New York University Abu Dhabi

ENGR-UH 3530 2022 Fall-001

Embedded Systems

Project Design Report

Author:
Manuel Padilla
Baraa Al Jorf

Student ID:
map971
baj321

Contents

1 Abstract

The project consists of creating an autonomous robot that is able to follow a black line while overcoming different problems in a modular field formed by tiles with different patterns. The tiles can be rearranged and made into an endless number of different courses for the robots to traverse. The tiles are 30cm x 30cm with different patterns which indicate the robot where to go as well as test the ability of the robot to handle different cases. The robot must be controlled completely autonomously and should not have any prior information about the track before running on it.

2 Introduction

While no single definition exists as to what an embedded system is, embedded systems are microprocessor- or microcontroller-based systems of hardware and software that generally share the following characteristics:

1. Embedded Systems are specialized to an application domain, meaning they are often created to handle a specific task
2. Embedded Systems are more specialized and are thus more challenging to design than general-purpose computers
3. Embedded Systems interact with the real world and require consideration of the constraints associated with it

Our project aims to demonstrate an embedded system that controls an autonomous robot by guiding it through a designated path. To achieve this, we follow a few embedded system design foundational columns.

In addition, this project explores the field of autonomous driving. Autonomous driving is one of the most revolutionary technologies of our time and has immense potential applications, from reducing traffic congestion, increasing road safety, providing emergency services, to industrial robots in warehouses. Autonomous vehicles will be able to drive themselves to the desired destination, eliminating the need for a human driver.

Autonomous vehicles could potentially revolutionize the way people commute, allowing for more efficient and cost-effective transportation options. Autonomous vehicles are equipped with advanced sensors that allow them to detect hazards on the roads such as other vehicles, pedestrians, cyclists, and animals. This project explores the use of computer vision in order to follow a determined path and detect objects to avoid collisions. For our software, we interweave computer vision and control system concepts to process an image input and obtain outputs that properly control

the robot's motions. The following sections demonstrate our system design and our methodology for teaching the Jetbot to navigate a specified path.

3 Problem Statement

The goal laid out is to design a robot's system to allow the robot to autonomously follow a black line while overcoming different problems in a modular field formed by tiles with different patterns. The floor is white in color and has different patterns that the robot has to be able to decipher in order to do the corresponding motions.

3.1 Patterns

- **Black lines:** Most of the tiles contain either one or a series of black lines that the robot has to attempt to follow. The tiles can have straight lines, curves, sharp turns, gaps (maximum of 20cm), or patterns meant to disrupt the robot's navigation. The images in the figures below show some of the example patterns that the tiles can include.

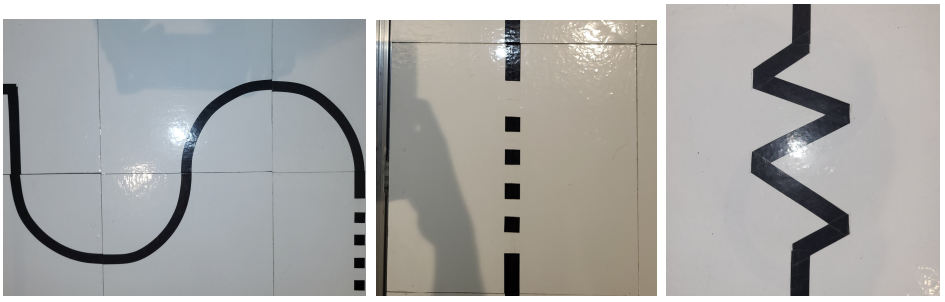

Figure 1: Left to right: curved lines, gaps, sharp turns

Figure 2: Examples of "disrupting" patterns

- **Intersections:** There are tiles that contain intersections in which the robot has to decide which way to go. To do this, the robot has to identify whether or not there are green squares somewhere on the tile and then determine the square's position with respect to the intersection. See the images below for the possible scenarios.

