Mechatronic design project 2

Rob Borrett

November 2021

Contents

1 Introduction

This report details the design of an image processing system for detecting and directing a Martian rolling robot as it navigates a maze of obstacles, and safely crosses an alien highway. Communication and path planning execution was also developed, however it is not detailed in this report.

While the location and orientation detection software was effective, it was less reliable than using ArUco codes. The path planning and obstacle avoidance algorithms proved effective, however using a node based approach rather than pixels to determine the best path may prove more efficient in the future. The road crossing algorithm is effective and reliable. Going forward the program could be made more versatile to enable the robot to cross the road at every opportunity rather than just some.

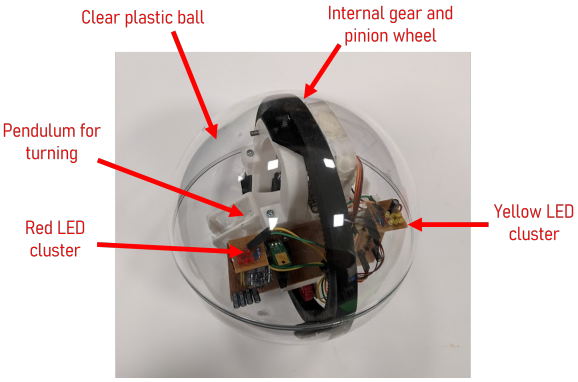
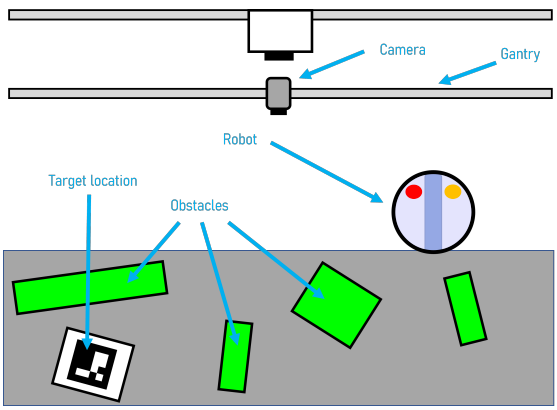


Figure 1: The robot

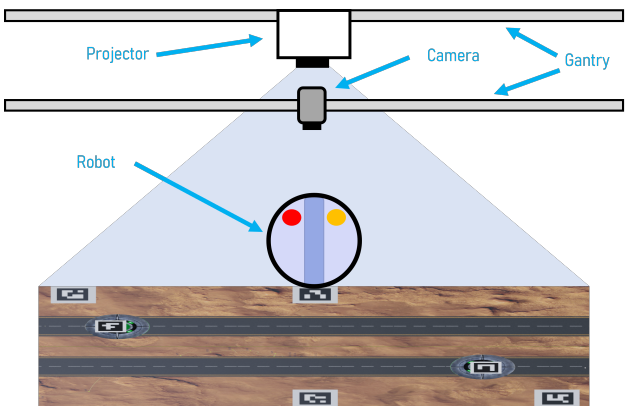
1.1 Background information

The robot itself is contained within a spherical plastic shell 1. In its current development state, the robot has to turn and move separately. The turning procedure involves rolling the robot onto its side. turning the inner body a specified number of degrees, and then rocking the robot back up to its vertical position. This process rotates the ball by the specified amount, but in doing so causes a translation.

Figure 2 shows the setup of the robot's test environment. The obstacles that the robot must navigate through are a series of green blocks.The highway is projected from a projector on an overhead gantry and displays moving ArUco codes on top of spaceships. The robot communicates wirelessly with mission control via User Datagram Protocol (UDP) and Transmission Control Protocol (TCP) data links. The equipment available to mission control is a geostationary satellite (camera fixed to gantry) and a super computer running advanced software (Laptop running Python). My main aims of the project were to gain develop my understanding of computer vision, and system, and apply my existing computing skills to python.The aims of the team were to develop something novel and exciting as it creates more learning opportunities.



(a) Navigation through obstacles



(b) Crossing martian highway

Figure 2: Different setups

2 Detecting the Robot's position and orientation

Appendix B contains the script used to detect the robot and plan a path through the obstacles in the environment.

2.1 Calibrating coordinate system

Throughout this project the following assumptions were made. The camera looks directly downwards, perpendicular to the floor, and there was no distortion caused by the lens. This meant that the camera pixel coordinates could be translated into coordinates on the ground by scaling them in a linear fashion, and pixel distances could be scaled into real life in a similar manner. This assumption was tested and it proved that it was reliable, giving results accurate to within 3cm of the calculated length. The conversion constants were created through calibration ArUco codes placed in the test environment at a specified distance. Knowing the ArUco spacing, in real life and their pixel spacing, a conversion factor can be calculated as shown in the code below:

```
# Find the distance between ArUco codes 0 and 1
Pixels_dist = math.sqrt((ArUco_0[:][0]-ArUco_1[:][0])**2 + (ArUco_0[:][1]-ArUco_1[:][1])**2)

# Work out the scaling factor to go from pixels to real life and vice versa
pixels2real = Real_dist/Pixels_dist
real2pixels = Pixels_dist/Real_dist
```

2.2 Finding the Robot's location

The mechanical design of the robot posed challenges for detecting the robots location. ArUco codes within the ball could not easily be detected due to the internal gear ring and glare from lighting. This problem was overcome by using image recognition to detect man made features on the surface of the robot. Detecting circles was a running theme throughout this project.This was initially done using OpenCv's built-in circle detection function called HoughCircles [1], however the ambiguous input variables made it hard to use.

The OpenCV function cv2.findContours() [2]function was used. While it does not directly identify circles, it can be used to identify contour shapes within an image. Combining this with the condition that the detect contour radius must be within a buffer of the known ball radius, made for reliable detection of the desired circular contours. Early concepts involved detecting the ball by painting it orange, and applying a HSV colour mask to the image to extracting only the orange pixels. From there the ball detection code can identify the circular contours created. This was effective and consistent as the colour mask could be easily tuned.

