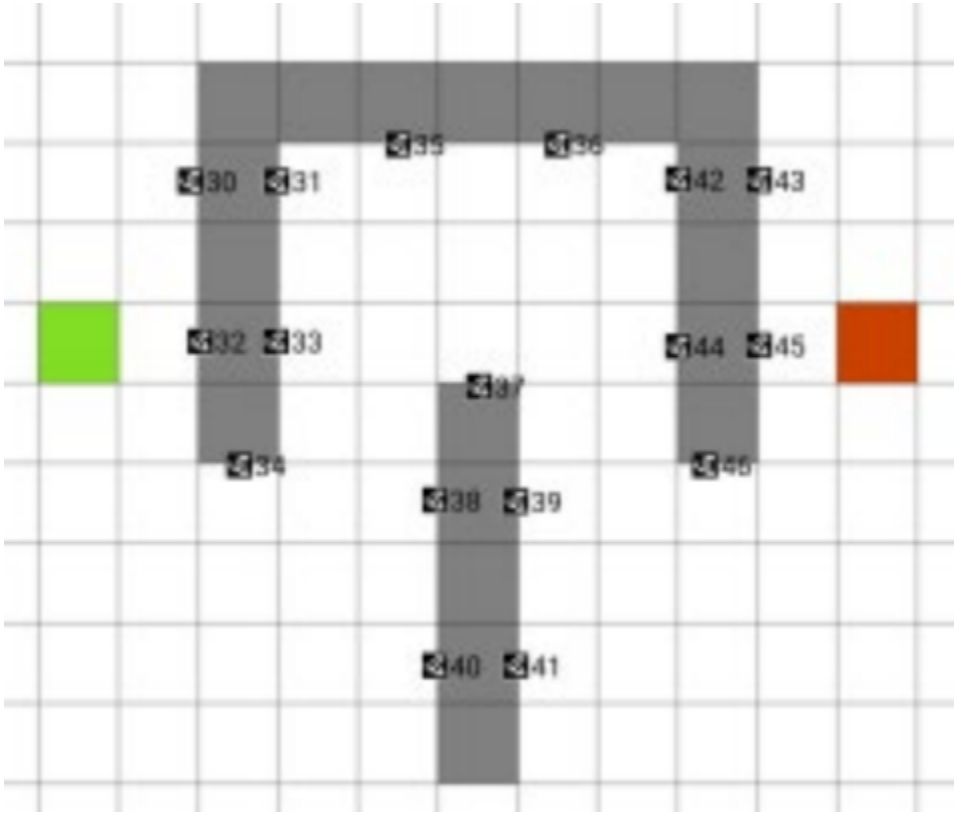

Figure 1: The maze. (start = green, goal = red, walls = gray)