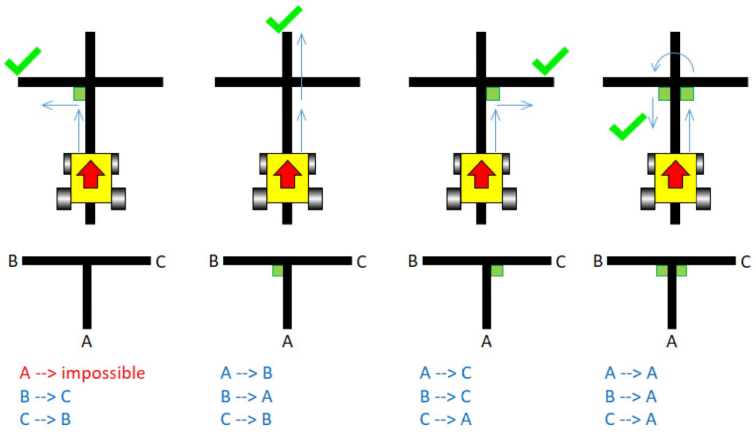

Figure 3: These are possible scenarios that the vehicle might face including the desired response from the vehicle.

- **Roundabout:** The roundabout uses the green square rules used at the intersections to indicate to the robot where to enter and where to exit. For example, in the roundabout shown below, if the robot enters through the left, the robot should turn right and take the long route until it exits going to the top tile.

Figure 4: Example of a roundabout in which the longer route is always taken

- **Finish Line:** The end of the path will be indicated with reflective silver reflective tape on the floor. When detected, the robot has to stop and can end its processing since the field has been completed.

Figure 5: Tin foil was used to indicate the finish line.

3.2 Obstacles

For this project, only one obstacle was presented during the navigation of the field. A little schoolgirl "Jane Doe" stands confidently in the middle of the road unaware of her surroundings, oblivious to the bright green car heading her way. An autonomous vehicle's goal is to arrive at its end destination; however, human safety is always the primary concern. Presenting the vehicle with an obstacle in its path will test its ability to handle corner cases that are necessary when developing an autonomous machine.

Figure 6: Frail schoolgirl Jane Doe

4 System Design

4.1 Solution Description

Although the problem above seems very complex, it is possible to be broken down into smaller tasks that the robot is able to handle. The main part of this process are being able to translate an image input into voltage signals that control the robot's motors according to what we desire.

- **Input** The robot will receive images from the built-in camera of the ground right in front of it. The camera has an 8MP HD resolution meaning that it produces images with file size 2448 x 3264 PPI which corresponds to an image size of 8" x 11". For our case, however, we will only use images that are 300 x 300. The main region of interest though will be at the bottom of the image since that portion of the image will determine what has to be done immediately after the previous motion has been done.
- **Intermediate Steps** Once the input image is received it will be processed using the python CV2 library functions in order to identify the pattern displayed on the tile. To determine the pattern, the tile will first be classified according to the objects that are present in the image. The image can be classified into three categories: no black lines, green squares, and black lines. The actions that will be performed for each case will be further explained in the implementation section as well as the method used to classify them.
- **Output** Once the robot detects the motions that it has to perform, the Jetbot will move by relaying the speed at which each wheel should move(e.g. $robot.set\_motors(speed = percentage, speed = percentage)$ which controls the speed of each individual wheel). The exact speed percentages will be determined by a proportional system controller and gains that have been optimized for the tasks at hand.

4.2 Flow Chart

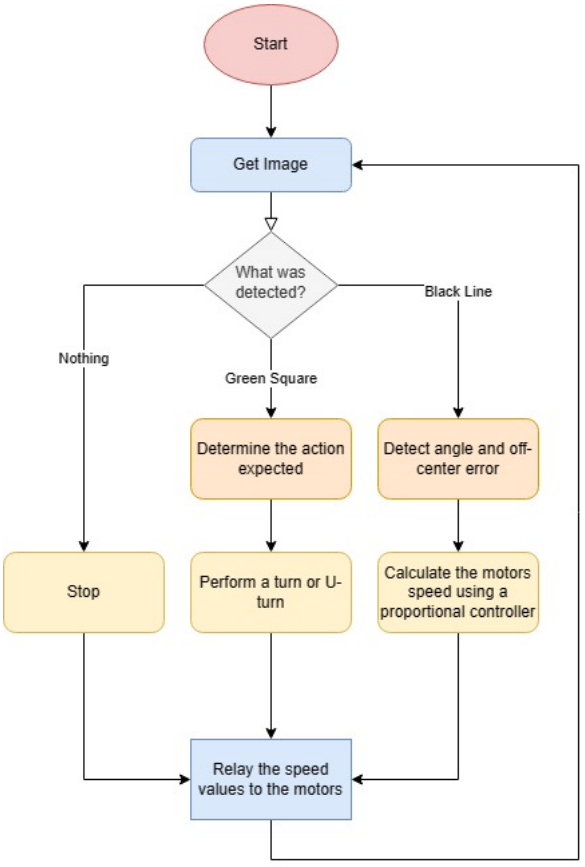

Figure 7: The flow chart above can be used to understand the logical flow of our implementation as it portrays the process of going from input to output through each possible classification.