2.3 Finding the robot's orientation

Finding the robot's orientation is also crucial information for planning where it needs to go. Design iterations were carried out as the project evolved as shown in figure 3. Iterations 1 and 2 are included included in Appendix A.

2.3.1 Direction detection using LEDs

During the latter stages of the project, it was discovered that the paint had an impact on the mechanical performance of the robot. Using threshold filtering a cluster of red and yellow LEDs within the ball can be located, see figure 4. The pixel location of each cluster can be used to work out the orientation of the ball with basic trigonometry as shown below:

$$\theta = \arctan\left(\frac{y_{red} - y_{yellow}}{x_{red} - x_{yellow}}\right) \tag{1}$$

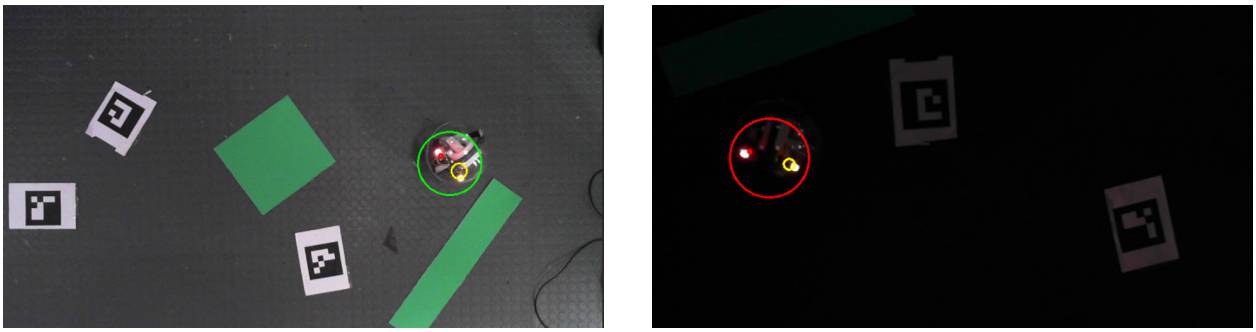
Where x_{red}, y_{red} and x_{yellow}, y_{yellow} are the calculated coordinates of the respective red and yellow LED clusters, giving θ the baring, defined as positive clockwise from the y axis. This is reflected in the code as follows.

(a) Iteration 1: Arrows

(b) Iteration 2: Black rectangles with coloured dots

(c) Iteration 3: LED clusters

Figure 3: Orientation detection iterations



(a) High contrast in dark

(b) Low contrast in dark

Figure 4: Finding orientation from LEDs

```
# calculate the robot's baring, defined as positive clockwise from the y axis
baring = math.atan2(y_red-y_yellow,x_red-x_yellow) # Atan2() to give sign of calculated angle
```

This function is show in appendix A defined as Robot_Baring_Location(). The LEDs were much harder to detect than dots of flat colour. The centres of the LEDs tended to be so bright that they were seen as white light by the camera. Furthermore, as the LEDs were housed within the ball of the robot, there was an element of scattering and distortion as shown in figure 5.

This meant sometimes the colour masks produced little to no valid contours, and sometimes the detected contours were unreasonably far from the true LED cluster location. For this reason a while, try and check structure was adopted. A while loop was initiated that was only exited once a reasonable approximation was detected within the while loop. The try structure means any errors associated with being unable to detect valid contours are ignored. Once found,the solution is compared to The real distance between the two LED clusters, to ensure the estimate is reasonable:

```
    while 1:
        # the try is needed because the image detection sometimes fails
        try:
            #print('\n Ball Coordinates:',x1,',',y1,'\n')
            # get baring and location from LED positions using ball_location function
            x_red,y_red,r_red,frame = ball_location(low_red,high_red,ball_rad_pixels/4)
            x_yellow,y_yellow,r_yellow,frame = ...
                ball_location(low_yellow,high_yellow,ball_rad_pixels/4)

            # filtering out false results by checking the distance between the red and ...
                yellow LEDs
            dist_xy =math.sqrt((x_red-x_yellow)**2+(y_red-y_yellow)**2)

```

(a) Red LED mask (manually cropped). Note the hollow centre due to the brightness of LEDs

(b) Yellow LED mask (manually cropped). Notice the scattering of the light caused by the reflections and refractions of the ball

Figure 5: Challenges of detecting LED clusters

```
12              # check that the soloution found is reasonable
13              if dist_xy< 2*ball_rad_pixels:
14
15                  # calculate the robot's baring defined as +ve clockwise from Yaxis
16                  baring = math.atan2(y_red-y_yellow,x_red-x_yellow)
17
18                  # calculate robot centre
19                  x1 = int((x_red + x_yellow)/2)
20                  y1 = int((y_yellow + y_red)/2)
21
22                  # If a baring is found, then break the loop
23                  if baring is not None:
24
25                      print('\n Baring:', baring*180/math.pi)
26                      break
27          except:
28              print('unable to find suitable image solution, trying again')
```

3 Route finding

The following section discusses the programs created to find a path through a series of obstacles as described in figure 2a.

3.1 Obstacle detection

In order to navigate the robot through a series of obstacles, a green HSV filter was applied to isolate the obstacles from the environment:

```
1  # Defining HSV threshold values
2  low_green = np.array([40, 80, 80])
3  high_green = np.array([90, 255, 255])
4
5  # capture a frame and convert to HSV
6  ret, frame = cap.read()
7  hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
8
9
10 # Apply green mask
11 Green_mask = cv2.inRange(hsv_frame, low_green, high_green)
```

```
12  #cv2.imshow('maassk',Green_mask)
```

These HSV threshold values were obtained from [3], a script which provides a real time view of the filters in action on the images that are being processed, allowing you to change the threshold values using sliders.