Block Diagram

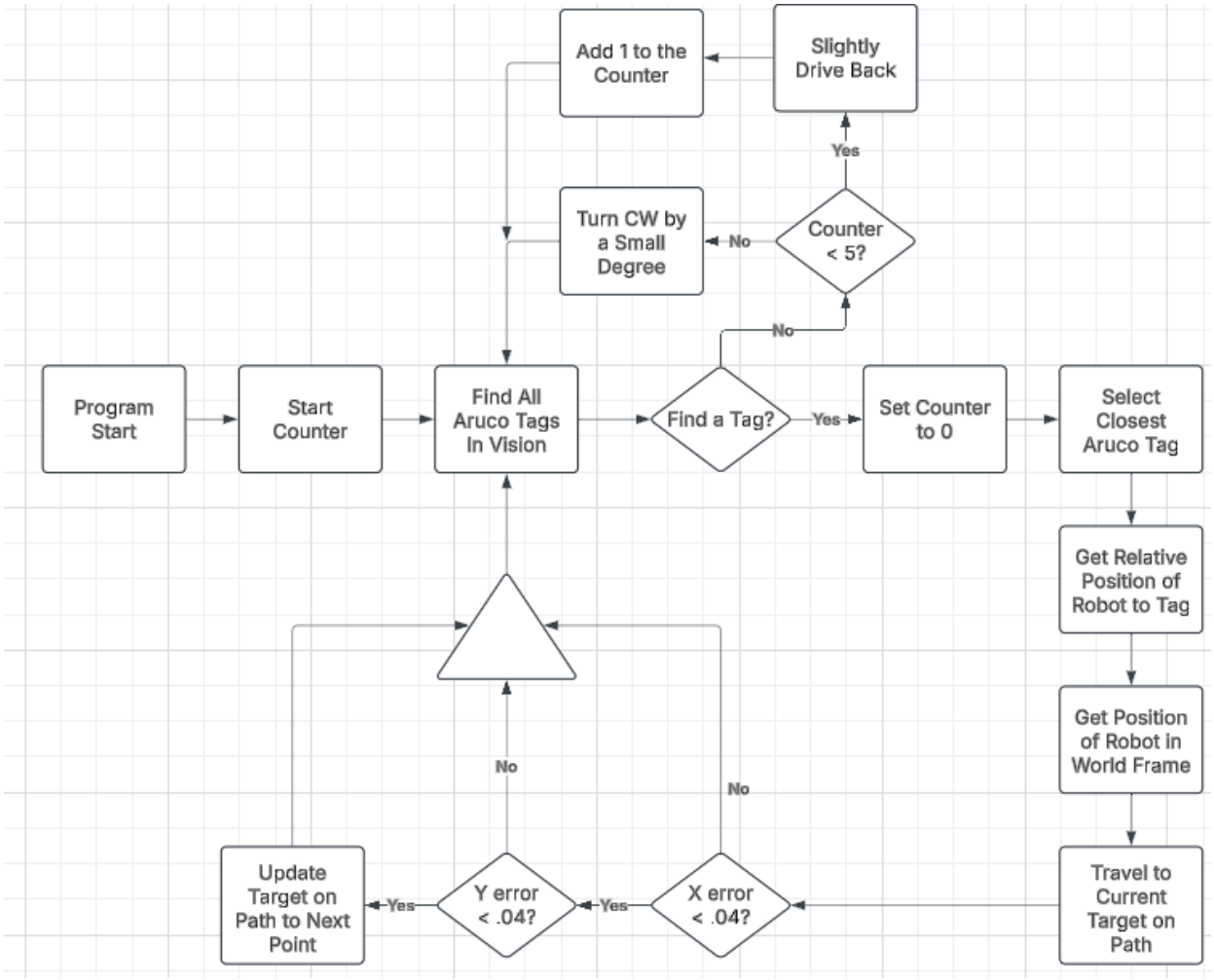

Figure 2: Decision making process for the program

Demonstration

https://youtube.com/shorts/qIS1KvEJMS4

Within the video, notice that the robot first aligns itself with the world coordinate system by navigating to the start point. Then, the robot navigates itself through solely translating to simplify control. At 00:36, the robot gets too close to the AprilTag, losing sight of the full tag, falling into its lost tag loop, rotating clockwise. It then detects a new tag in a new direction and begins navigating again. This cycle continues until the robot reaches the other side of the course.

Methodology

The robot navigation code consists of three core components: path planning, perception, and motion. These three components work in tandem so that path planning provides the coordinates for the robot to move in and perception allows the robot to be aware of where it is relative to obstacles and the target coordinates. With context provided by perception, motion moves the robot from its current position to the desired position using a control loop.

Path planning is based on the shortest path, which is found via Dijkstra's algorithm. The obstacles are modeled with comma separated values (CSV) file along a grid system shown in the following figure.

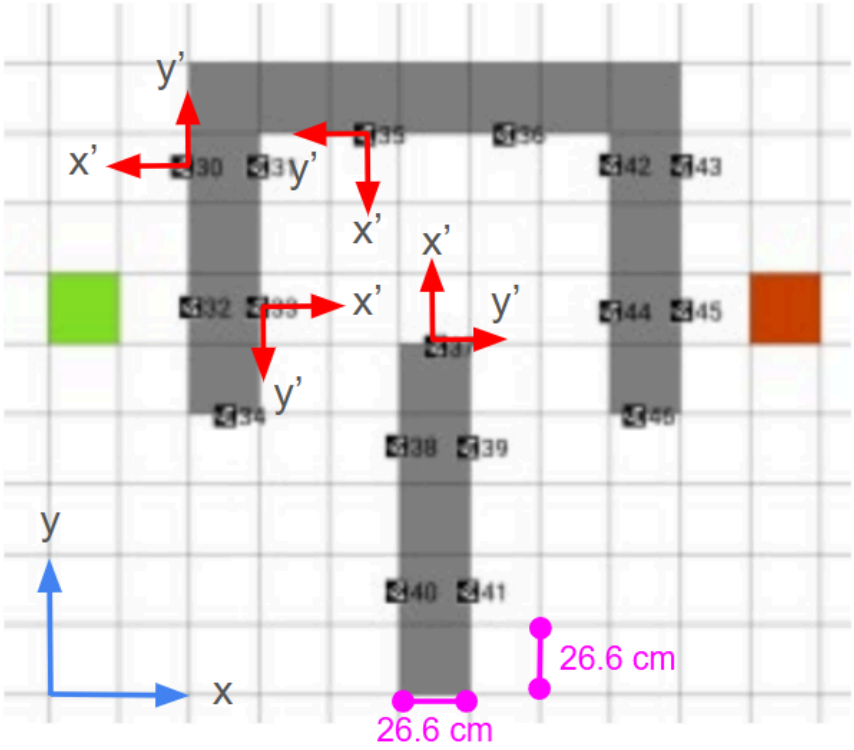

Figure 3: Maze Coordinate Frame and Real Coordinate Correlation

From Figure 3, the blue arrows represent the primary axis that the grid system refers to. Each grid is subdivided into a 9 x 9 grid representation to get more interpolation on the path. The actual grid size (not shown in Figure 3) is 8.9 cm in real measurements. This map of the

obstacles alone will not be enough to navigate the robot through, and some adjustments need to be made. Notably, if the robot got too close to a tag, its gripper claws would block the AprilTag from being viewed. Additionally, Dijkstra's algorithm has a tendency to hug the walls, leaving no room for the physical size of the robot and resulting in collisions with the obstacles. The map was modified in the created CSV to include a buffer for the robot to avoid, preventing collisions and obstruction of the tags, as shown in Figure 4.