4.3 Finite State Model

A finite-state machine-based system is used to control the robot. The camera will capture an image of the track directly ahead and classify it according to what is detected. Depending on the classification, the robot will perform a determined action (such as turning or stopping) or adjust the speed of the wheels to follow the road.

4.3.1 Possible States

1. **Classification Stage:** In this state the robot captures a picture of the environment and processes the data. The robot runs a contour detection algorithm

that attempts to determine whether black lines or green squares are on the image's region of interest. Depending on what is found in the image, the robot can classify as in the following categories: nothing, turning, or line following.

2. **Nothing:** If nothing is found, then the robot has to stop since the road is not found or there might be something blocking the road.
3. **Green Squares:** If a green square is detected, the robot determines where the green square lies with respect to the intersection in order to perform the following action. Once the action is detected the robot sets the motor values to a predetermined speed and sleeps for a short while as the robot is relocating to a new position. The goal of the predetermined speed is to direct the robot in the correct direction and face the next black line it has to follow. The values for the robot's speed and sleep time were determined through experimentation and optimized manually.
4. **Black Line:** If only black lines are found on the region of interest, then the robot selects the line closest to itself and calculates the distance of the line to its own center as well as the angle of the line in comparison to the direction it is headed. The difference between the distance of the centers and angles are considered as "errors" that are input through a proportional controller to obtain speed values for the motor that will reduce the error as best as possible.

5 Model Attributes

To better understand the properties of our model, we determine the following attributes that describe it.

- **Time Invariant-** The processing of data is done in constant time as well as the time that it will spend within each state. The model output does not rely on the time of execution, it only relies on the images that are provided.
- **Discrete Model-** Our model states are discrete and no continuity exists among them. (i.e there is no state that is halfway in between being idle and moving forward)
- **Event Driven-** The model is event driven meaning that occurrence of events (in this case receiving images) trigger the switch between states. This is in contrast to time-driven models that rely on time to switch states,
- **Deterministic-** There is no randomness to our model. For an input I that produces output O, any input X identical to I would also produce O. (not stochastic)

- **Linear-** Altering our input image matrices by multiplication or addition produces a similar alteration to the output. (output plot is linearly related to input image matrix)
- **Moore State Machine-** a Moore machine is a finite-state machine whose current output values are determined only by its current state. In our case, our output relies solely on the input images currently provided and the state the model is currently in.
- **Non Preemptive-** A state executes to completion and actions suggested by the received images are performed entirely before other images are processed.

6 Implementation Details:

As discussed previously, the robot relies on the images captured using its camera. The algorithm we developed follows the following steps:

1. The region of interest in the image captured by the robot is analyzed. This makes sure that the robot only performs actions that are determined using the part of the track that is closest to it.
2. The region of interest is then analyzed for black and green pixels with the cv2 function `cv.inRange(image, bottom threshold, top threshold)`. In this function, we provide RGB values in order to identify the pixels that fall within the specified range. For example, black we considered it to be within the range (0,0,0)-(68,68,68). Although ideally black is supposed to be 0, the reflected light causes the values to be slightly higher. Once the pixels are detected, the region of interest is eroded to remove any white noise and then dilated in order to return the image to its original size. For this we use the functions `cv.erode()` and `cv.dilate()`.
3. Once the pixels are detected, the region of interest and the information of the pixels detected is passed to the function `cv.findContours()`.
4. Assuming no green was detected, the black contours are added to a list of candidates for which a rectangle is drawn around them according to the smallest possible area (`cv.minAreaRect()`). Then the bottom corner, height, and with are obtained with the function (cv.boundingRect() from which it through simple algebraic calculations makes it possible to estimate the angle and distance that the line is from the center and pose of the robot.
5. The difference between the angle and distances are calculated and input into a control system that is able to determine the appropriate speeds required to

make the robot turn in the directions that it has to. All the constants of the controller; however, are hard-coded for optimal performance through a lot of experimentation.