This was then pixelated to make processing much faster and turned into a binary map, for the path planning algorithm to use. A buffer was added to around the obstacles to account for the robot's size and the fact that turning causes a translation.

(a) Original frame

(b) Green mask applied to frame to create binary image, isolating obstacles

(c) Mask pixelated and path established through obstacles

Figure 6: Process obtaining the obstacle map from a frame

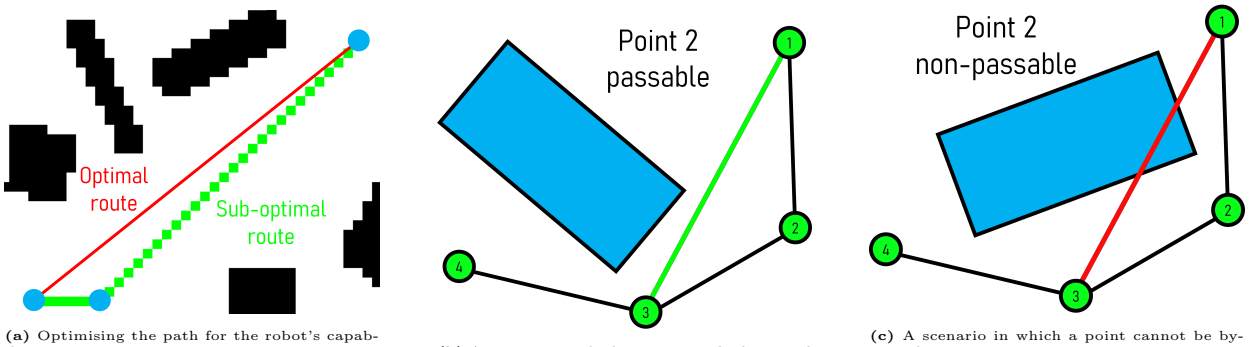
3.2 A* search algorithm

To navigate the obstacles, it was decided to use the A* search algorithm, an extension of Dijkstra's algorithm. This decision was made because A* is simple to implement, commonly used in real-time gaming scenarios [4] and is an 'optimal' method. [5] Dijkstra's algorithm tells you the shortest distance from one node to every other node in a network, and it is guaranteed to do so, provided a solution exists. A* differs from Dijkstras by the cost function:

$$F = G + H \tag{2}$$

Where G is the distance between start point and the current node being and H is the estimated distance from the current node to the end node. This means that A* prioritises searching routes that have the best chance of reaching the end point. [5] The code used for this algorithm was based off code from [6] (see the function astar() in appendix B). The code works by defining two lists, an open list and a closed list. The open is a list of unexplored nodes which could still be explored. The closed list is a list of explored nodes. The path through the nodes is defined by assigning the lowest cost node prior to the current node as the current node's parent. This essentially means once you arrive at the target node, you can follow the path of parent nodes back and you will arrive at the starting node, having taken the optimal route. The code works as follows [4]:

1. Initially the closed list should be empty, and the starting node is the only node in the open list.
2. The open list is searched for the lowest cost square f=g+h, and this square is made the current node.
3. The current node is added to the closed list.
4. The adjacent nodes to the current square are analysed and their cost functions are calculated. If a node is already on the closed list, or it is defined as impassable ignore it, otherwise add it to the open list if it is not already there.
5. Make the current node the parent to its connected nodes, however if a node is already on the open list, use the G cost to work out if the path to that node is more efficient than the path currently defined by its existing parent node.



(a) Optimising the path for the robot's capabilities

(b) A scenario in which a point can be bypassed

(c) A scenario in which a point cannot be bypassed

Figure 7: Simplifying the path

6. Stop only if you reach the destination square, or the open list is empty, in which case there are no solutions.

The code obtained from [6] contained bugs. It did not check to see if the children nodes were on the closed list which meant if the current node was adjacent to a lower cost node on the closed list, it would select this node as the next current node, and end up repeatedly re-exploring the same closed paths. This was fixed by adding the statement:

```
1             # Make sure the current node isn't already in the closed list
2             if Node(current_node, node_position) in closed_list:
3                 continue
```

3.3 Further simplifying the shortest path

The path produced by the function astar()(appendix B) is a series of (x,y) pixel locations, as shown green in figure 8 This needs to be translated into a functional form for the robot's communication and movement capabilities. The function simplify_path() B was created to do this. It starts by removing any points that are along the same line, which leaves a series of (x,y) pixel locations that mark the corner points that rotations need to occur at (figure 8). Travelling through a grid means you can only travel vertically horizontally or diagonally. Reducing the number of turning points improves the efficiency of the robot's movement. This means more optimal paths could be taken if these non diagonal, horizontal or vertical paths are taken as illustrated in figure 7a. The second half of simplify_path() does this using the shapely.geometry libraries Polygon, LineString and Point. This section of code turns the binary map of the environment into a series of polygons and then assesses whether bypassing a point on the current path will cause the line between the two adjacent points to intercept any of the polygons as shown in figures 7b and 7c. If there is no interception, the point can be bypassed:

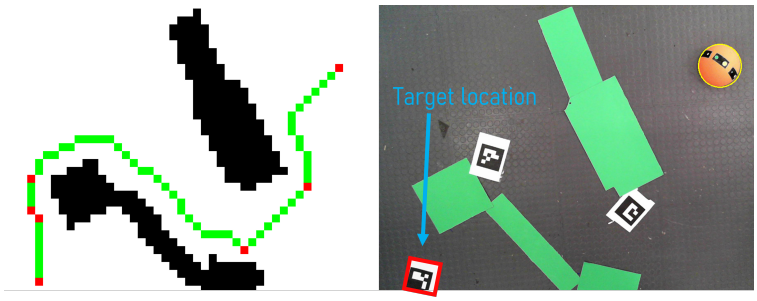


Figure 8: An example path produced through the environment. The red dots show the attempts at simplifying the path, with mixed success

```
1
2         # create a connecting line between current point and the evaluated point
3         path_line = LineString([Point(path_new[n]),Point(path_new[n+2])]) # create polygoy ...
              line which skips a point
4
5         # initally set intersection to 0
```

```
        intersection = 0

        # evaluate whether skipping this point causes you to hit the maze walls
        # loop through each  maze contour
        for j in range(0, len(contours)):

            # create a polygon from the given contour
            contour = np.squeeze(contours[j])
            maze_poly = Polygon(contour)

            # check whether the line created above intersects with current polygon
            if maze_poly.intersects(path_line):

                # if there is an inersection make intersection 1
                intersection = 1

        # if the proposed path does not intersect the maze polygons,
        # make this the new path by deleting any points between there and point n
        if intersection == 0:

            del path_new[n+1]

        else : # otherwise increment n to move onto the next point
            n = n+1
```