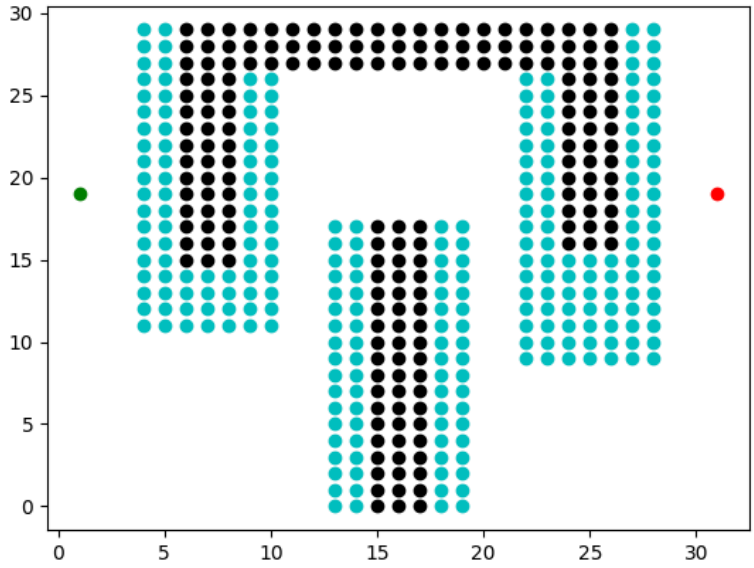

Figure 4: Grid system in actual implementation

From Figure 4, the green dot represents the start position, and the red dot represents the end position. Black dots represent hard obstacles (the boxes), and cyan dots represent the boundary to stay away from to avoid obstruction of the camera or clipping with the walls.

As for the perception algorithm, each AprilTag was mapped to the real coordinate frame using the aforementioned grid system in Figure 3. Each AprilTag was positioned in the center of the edge of a grid, and each tag could be on one of four edges, based on where it was adhered to in real life. The AprilTags normal vectors were also expressed in terms of the rotation needed to align the normal vector to the grid system main axes. Both position and orientation of the

AprilTags were saved in a dictionary with each tag ID. Table 1 shows the positions of the AprilTags relative to the grid system.

Table 1: X and Y Positions of each tag in the grid system

Tag ID	X-Position	Y-Position	Rotation
30	0.532	1.995	0
31	0.798	1.995	180
32	0.532	1.463	0
33	0.798	1.463	180
34	0.665	1.064	270
35	1.197	2.128	270
36	1.729	2.128	270
37	1.463	1.33	90
38	1.33	0.931	0
39	1.596	0.931	180
40	1.33	0.399	0
41	1.596	0.399	180
42	2.128	1.995	0
43	2.394	1.995	180
44	2.128	1.463	0
45	2.394	1.463	180
46	2.261	1.064	270

Once an AprilTag is detected, the tag's position can be found by searching the corresponding ID in the dictionary. In order to get the most accurate distance readings, the perception algorithm is programmed to identify both the closest tag and the tag the robot was

most directly viewing. In order to find the bearings of the robot, the relative position from the tag to robot is rotated to the grid system's axes, utilizing the rotation values in the dictionary. The rotated, relative positions are then added to the tag's absolute position in the grid system.

The motion algorithm uses a proportional control loop utilizing the error between the actual and desired coordinates to drive a velocity constant in x and y-directions. The robot is also commanded to constantly be normal to the AprilTag, which drives the rotational velocity. In nominal operations, the robot only needs to strafe, keeping one or more AprilTags in view to orient itself in the correct coordinates. Once it sees no AprilTags in its view, the robot will slowly rotate clockwise until it sees a valid AprilTag to find its bearings.

Results

Once Dijkstra's algorithm is completed, the shortest path is found between the start and goal position. Figure 5 shows the planned path in red, with blue being the algorithm's visited nodes.