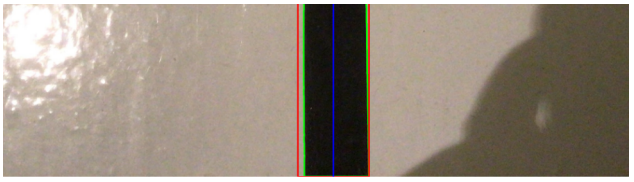

Figure 8: straight line-following

6. if a green box is detected, the center is located and compared to the center of the black contoured region. The robot then moves in the direction of the center.

7. If two green boxes are detected (signaling a U-turn), the center of the combined green boxes is going to be near the center of the black line. Therefore, if green boxes are located but the center is a short distance away from the black center, the robot performs a U-turn.

7 Results

The robot performs relatively well. The Proportional controller in particular follows the lines and completes the turns with high accuracy. Most of the disrupting conditions were able to be overcome except for the 20cm gap in which the robot was unable to detect the line it was supposed to follow. A problem that arose was when the robot identified a nearby line of a neighboring tile as if that was the road it was supposed to follow. This can be altered by fixing the region of interest to focus directly ahead of the robot and ignore the peripherals of the image. In addition, the green detection algorithm might misidentify the location of the green squares due to the surrounding noise, especially for the U-turn case. Because of the varying light conditions, the robot tends to detect green sometimes in places where there is not any. The U-turn case specifically faces an issue since the algorithm can give priority to the detection of one of the green squares before detecting the second square depending on which one is easier to detect and comes into the frame first. This causes the robot to turn to a side and sleep and completely miss the opportunity to detect both squares.

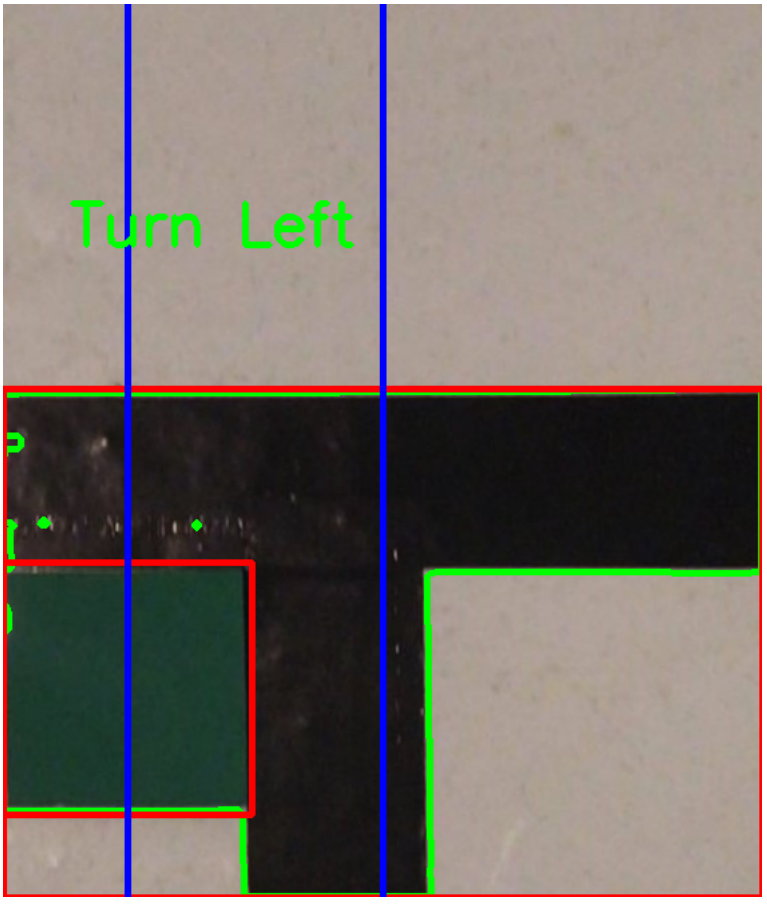

Figure 9: green box detected, left turn

The motors were also hard to control. The screws loosened and the robot's wheels lost grip leading the robot to struggle to keep moving and on track when crossing tiles. In some parts, that caused the robot to require a push in order to move/turn.

The collision avoidance was also functional for the finish line since once the black line was covered by the tin foil, the robot stopped. However, in the case of the little girl, the robot would detect black lines on the box (either the schoolgirl herself or the bar codes) and collide.