This simplification is still requires further development. It currently works for simple paths, however has issues with removing the correct nodes for longer, more complicated paths as illustrated by figure 8.

Going forward, it may be fruitful to investigate implementing the path planning using nodes rather than pixels.

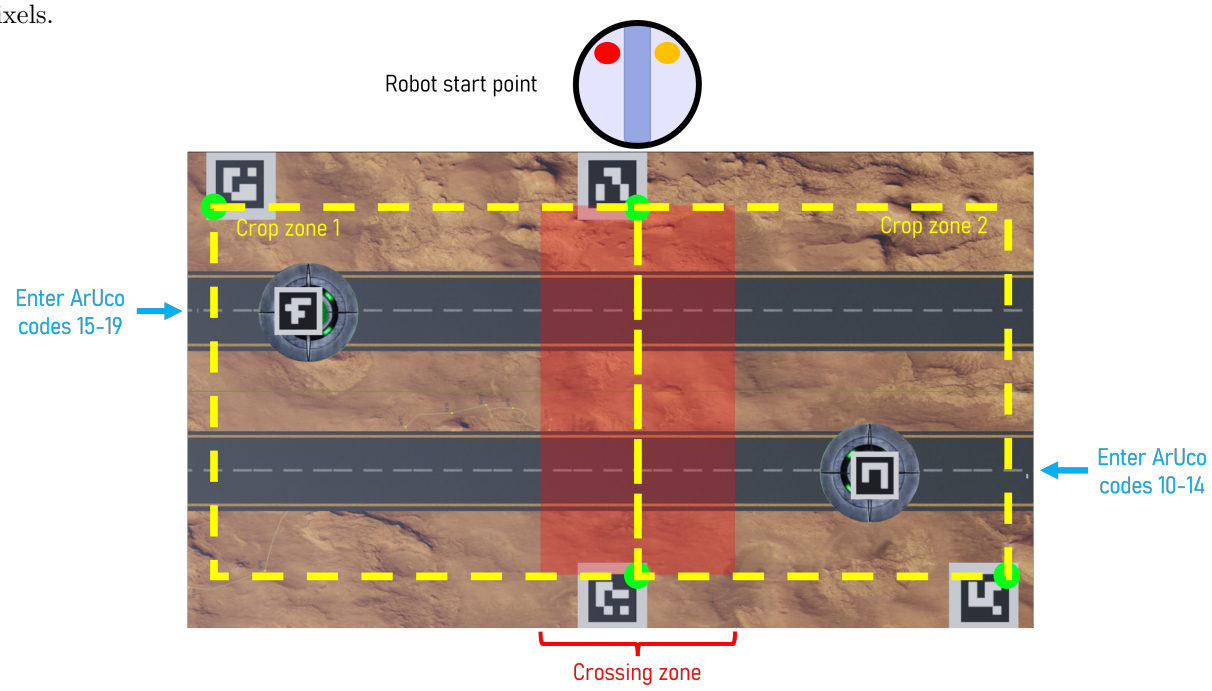


Figure 9: The Martian highway environment, showing the different crop zones

4 Road crossing

Figure 9 breaks down the projected model used to simulated the martian highway. The model was created in Blender and randomly spawns spaceships entering from the left and right with ArUco codes on top. Spaceships entering from the right and left can have any value ArUco code between 10 and 14, and 15 and 19 respectively. The python script in Appendix C was created to detect when is safe for the robot to cross

the road. The projected graphic and Python program that processes the camera image feed are completely independent.

Given an input of how wide the robot is and how long it takes it to cross the road, the algorithm identifies the spaceships (ArUco codes) on the road and their speeds, and calculates at what time they will enter and leave the crossing zone. The flow diagram explaining the overall architecture of the algorithm is shown in figure 10. The speeds of the spaceships are calculated by working looking at the location of an ArUco code at a given point in time, waiting for half a second, and then finding the pixel location of the ArUco code. The distance travelled is divided by the time between the two frames, giving the speed. This speed is assumed to be constant, and is then used to determine the time that it will take the spaceship to travel the distance to the crossing zone. The times for when the ArUco spaceships will enter the crossing zones is stored in an array called Entry_Exit, which is remembered throughout the loops. This means if for whatever reason the ArUco spaceships stop being detected, the previously calculated values will be remembered and still used. Once the entry and exit times to the crossing zone are harvested, crossing times are suggested at a given time interval, and each element in Exit_Entry is examined to see if this time creates a viable solution. The suitability of a solution is determined with the following statements:

```
1               # loop through the next 10 s to find a time when it is suitable for the robot ...
                    to cross
2               for i in range (0,int(10/t_step)):
3
4                   # loop through each time step
5                   ball_entry_time = time.time() + 4 + i*t_step # + 4 included to give ...
                        checking time
6                   ball_exit = ball_entry_time + Ball_cross_time
7                   #print('\n Ball Entry time:',ball_entry_time )
8
9                   # loop through each exit and entry time
10                  total_feasibiity = 0
11                  for j in range(0, len(Entry_Exit)):
12                      feasibility = 0
13                      # make feasibilty 1 if this time instance is feasible for ecach car
14                      # if car has fully crossed by the time the ball enters the crossing zone
15                      if ball_entry_time≥ Entry_Exit[j][1]:
16                          feasibility = 1 # this time is feasible
17                      # if the ball leaves before the car enters the crossing zone
18                      if ball_exit≤ Entry_Exit[j][0]:
19                          feasibility = 1 # this time is feasible
20
21                      # create a running total
22                      total_feasibiity  = total_feasibiity + feasibility
23
24                  #print('\n Total Feasiblity:', total_feasibiity)
25
26                  # if all of the cars are feasible break the loop and use that time
27                  if total_feasibiity == len(Entry_Exit):
28                      #print('\n POTENTIAL SOLUTION FOUND AT TIME =',ball_entry_time)
29                      break
30
31                  else:
32                      ball_entry_time = None
```