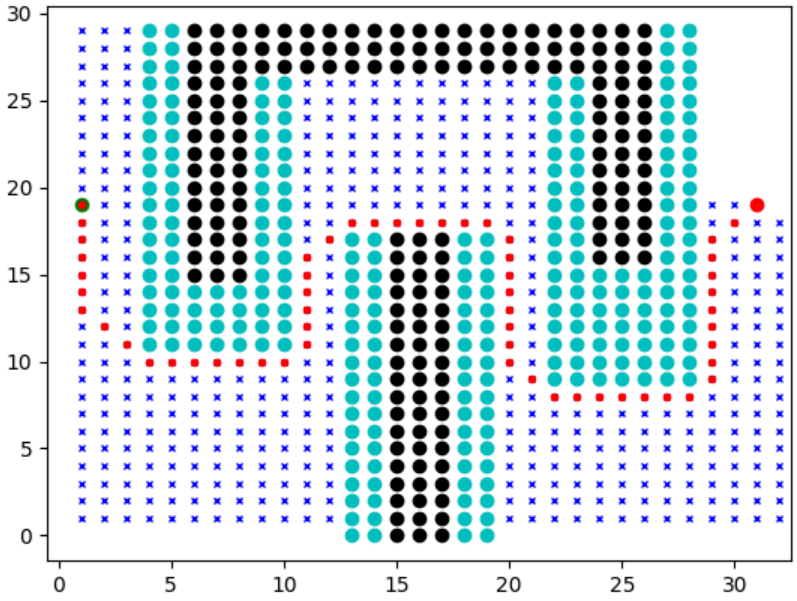

Figure 5: Dijkstra execution map

The robot followed the coordinates in succession, reaching the goal point with an actual path shown in Figure 6 as the green path.

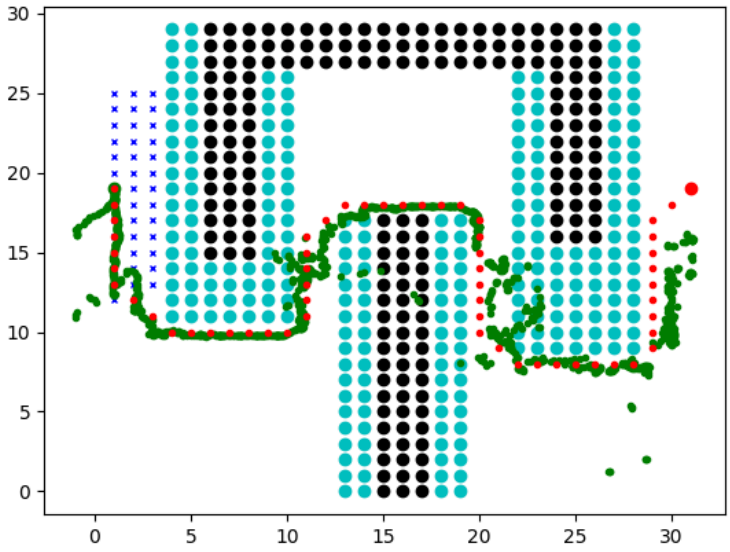

Figure 6: Robot actual path

Note that the actual path has some anomalous positions far off the desired path due to errors in how the robot's position was tracked in real coordinates using the AprilTags.

Conclusion

Throughout this project, the path planning challenge was fulfilled, but there are still various points of improvement which could be implemented with some more time.

Rather than creating a buffer manually, Dijkstra's algorithm could be modified to automatically account for the robot's dimensions to avoid collisions with the walls. Additionally, this would aid with the robot getting too close to the wall, losing sight of the full tag due to the gripper. These changes would bring the actual path closer to the true shortest path as the buffer would not be created arbitrarily.

The perception/localization algorithm could have used multiple tags to use some sort of triangulation rather than depending on the closest tag to provide a more precise location in the world frame.

As for the motion algorithm, losing sight of all tags poses a serious problem. Currently, the robot drives backward and turns slightly to try and find a tag. Instead, it could remember its last position in the world frame and move forward based on that, using new tags to minimize error as it progresses through the maze.