Overall, the robot was able to complete the path. For future consideration, if the computer vision approach is maintained, we believe that having a dynamic region of interest would help the robot to perform better in some of the obstacles such as the gaps and squares. However, this suggests that once one of these cases is reached it might be necessary to slow down or completely stop to allow the vehicle to inspect its surroundings until it finds the next indicators. Additionally, the performance of the robot was hindered due to poor construction and materials. Even with tightened screws, the robot wheels would wobble as the connection to the motor was weak. It might have also been beneficial to remove the front omnidirectional sphere since it would allow for more weight to lie on the large wheels and allowed them to have more traction. This would be possible since the center of gravity of the robot was mostly at the back of the vehicle.

8 Alternative Implementations:

The following section discusses alternative implementations of the algorithm that were experimented with but then retired.

8.1 Histograms:

Initially, the algorithm relied on histograms and followed the described steps below:

1. The image values are inverted, making the black lines white(this is not an essential step but is helpful in making more sense of the output)
2. image is converted to gray-scale (green matrix from RGB is also extracted but is not necessary)

3. The image is binarized: This basically means that the image values are represented using two values (either black or white- no in between)

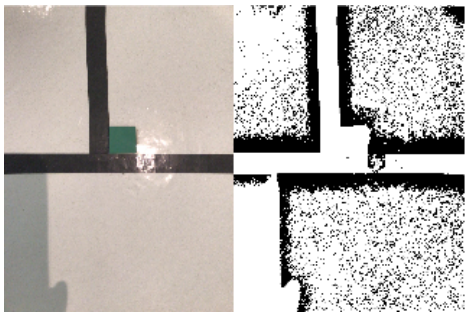

Figure 10: gray-scale conversion

4. A row sum (summing the values along each row) and a column sum (summing the values along each column) array are returned and plotted (green line is row sum and column sum of the green matrix)

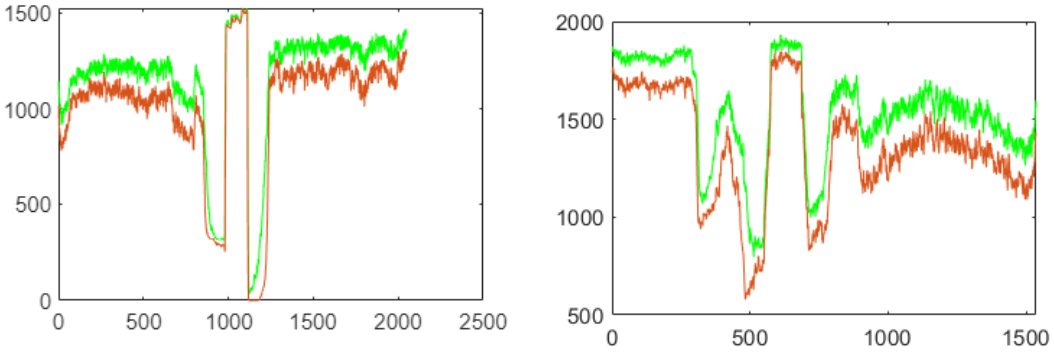

Figure 11: Plot of row sum (left) and plot of column sum (right)

5. the plots are studied for any remarkable jumps (determined through number of maxima)

6. Depending on number of maxima, determine direction (in presented example, 2 maxima in rows and 1 maximum in columns means robot needs to go left but 3 maxima in rows indicate robot needs to do a U-turn)