This is designed to ensure the robot has fully exited the crossing zone before the ArUco spaceship enters the crossing zone, or the ArUco spaceship has fully exited the crossing zone before the robot enters it. Once a valid solution is found, the checking process begins. The code enters a loop which is exited once the crossing time is reached. While the code waits it is continually checking to see if any new ArUco Cars have entered the road that are are a danger to the robot. Notice the time delay of 4 seconds on line 5 of the above code sample. This is introduced to ensure at least 4 iterations of checking are carried out before the car does cross the road. It was found during testing that if crossing time solutions were found at times in the very near future, no checks would be carried out.

From figure 9 The ArUco codes 3 to 6 are used to mark out the different zones in the map. Note how the green dots, which mark the coordinate of the code that is read, are positioned. ArUco codes 3 and 4 mark the start and end points of the robot's crossing path respectively. Analysing the spaceships once they have travelled through the crossing zone is useless as they are no longer a threat to the robot. Cropping the images to look at the left and right half of the image separately enables the code to filter out any useless information, by only searching for the useful ArUco codes in each cropped zone. For example, in the left zone this means only searching for ArUco codes 14-19. It also means all the important ArUco codes are travelling in the same direction.

Admittedly, this hurriedly created program has many processes that could be simplified. For example, the process of cropping the image into two halves is not necessary if velocities were used instead of speeds. What seemed like a god solution to dealing with spaceships coming from opposite directions are the time actually complicates whole process. The directional information could be used to change the calculations for the entry and exit times to the crossing zones.

Currently the code overwrites any old and new position data with each loop if a corresponding ArUco code is present, always calculating the velocity with the same time interval 0.5 seconds. A smaller time interval magnifies errors in the velocity calculations, in comparison to a larger one. Therefore, the code could be improved by storing the ArUco car's first entry time and position, and instead of overwriting it each loop, only updating the new position. It means the velocity and hence the predicted entry and exit times will become more accurate as the ArUco Spaceship nears crossing zone. It also means smaller time steps, could then be used, meaning more checking, and more accuracy.

Despite it's quirks this is an effective program that delivers on what it is supposed to do as shown in figure 11, illustrating the code correctly identifying, waiting and commanding the robot to cross at the right time.

More research should have been done initially into how best to approach this task. It is only in writing this report that existing literature on road crossing algorithms and AI was discovered. For example [5] contains a wealth of knowledge around the subject of Model Based Reflex Agent algorithms which are applicable to this situation. Had the time been taken at the start to research this, the outcome may be very different.