Code Listing

dijkstra.py - path planning algorithm

```
import csv
import matplotlib.pyplot as plt
import matplotlib.animation as animation
import time
import ast
import os
import numpy as np

class DJI:
    def __init__(self, givenFilepath):
        self.filepath = givenFilepath
        self.matrix = []

        # Finding the start and end
        self.startRow = 0
        self.startCol = 0
        self.createMatrix()

        # Plotting variables
        self.xs = []
        self.ys = []
        self.start = []
        self.end = []

        # Result and path variables
        self.resultx = []
        self.resulty = []

        self.pathx = []
        self.pathy = []
        self.robotx = []
        self.roboty = []

        self.interpx = []
        self.interpy = []
```

```
def createMatrix(self):
    currDir = os.getcwd()
    path = os.path.join(currDir, self.filepath)
    with open(path, mode="r") as file:
        for lines in csv.reader(file):
            self.matrix.insert(0,lines)

def giveStart(self):
    for row in self.matrix:
        for element in row:
            if element == '2':
                self.startRow = self.matrix.index(row)
                self.startCol = row.index(element)
                return (self.startRow, self.startCol)

def giveEnd(self):
    for row in self.matrix:
        for element in row:
            if element == '3':
                self.startRow = self.matrix.index(row)
                self.startCol = row.index(element)
                return (self.startRow, self.startCol)

def dist(self, x1, x0):
    return float(((x1[0]-x0[0])**2+(x1[1]-x0[1])**2)**0.5)

def initPlot(self):
    self.xs = []
    self.ys = []
    self.buffer_x = []
    self.buffer_y = []
    self.start = []
    self.end = []

    for row in range(0, len(self.matrix[0])):
        for col in range(0, len(self.matrix)):
            if self.matrix[col][row] == '2':
                self.start.append(row)
```

```
                    self.start.append(col)
                elif self.matrix[col][row] == '3':
                    self.end.append(row)
                    self.end.append(col)
                elif self.matrix[col][row] == '1':
                    self.xs.append(row)
                    self.ys.append(col)
                elif self.matrix[col][row] == '4':
                    self.buffer_x.append(row)
                    self.buffer_y.append(col)

    def interp(self):
        self.interpx = np.arange(1, 31, 0.2)
        self.interpy = np.interp(self.interpx, self.pathx[::-1],
self.pathy[::-1])

    def plot(self, pathBool=False):
        fig, ax = plt.subplots()
        scalingFactor = 0.266/3

        self.graph = ax.plot(self.xs,self.ys,'ko')
        self.graph = ax.plot(self.buffer_x,self.buffer_y,'co')
        self.graph = ax.plot(self.start[0],self.start[1],'go')
        self.graph = ax.plot(self.end[0],self.end[1],'ro')
        self.graph = ax.plot([],[], 'bx', markersize = 3)[0]

        ani = animation.FuncAnimation(fig=fig, func=self.animate, frames=500,
interval=0)
        if pathBool:
            with open("path.txt", "r") as file:
                lines = file.readlines()
                for string in lines:
                    string = string.strip()
                    string_list = string.split(",")
                    self.robotx.append(float(string_list[0])/scalingFactor)
                    self.roboty.append(float(string_list[1])/scalingFactor)

            ax.plot(self.robotx, self.roboty, 'go', markersize=3)
            ax.plot(self.pathx, self.pathy, 'ro', markersize=3)
```

```
        plt.show()

    def search(self):
        unvisited = {}
        visited = {}
        map = {}
        path = []

        for row in range(len(self.matrix)):
            for ind, element in enumerate(self.matrix[row]):
                look = [row, ind]
                if element == '0' or element == '3':
                    location = (row, ind)
                    unvisited[str(location)] = float('inf')

        unvisited[str(self.giveStart())] = 0
        start = time.time()

        while unvisited:
            unvisited = dict(sorted(unvisited.items(), key=lambda item: item[1]))

            look = ast.literal_eval(next(iter(unvisited)))
            shortest = unvisited[str(look)]

            if self.matrix[look[0]][look[1]] != '1':
                adjacent = [(look[0]-1, look[1]), (look[0]+1, look[1]), (look[0],
look[1]-1), (look[0], look[1]+1), (look[0]-1, look[1]-1), (look[0]+1, look[1]+1),
(look[0]+1, look[1]-1), (look[0]-1, look[1]+1)]
                for adj in adjacent:
                    if (str(adj) in unvisited) and adj[0] > 0 and adj[1] > 0:
                        distance = self.dist(adj, look)+unvisited[str(look)]
                        if distance < unvisited[str(adj)]:
                            unvisited[str(adj)] = distance

                        if not(str(adj) in map):
                            map[str(adj)] = str(look)

                visited[str(look)] = unvisited[str(look)]
```

```
                del unvisited[str(look)]

            if self.matrix[look[0]][look[1]] == '3':
                print("Found goal: " + str(look))
                print("Steps taken: " + str(len(visited)))
                print("Shortest Path: " + str(shortest))
                print("Time taken: " + str(time.time()-start))
                break

        curr = str(self.giveEnd())
        while (curr != None):
            try:
                path.append(curr)
                curr = map[curr]
            except KeyError:
                break

        for point in visited:
            point = point[1:-1].split(", ")
            self.resultx.append(int(point[1]))
            self.resulty.append(int(point[0]))

        for nodes in path:
            nodes = nodes[1:-1].split(", ")
            self.pathx.append(int(nodes[1]))
            self.pathy.append(int(nodes[0]))

        self.interp()

    def animate(self, frame):
        self.graph.set_xdata(self.resultx[:frame*10])
        self.graph.set_ydata(self.resulty[:frame*10])
```