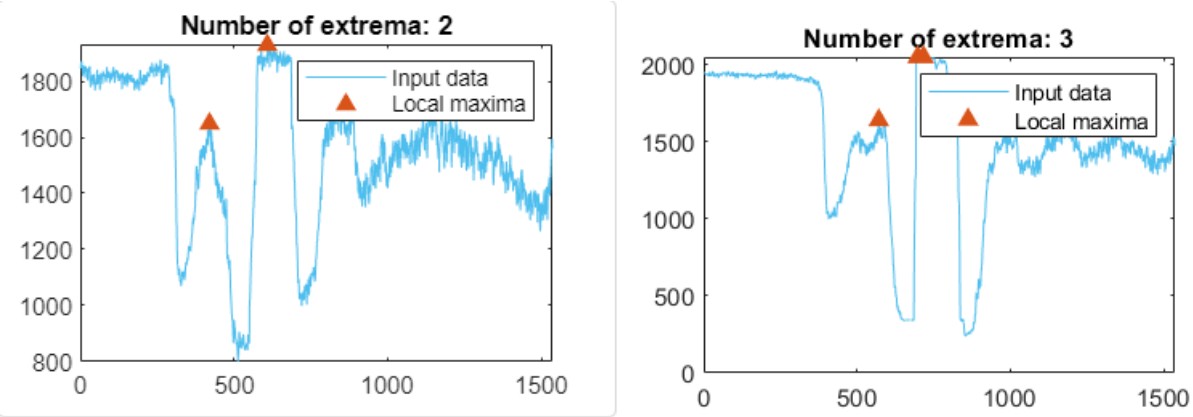

Figure 12: Plot of corner with 1 green sticker(left) vs plot of corner with 2 green stickers(right)

The histogram algorithm was unnecessarily complicated however and thus retired. The approach constantly relied on multiple comparisons (more computationally intensive) and was unreliable since lighting conditions heavily influenced how many maxima were detected.

8.2 Hough Lines:

The next approach relied on edge detection. This was performed using OpenCV's cv.canny() function and returned output that looked like the figures below.

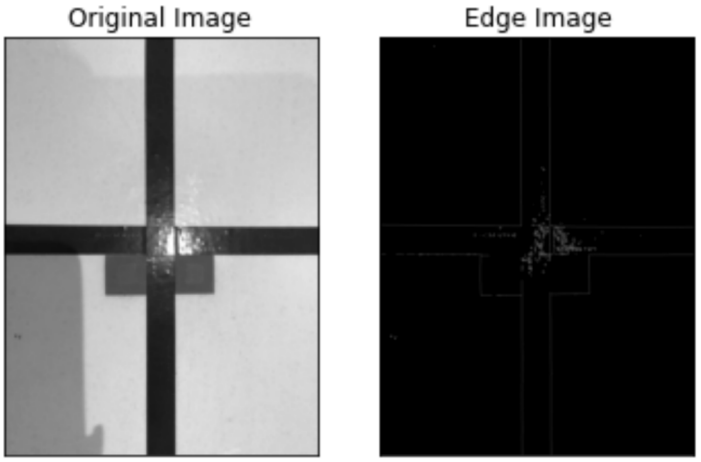

Figure 13: edge detection demonstrated

Figure 14: Canny Edges Close-up

After detecting the edges, they were fed into the cv.HoughLinesP() function which returned all detected lines that the edges form. This returned output like figure 12. As can be see, the lines are very unreliable and choppy. Consequently, using this approach would have returned inappropriate results. It was therefore retired.

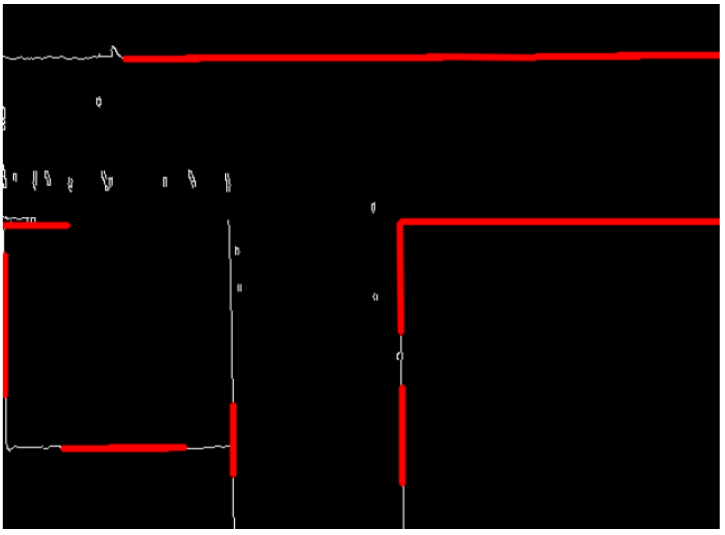

Figure 15: Choppy line formation using hough line detection algorithm

—

References

[1] Shafique, Mohammad, Embedded Systems Lecture Slides sets 1 & 2, Abu Dhabi. taught in 2022.

[2] "What is an embedded system? definition and faqs," What is an Embedded System? Definition and FAQs — HEAVY.AI. [Online]. Available: https://www.heavy.ai/technical-glossary/embedded-systems. [Accessed: 30-Oct-2022].