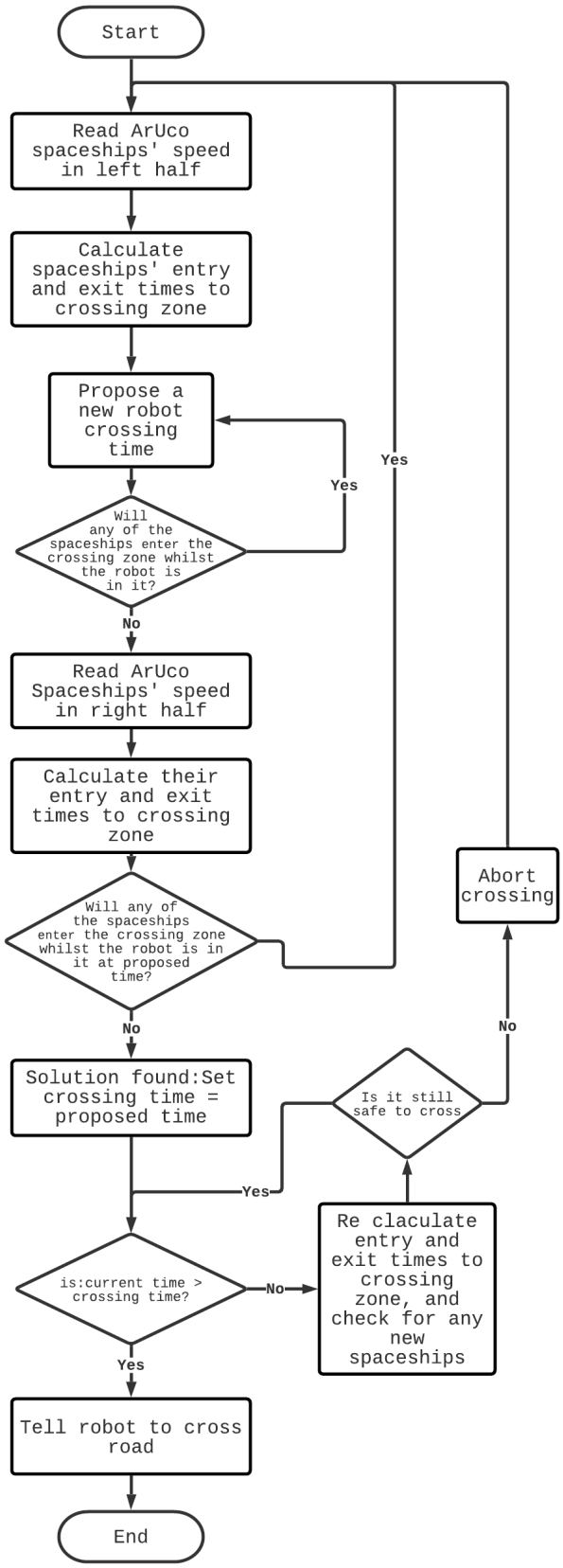
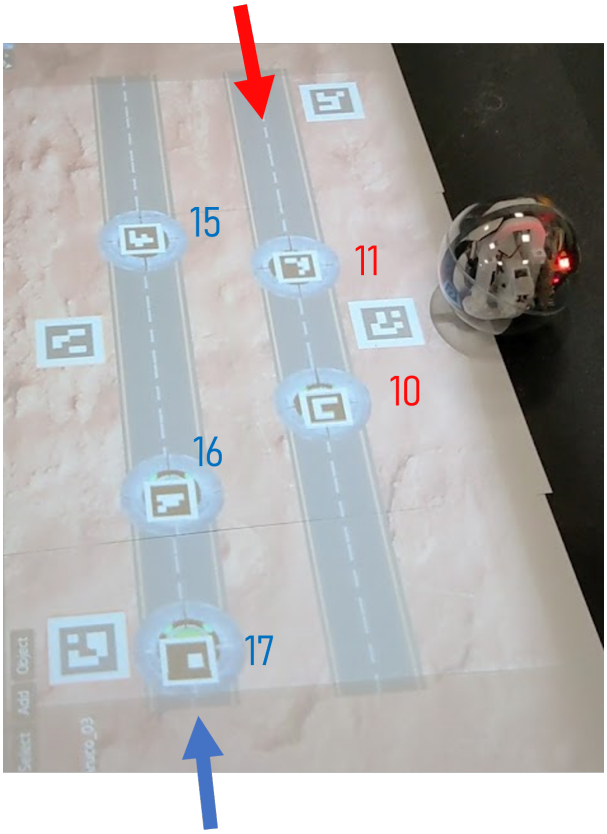
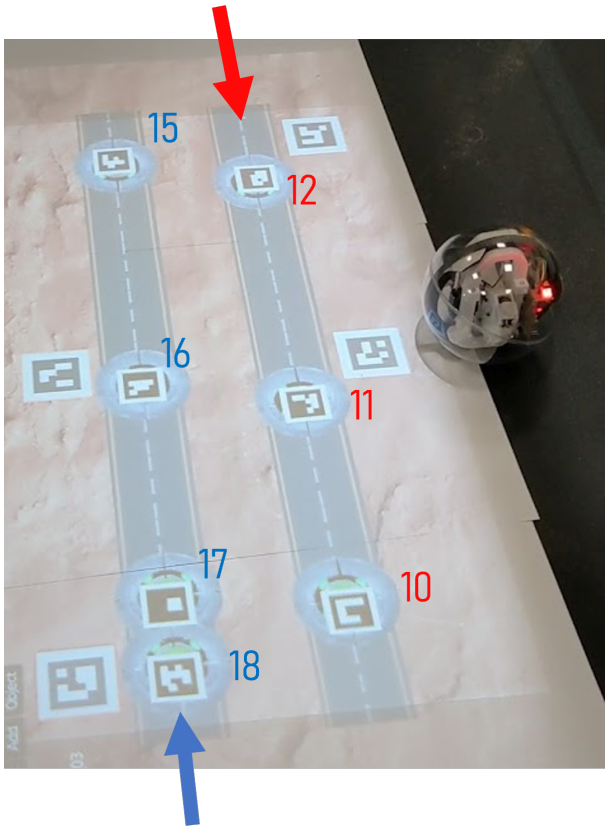


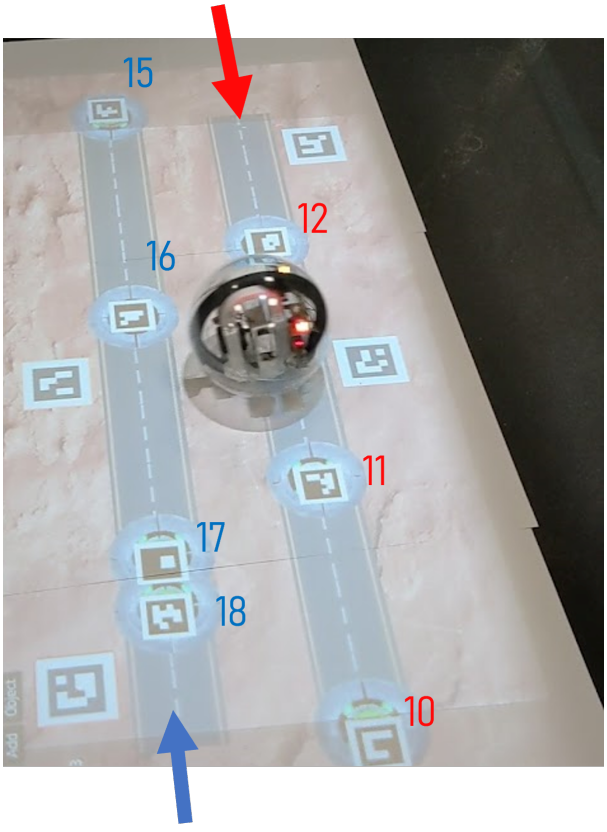
Figure 10: A flow diagram for the road crossing algorithm