motion.py - perception and motion control loops

```
import pupil_apriltags
import cv2
import numpy as np
import time
```

```
import traceback
from queue import Empty
import robomaster
from robomaster import robot
from robomaster import camera
import matplotlib.pyplot as plt
import matplotlib.animation as animation
from dijkstra import DJI
import multiprocessing

class AprilTagDetector: # Given
    def __init__(self, K, family="tag36h11", threads=2, marker_size_m=0.16):
        self.camera_params = [K[0, 0], K[1, 1], K[0, 2], K[1, 2]]
        self.marker_size_m = marker_size_m
        self.detector = pupil_apriltags.Detector(family, threads)

    def find_tags(self, frame_gray):
        detections = self.detector.detect(frame_gray, estimate_tag_pose=True,
            camera_params=self.camera_params, tag_size=self.marker_size_m)
        return detections

'''
(Dictionary)
Desc: describes the following april tag numbers into its real coordinate
position and the normal unit vector of the tag in real space
'''
# X, Y coordinates, X and Y orientation
tagDict = {
    30: [[0.532, 1.995, 0], [1, 1, 0]],
    31: [[0.798, 1.995, 0], [1, 1, np.pi]],
    32: [[0.532, 1.463, 0], [1, 1, 0]],
    33: [[0.798, 1.463, 0], [1, 1, np.pi]],
    34: [[0.665, 1.064, 0], [1, 1, np.pi*3/2]],
    35: [[1.197, 2.128, 0], [1, 1, np.pi*3/2]],
    36: [[1.729, 2.128, 0], [1, 1, np.pi*3/2]],
    37: [[1.463, 1.33, 0], [1, 1, np.pi/2]],
    38: [[1.33, 0.931, 0], [1, 1, 0]],
    39: [[1.596, 0.931, 0], [1, 1, np.pi]],
    40: [[1.33, 0.399, 0], [1, 1, 0]],
```

```
    41: [[1.596, 0.399, 0], [1, 1, np.pi]],
    42: [[2.128, 1.995, 0], [1, 1, 0]],
    43: [[2.394, 1.995, 0], [1, 1, np.pi]],
    44: [[2.128, 1.463, 0], [1, 1, 0]],
    45: [[2.394, 1.463, 0], [1, 1, np.pi]],
    46: [[2.261, 1.064, 0], [1, 1, np.pi*3/2]]
}

scalingFactor = 0.266/3

'''
Desc: gets the april tag orientation in Euler Angles (Roll, Pitch, Yaw)
and position of the tag with corrected pitch (accounting for camera)

Input: detection object (comes from AprilTagDetector)
Output: t_ca (position array 1x3), and rotation (1x3)
'''
def get_pose_apriltag_in_camera_frame(detection):
    R_ca = detection.pose_R
    t_ca = detection.pose_t

    roll = np.arctan2(R_ca[1][0],R_ca[0][0])
    yaw = np.arctan2(-R_ca[2][0],np.sqrt(R_ca[2][1]**2+R_ca[2][2]**2))
    pitch = np.arctan2(R_ca[2][1],R_ca[2][2])
    const = 1 #180/np.pi

    rotation = [const*roll, const*yaw, const*pitch]

    t_ca = t_ca.flatten()
    t_ca[2] = t_ca[2]*np.cos(pitch)
    t_ca[0] = t_ca[0]*np.cos(yaw)

    return t_ca, rotation

def draw_detections(frame, detections): # Given
    for detection in detections:
        pts = detection.corners.reshape((-1, 1, 2)).astype(np.int32)
```

```
        frame = cv2.polylines(frame, [pts], isClosed=True, color=(0, 0, 255),
thickness=2)

        top_left = tuple(pts[0][0])  # First corner
        top_right = tuple(pts[1][0])  # Second corner
        bottom_right = tuple(pts[2][0])  # Third corner
        bottom_left = tuple(pts[3][0])  # Fourth corner
        cv2.line(frame, top_left, bottom_right, color=(0, 0, 255), thickness=2)
        cv2.line(frame, top_right, bottom_left, color=(0, 0, 255), thickness=2)

'''
Desc: Proportional control loop for the robot to follow an April tag

Input:
- robot_pos: Robot current position
- dest_pos: Desired destination
Output: going to a desired destination
'''
def control_loop(ep_robot, ep_chassis, robot_pos, dest_pos, Rpose, angle):
    Px = 1.5
    offsetX = 0

    Py = Px
    offsetY = 0

    Pz = 300

    # pi/2 = straight up +y
    # 0 = looking to the +x (starting position)
    # pi = looking to the -x

    errorX = dest_pos[0]-robot_pos[0]-offsetX
    errorY = dest_pos[1]-robot_pos[1]-offsetY

    if(abs(errorX) < 0.02):
        errorX = 0
    if(abs(errorY) < 0.02):
        errorY = 0
```

```
    velx = Px*(errorX)
    vely = -Py*(errorY)
    velz = Pz*(Rpose[1])

    abs_velx = velx*np.cos(angle) - vely*np.sin(angle)
    abs_vely = velx*np.sin(angle) + vely*np.cos(angle)

    # if(abs(errorX) < 0.02):
    #     velx = 0
    # if(abs(errorY) < 0.02):
    #     vely = 0
    # if(abs(Rpose[1]) < 0.044):
    #     velz = 0

    ep_chassis.drive_speed(x=abs_velx, y=abs_vely, z=velz, timeout = 0.05)

    return errorX, errorY, angle

'''
Desc: finds the nearest april tag in the camera view based on pure distance to
robot,
and has a cutoff based on how angled the tag is

Input: detections
Output: closestTag object
'''
def closest(detections):
    closestDist = float('inf')
    closestTag = 0
    yaw_lim = np.pi/3
    # yaw_lim=5

    for tag in detections:
        pos, rot = get_pose_apriltag_in_camera_frame(tag)