9 Appendix

```
#Road Following Car
#First import the library
from jetbot import bgr8_to_jpeg
import traitlets
import ipywidgets.widgets as ipywidgets
from jetbot import Camera
import uuid
import subprocess
from jetbot import Robot
from IPython.display import display
import sys
import math
import cv2 as cv
import numpy as np
import io
from PIL import Image
from uuid import uuid1 #to generate file names
import os
import time
from PIL import Image as im
#Create the robot instance as well as the Camera instance.
#Make sure that the robot is not moving
#in order to avoid accidents once the code starts running
robot = Robot()
camera = Camera.instance(width=300, height=300)
#initializes the camera instance
robot.stop()
#Define functions in order to process images. <br>
#‘convert_image_to_numpy‘ is a function made to transform the images into numpy
#array for preprocessing. <br>
#‘save_snapshot‘ saves images into a pre-set directory.
def convert_image_to_numpy(image):
    imageBinaryBytes = image.value
    imageStream = io.BytesIO(imageBinaryBytes)
    imageFile = Image.open(imageStream)
    I = np.asarray(imageFile)
    return I

from uuid import uuid1 #to generate file names

def save_snapshot(directory):
    image_path = os.path.join(directory, str(uuid1()) + '.jpg')

    with open(image_path, 'wb') as f:
        f.write(image.value)
#Use the python notebook widgets library in order to
#be able to stream what the camera is capturing.
```

```
image = ipywidgets.Image(format='jpeg', width=300, height=300)
# this width and height doesn't necessarily have to match the camera
camera_link = traitlets.dlink((camera, 'value'),(image, 'value'), transform=bgr8_to_jpeg)
display(image)
#Looping Code:
#1. Identify the black and green pixels in
#the image (rgb was determined through trial and error and using #paint)
#2. Pre process the image by removing the noise
#    - use erode to remove the small white noises
#        - https://tinyurl.com/s45zmj5a
#    - use dilate after doing erosion to return the image to
#    #the original size. It helps to join the the broken parts of an object
#        - https://appdividend.com/2022/03/15/python-cv2-dilate/
#    - if no pixels fall under our black rgb set
#range for a predetermined number of frames we stop and finish the process
#3. For the sake of debugging and understanding our measurements
#we draw contours onto the image being displayed on the notebook
#4. Calculate the angle and center of the boxes
#5. Send the "error" distance from center and angle to the PD controller
ang=0
image = ipywidgets.Image(format='jpeg', width=300, height=300)
# this width and height doesn't necessarily have to match the camera
camera_link = traitlets.dlink((camera, 'value'), (image, 'value'), transform=bgr8_to_jpeg)

count=0
while(True):

    # 1 isolate blackline
    img_arr = convert_image_to_numpy(image)
    roi = img_arr[200:300, 0:300]
    blackline= cv.inRange(roi,(0,0,0),(68,68,68)) # use rgb values to identify the black lines
    greenbox = cv.inRange(roi, (0,75,0), (90,150,90)) # use rgb values to identify the green squares

    # 2 remove noise by convoluting
    kernel = np.ones((3,3), np.uint8)
    # moves a kernel through the image as in 2D convolution to remove white noises
    blackline = cv.erode(blackline, kernel, iterations=5)
    blackline = cv.dilate(blackline, kernel, iterations=9)
    greenbox = cv.erode(greenbox, kernel, iterations=5)
    greenbox = cv.dilate(greenbox, kernel, iterations=9)

    # 3 find contours
    contours_blk, hierarchy_blk = cv.findContours(blackline.copy(), cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
    contours_grn, hierarchy_grn = cv.findContours(greenbox.copy(), cv.RETR_TREE, cv.CHAIN_APPROX_SIMPLE)
    # draw contours, bounding box, and center line
    greend= False
    # centb=0
    x_last = 300
    y_last = 150

    # no black was detected
    if len(contours_blk) == 0 :
        count=count+1
        # if the black line is not detected for count number of frames then we stop moving and finish the program
        if count == 5:
            newimage = im.fromarray(roi)
            display(newimage)
            print("error")
            robot.stop()