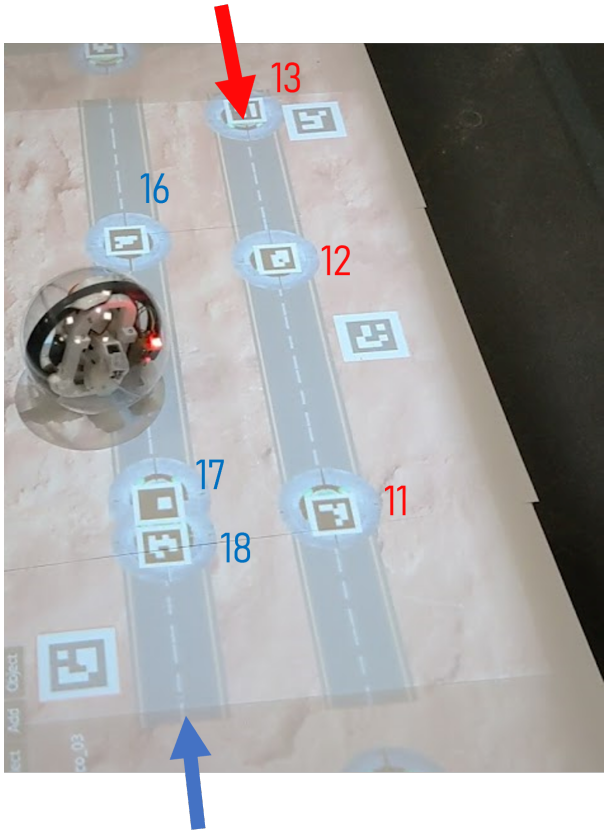
(a) Solution spotted after ArUco 16 leaves crossing zone, Program is checking for any new spaceships to enter the road



(b) ArUcos 12 and 18 enters, however their speed is too slow to cause an abort, ArUco 16 enters crossing zone



(c) ArUco 16 leaves the crossing zone , making it safe to cross



(d) Ball safely crosses road

Figure 11: Road crossing example

5 Reflections

Reflecting upon the aims of this project, I feel as though they were largely met. Image processing was completely new to me prior to this project and I now feel confident in how it works and how to approach images processing challenges. Having an understanding on the way that image processing works will useful to me in the future regardless of whether I am implementing it in a system myself. The understanding it has given me will allow me to work better in a team where there is an image processing aspect, allowing me to design to integrate effectively with such a system.

I have also learned the importance of taking the time to explore all solutions thoroughly before jumping into creating a program. Software is very forgiving in the sense that you commit no cost except time in developing something, making it easy to redo something;as I learned from the three methods of detecting the ball orientation. I think had I spent an extra hour fully exploring and planning each program I would have realised the flaws in the arrow design before creating it. However it was the 'fail fast learn quickly' approach that allowed me to successfully churn through all three iterations, whilst developing the other software.

I think the team aims were certainly met. Whilst the robot produced was not the most practical, it stood out for its uniqueness. It did not require ArUco codes to detect its location, it had a unique locomotion system. The road crossing critical event was also unique, and the team on the whole did well given the time frame to produce such work.

6 Conclusion

In conclusion, the project was overall a success. An effective method for deriving the robot's location and orientation was created, relying on a red and a yellow LED cluster within the ball. The A* searching algorithm was implemented to navigate obstacles in the test environment, and the proposed paths were simplified using geometric relations with the environment. However further development is needed to produce reliable results. The road crossing algorithms worked effectively allowing the robot to safely cross the road. The creation of this algorithm was rushed, however and research into a more logical workflow could be done in the future.

References

[1] OpenCV, *Feature Detection*. OpenCV, 2021. [Online] Available from: `https://docs.opencv.org/4.5.3/dd/d1a/group__imgproc__feature.html#ga47849c3be0d0406ad3ca45db65a25d2d` [Accessed: 04-11-21].

[2] OpenCV, *Structural Analysis and Shape Descriptors*. OpenCV, 2015. [Online] Available from: `https://docs.opencv.org/4.5.3/d3/dc0/group__imgproc__shape.html#gadf1ad6a0b82947fa1fe3c3d497f260e0` [Accessed: 04-11-21].

[3] Praveen, *How to find HSV range of an Object for Computer Vision applications?* Medium.com, 2020. [Online] Available from: `https://medium.com/programming-fever/how-to-find-hsv-range-of-an-object-for-computer-vision-applications-254a8eb039fc` [Accessed: 04-11-21].

[4] G. Seemann and D. M. Bourg, *AI for Game Developers*. O'Reilly Media, Inc, 2004.

[5] R. Stuard and P. Norvig, *Artificial Intelligence: a Modern Approach*. London: Pearson Education, 2016.

[6] N. Swift, *Easy A(star) Pathfinding*. Medium.com, 2017. [Online] Available from: `https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2` [Accessed: 04-11-21].

A Alternative image detection methods

A.0.1 Arrow direction detection

Iteration 1 proved to be inconsistent and inaccurate as soon as it was implemented onto a spherical ball. It relied on detecting an arrow on the ball and approximating this to a polygon using the python function cv2.approxPolyDP(). The polygon created is an array of coordinates, however, for it to be useful, the arrow vertex that each array cell corresponds to must be discovered. This is done by examining the angles formed at each vertex. For three points ABC (see fig 12), the subtended angle can be described as as follows:

$$\theta = \arccos\left(\frac{A \cdot B}{|A||B|}\right) \tag{3}$$

By analysing the angles at each vertex, the base vertices of the arrow can be oriented by finding two consecutive right angles. lets us say the first right angle is at vertex A, the second right angle is at point B as shown in figure12. Having mapped the array points to the arrow vertices, the direction that the arrow points in can be deduced by finding the direction of the vector $\overrightarrow{BC}$ where C corresponds to the point directly after B.