        if np.linalg.norm([pos[0], pos[2]]) < closestDist:
            if abs(rot[1]) < yaw_lim:
                closestDist = np.linalg.norm([pos[0], pos[2]])
                closestTag = tag
    if closestTag == 0:
```

```
        return None

    return closestTag

'''
Desc: converts the position in the maze to the real world

Input:
- tag: the object with the information on the tag
Output:
- abs_x, abs_y: the position of the robot in real world
'''
def relative2world(tag):
    tag_id = tag.tag_id
    pos, rot = get_pose_apriltag_in_camera_frame(tag)
    r_x = pos[2]
    r_y = pos[0]

    # yaw = rot[1]
    yaw = 0

    tgx = tagDict[tag_id][0][0]
    tgy = tagDict[tag_id][0][1]
    tgox = tagDict[tag_id][1][0]
    tgoy = tagDict[tag_id][1][1]
    multiple = tagDict[tag_id][1][2]//(np.pi/2)
    angle = tagDict[tag_id][1][2]

    x_rel = r_x*np.cos(yaw) - r_y*np.sin(yaw)
    y_rel = r_x*np.sin(yaw) + r_y*np.cos(yaw)
    # dims_unrotated = np.array([[x_rel*tgox, 0],[0, y_rel*tgoy]])
    # dims_rotated = np.rot90(dims_unrotated, multiple)
    x_rel_rot = x_rel*np.cos(angle) - y_rel*np.sin(angle)
    y_rel_rot = x_rel*np.sin(angle) + y_rel*np.cos(angle)

    x_abs = tgx-x_rel_rot*tgox
    y_abs = tgy+y_rel_rot*tgoy
    return x_abs, y_abs
```

```
'''
Solely to test how well robot can track it position in the real world

Input: april tag (in view of camera)
Output: pose/orientation of the robot (in console)

'''
def test_tag(ep_camera, apriltag):
    while True:
        try:
            img = ep_camera.read_cv2_image(strategy="newest", timeout=0.5)
        except Empty:
            time.sleep(0.001)
            continue

        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        gray.astype(np.uint8)
        detections = apriltag.find_tags(gray)

        if len(detections) > 0:
            tag = closest(detections)
            x, y = relative2world(tag)
            _, rot = get_pose_apriltag_in_camera_frame(tag)
            print(rot)

        draw_detections(img, detections)
        cv2.imshow("img", img)

        if cv2.waitKey(1) == ord('q'):
            break

'''
Without using motion commands, track the robot going through a maze.

Input: april tag
Output: mapping of the robot in the maze

'''
def detect_tag_loop(ep_camera, apriltag):
```

```
    # Constants
    yaw_lim = np.pi/3

    # Init Graphing Functions
    pathfinding = DJI("Project 1\\Lab1.csv")
    pathfinding.initPlot()
    fig, ax = plt.subplots()

    graph = plt.plot(np.array(pathfinding.xs)*scalingFactor,
np.array(pathfinding.ys)*scalingFactor,'ko')
    graph = ax.plot([],[],'go')[0]
    ax.set(xlim=[0, 3.5],ylim=[0, 3.5])
    plt.draw()
    plt.pause(0.0001)

    while True:
        try:
            img = ep_camera.read_cv2_image(strategy="newest", timeout=0.5)
        except Empty:
            time.sleep(0.001)
            continue

        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
        gray.astype(np.uint8)
        detections = apriltag.find_tags(gray)

        if len(detections) > 0:
            tag = closest(detections)
            x, y = relative2world(tag)
            Tpose, Rpose = get_pose_apriltag_in_camera_frame(tag)
            # print(f'Tag ID: {tag.tag_id}| X dist: {Tpose[0]} | Y dist:
{Tpose[2]} | Yaw: {180/np.pi*Rpose[1]}| Yaw (alt):
{180/np.pi*np.arctan(Tpose[0]/Tpose[1])}')
            # print(f'Tag ID: {tag.tag_id}| X dist: {Tpose[0]} | Y dist:
{Tpose[2]} | Ratio: {Tpose[0]/Tpose[2]}')

            # Graphing Functions
            graph.set_xdata(x)
            graph.set_ydata(y)
```

```
            plt.draw()
            plt.pause(0.00001)

        draw_detections(img, detections)
        cv2.imshow("img", img)

        if cv2.waitKey(1) == ord('q'):
            break

'''
For Project 0, this is the original loop that would follow an april tag
using a PI loop

Input: singular april tag (in camera view)
Output: robot follows april tag

'''
def motionControl(ep_robot, ep_chassis, ep_camera, apriltag):
    errorSumX = 0
    errorSumY = 0
    errorSumZ = 0

    plotStart = time.time()
    runTime = 0

    timeArray = []
    errorXPlot = []
    errorYPlot = []

    while True:
        startTime = time.time()

        try:
            img = ep_camera.read_cv2_image(strategy="newest", timeout=0.5)
        except Empty:
            time.sleep(0.001)
            continue

        gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
```

```
        gray.astype(np.uint8)

        detections = apriltag.find_tags(gray)

        if len(detections) > 0:
            # assert len(detections) == 1 # Assume there is only one AprilTag to
track
            detection = detections[0]