            break

    # black is detected
    if len(contours_blk) > 0 :

        # only one black shape was detected
```

```
    if len(contours_blk)==1:
        blackbox=cv.minAreaRect(contours_blk[0])

    else:
        cands=[] # list of candidates
        off_bottom=0

        # Analyze every black object detected
        for num in range(len(contours_blk)):
            # determine the best fitted box around the object
            blackbox = cv.minAreaRect(contours_blk[num])
            (x_min, y_min), (w_min, h_min), ang = blackbox
            box = cv.boxPoints(blackbox)
            (x_box,y_box) = box[0] # gets the lowest corner of the box

            # check if it falls in roi (roi = bottom of image)
            if y_box > 298 :
                off_bottom += 1

            # append the values of the box, the idx of the contour, and the x_min, y_min to candidates
            cands.append((y_box,num,x_min,y_min))


        # sort the candidates according to the y_box bottom to top
        cands=sorted(cands)
        if off_bottom>1:
            cands_ob=[]
            for num in range((len(contours_blk)-off_bottom),len(contours_blk)):
                (y_highest,con_highest,x_min, y_min) = cands[num]
                total_distance = (abs(x_min - x_last)**2 + abs(y_min - y_last)**2)**0.5
                cands_ob.append((total_distance,con_highest))
                cands_ob = sorted(cands_ob)
                (total_distance,con_highest) = cands_ob[0]
                blackbox = cv.minAreaRect(contours_blk[con_highest])
        else:
            (y_highest,con_highest,x_min, y_min) = cands[len(contours_blk)-1]
            blackbox = cv.minAreaRect(contours_blk[con_highest])
    (x_min, y_min), (w_min, h_min), ang = blackbox
    x_blk,y_blk,w_blk,h_blk = cv.boundingRect(contours_blk[0])
    centb=x_min+(w_min/2)
    x_last = x_min
    y_last = y_min
    if ang < -45 :
        ang = 90 + ang
    if w_min < h_min and ang > 0:
        ang = (90-ang)*-1
    if w_min > h_min and ang < 0:
        ang = 90 + ang

if len(contours_grn) > 0 :
    x,y,w,h = cv.boundingRect(contours_grn[0])
    centg=x+(w/2)
    cv.rectangle(roi,(x,y),(x+w,y+h),(0,0,255),3)
    cv.line(roi, (int(x+(w/2)), 200), (int(x+(w/2)), 250),(255,0,0),3)
    if y_last> y:
        greend= True

if greend:
    if centg-centb > 5 :
        cv.putText(roi, "Turn Right", (50,50), cv.FONT_HERSHEY_SIMPLEX, 1, (0,255,0),3)
        robot.right(0.3)
        time.sleep(0.1)
        robot.stop()
        #robot.forward()
    else :
```

```
        if centg-centb > 0 :
            cv.putText(roi, "U-Turn", (50,50), cv.FONT_HERSHEY_SIMPLEX, 1, (0,255,0),3)
            robot.right(0.3)
            time.sleep(1)
            robot.stop()
            #robot.forward()
        else:
            cv.putText(roi, "Turn left", (50,50), cv.FONT_HERSHEY_SIMPLEX, 1, (0,255,0),3)
            robot.left(0.3)
            time.sleep(0.1)
            robot.stop()
            robot.forward()
    box = cv.boxPoints(blackbox)
    box = np.int0(box)
    cv.drawContours(roi,[box],0,(0,0,255),3)
    cv.putText(roi,str(ang),(10, 40), cv.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
    cv.putText(roi,str(error),(50, 50), cv.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
    cv.line(roi, (int(x_min),0 ), (int(x_min),100 ), (255,0,0),3)

else:
    setpoint = 150
    error = int(x_last - setpoint)
    ang = int(ang)
    box = cv.boxPoints(blackbox)
    box = np.int0(box)
    cv.drawContours(roi,[box],0,(0,0,255),3)
    cv.putText(roi,str(ang),(10, 40), cv.FONT_HERSHEY_SIMPLEX, 1, (0, 0, 255), 2)
    cv.putText(roi,str(error),(50, 50), cv.FONT_HERSHEY_SIMPLEX, 1, (255, 0, 0), 2)
    cv.line(roi, (int(x_min),0 ), (int(x_min),100 ), (255,0,0),3)



    speed=0.24
    kp=0.1
    ap=0.3
    steer= error*kp + ang * ap
    if error>100:
        steer=steer*0.7
    # if line is to the left of jetbot, turn left
    if steer>0:
        steer=10-steer
        robot.set_motors(speed,speed*steer/10)
    # if line is to the right of jetbot, turn right
    else:
        steer=steer*-1
        steer=10-steer
        robot.set_motors(speed*steer/10,speed)

newimage = im.fromarray(roi)
display(newimage)
```