            t_ca, R_ca = get_pose_apriltag_in_camera_frame(detection)
            # print('t_ca', t_ca)
            # print('R_ca', R_ca)
            # print("poll")
            oldRunTime = runTime
            runTime = time.time()
            # sumX, sumY, sumZ, errorX, errorY = control_loop(ep_robot,
ep_chassis, t_ca, R_ca, errorSumX, errorSumY, errorSumZ, 0.03)
            # errorSumX = sumX
            # errorSumY = sumY
            # errorSumZ = sumZ

        draw_detections(img, detections)
        cv2.imshow("img", img)

        if cv2.waitKey(1) == ord('q'):
            plt.scatter(timeArray, errorXPlot)
            # plt.scatter(timeArray, errorYPlot)
            plt.xlabel("Time (sec)")
            plt.ylabel("Error (sec)")
            plt.show()
            break

'''
Using P loop and coordinates on the desired path, will follow the shortest path
from start to end goal

Input: path coordinates and april tags
Output: follow coordinates and orient itself using the tags in real world
```

```
'''
def maze_movement(ep_robot, ep_chassis, ep_camera, apriltag):
    # Constants
    plotStart = time.time()
    runTime = 0
    angle = 0
    tries = 0

    timeArray = []
    errorXPlot = []
    errorYPlot = []
    yaw_lim = np.pi/3

    # Init Graphing Functions
    pathfinding = DJI("Project 1\\Lab1.csv")
    pathfinding.initPlot()
    pathfinding.search()

    path_xcoord = np.array(pathfinding.pathx)*scalingFactor
    path_ycoord = np.array(pathfinding.pathy)*scalingFactor
    path_xcoord = path_xcoord[::-1]
    path_ycoord = path_ycoord[::-1]

    path_ind = 0

    fig, ax = plt.subplots()
    graph = plt.plot(np.array(pathfinding.xs)*scalingFactor,
np.array(pathfinding.ys)*scalingFactor,'ko')
    graph = plt.plot(path_xcoord, path_ycoord, 'ro', markersize=3)
    graph = ax.plot([],[],'go')[0]
    ax.set(xlim=[0, 3.5],ylim=[0, 3.5])


    plt.draw()
    plt.pause(0.0001)
    with open("path.txt", "w") as file:
        while True:
            try:
                img = ep_camera.read_cv2_image(strategy="newest", timeout=0.5)
```

```
            except Empty:
                time.sleep(0.001)
                continue

            gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)
            gray.astype(np.uint8)
            detections = apriltag.find_tags(gray)

            tag = closest(detections)
            if tag != None:
                tries = 0
                x, y = relative2world(tag)
                _, Rpose = get_pose_apriltag_in_camera_frame(tag)
                print(f'Tag ID: {tag.tag_id}| Tries: {tries} | Robot X: {x}|
Robot Y: {y}| Target X:{path_xcoord[path_ind]}| Target Y:
{path_ycoord[path_ind]}')

                tag_angle = tagDict[tag.tag_id][1][2]
                angle = 2*np.pi-tag_angle
                errorX, errorY, _= control_loop(ep_robot, ep_chassis, [x, y],
[path_xcoord[path_ind], path_ycoord[path_ind]], Rpose, angle)
                # print(f'Errors| Robot X: {errorX}, Robot Y: {errorY}')

                # Graphing Functions
                graph.set_xdata(x)
                graph.set_ydata(y)

                file.write(f"{x}, {y}\n")

                plt.draw()
                plt.pause(0.00001)

                if(abs(errorX) < 0.04) and (abs(errorY) < 0.04):
                    path_ind += 1
            else:
                if tries < 5:
                    ep_chassis.drive_speed(x=-0.1, y=0, z=0, timeout = 0.05)
                    tries += 1
                else:
```

```
                    ep_chassis.drive_speed(x=0, y=0, z=8, timeout = 0.05)

            draw_detections(img, detections)
            cv2.imshow("img", img)

            if cv2.waitKey(1) == ord('q'):
                break

if __name__ == '__main__':
    robomaster.config.ROBOT_IP_STR = "192.168.50.121"
    ep_robot = robot.Robot()
    ep_robot.initialize(conn_type="sta", sn="3JKCH8800100UB")

    ep_chassis = ep_robot.chassis
    ep_camera = ep_robot.camera
    ep_camera.start_video_stream(display=False, resolution=camera.STREAM_360P)

    K = np.array([[314, 0, 320], [0, 314, 180], [0, 0, 1]]) # Camera focal length
and center pixel
    marker_size_m = 0.153 # Size of the AprilTag in meters
    apriltag = AprilTagDetector(K, threads=2, marker_size_m=marker_size_m)

    try:
        # test_tag(ep_camera, apriltag)
        # detect_tag_loop(ep_camera, apriltag)
        maze_movement(ep_robot, ep_chassis, ep_camera, apriltag)
    except KeyboardInterrupt:
        pass
    except Exception as e:
        print(traceback.format_exc())
    finally:
        print('Waiting for robomaster shutdown')
        ep_camera.stop_video_stream()
        ep_robot.close()
```