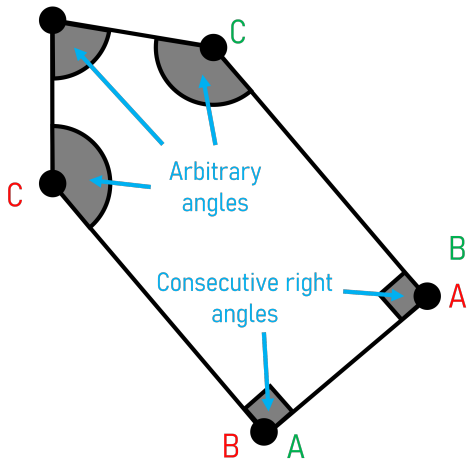


Figure 12: Calculating the baring of an arrow by finding consecutive right angles

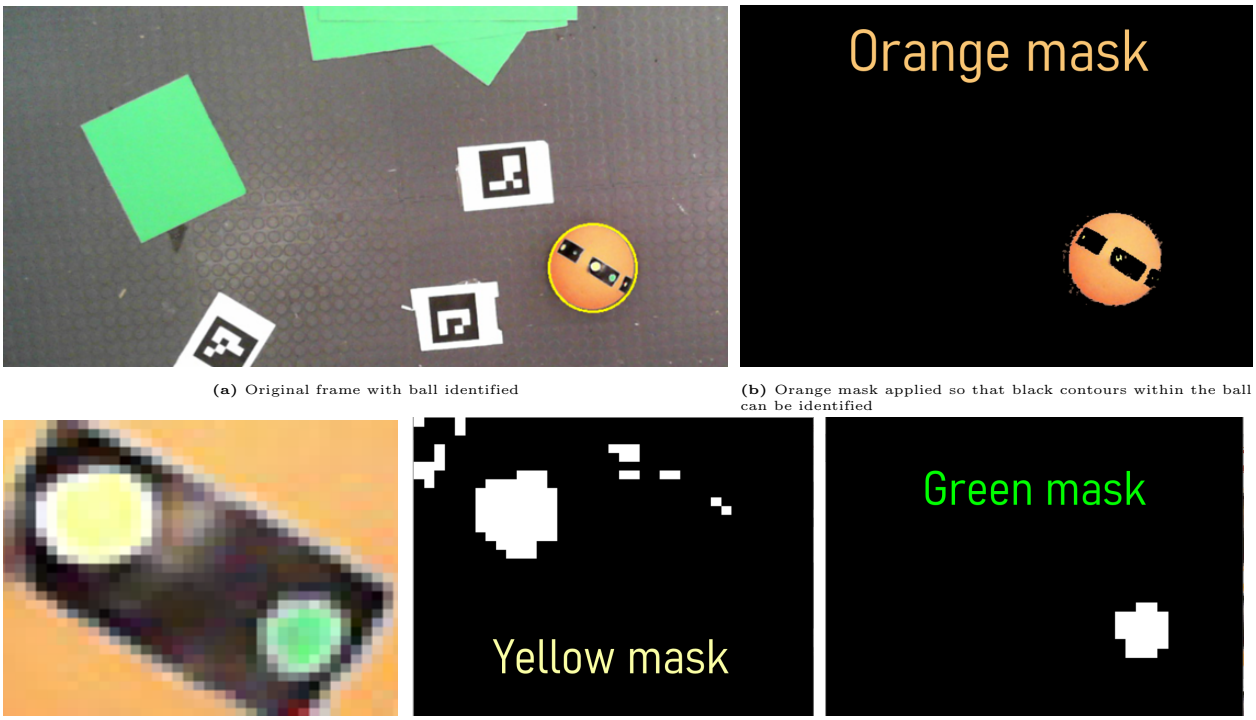
The initial testing was done on a flat piece of paper meaning there was no distortion in the shape of the arrow due to parallax.

A.0.2 Detecting direction using dots

Iteration 2 was more reliable. The direction of the robot was indicated by the green and yellow dots. The black rectangles were required to indicate the grouping of each pair of dots. The process was as follows:

1. detecting the ball and crop the image to the ball
2. identify the largest black rectangle and crop the image to smallest bounding rectangle around it
3. Identify the coords of the dots centres by applying a HSV threshold filters
4. Use trig to calculate the robot's orientation

The colour masks applied to the cropped image consistently allowed the detection of the coloured dots, however again parallax meant the calculated orientation of the robot was inaccurate.



(a) Original frame with ball identified

(b) Orange mask applied so that black contours within the ball can be identified

(c) Largest black contour identified and image cropped to smallest bounding rectangle around it

(d) Yellow mask applied to cropped image isolate yellow dot

(e) Green mask applied to cropped image to isolate green dot

Figure 13: Process of extracting the robot's baring from a frame

B Path planning and execution script

```
# Main executable program for imagae processing and communications between robot and computer
# This script is for the final version of the robot, using internal LEDs to detect its ... location and orientation
# It can find a path through green obstacles
# Written by R J Borrett, University of Bath, Nov-2021
# Credit to Nicholas Swift and Adrian Rosebrok as indicated in the code.


# Import necessary libraries
from mmap import mmap
import cv2                          # This is a library for compuer vision functions
import numpy as np                  # This is a library for mathematical functions for python
import time                         # This is a library to get access to time-related ... functionalities
import os                           # This is for finding the working directory
import cv2.aruco as aruco           # This library allows the detection of aruco codes
import math                         # This library contains lots of mathematical functions
import matplotlib.pyplot as plt     # This library is for plotting nice graphs
import socket                       # This library will allow you to communicate over the ... network
import time                         # This library will allow us to access the system ... clock for pause/sleep/delay actions
import logging                      # This library will offer us a different method to ... print information on the terminal (better for debugging purposes)
import imutils                      # This library will offer us basic image precessing ... functions
# You will need to pip install shapely if you don't already have it
from shapely.geometry import Polygon,LineString, Point  # This library will allow ... manuipulation of shapes


def ball_location(HSVball_Low, HSVball_High, ball_rad_pixels):
    # A fuction to detect a ball/ circular shapes from a catured camera frame
```

```
27      # Where:
28      #   HSVball_Low is the lower HSV threshold
29      #   HSVball_High is the upper HSV threshold
30      #   Ball_rad_pixels is the expected size of the object being detected.
31      #
32      # This function returns x,y,r and frame which are the x and y pixel coords of the ...
            centre of teh detected ball,
33      # r is the rad of the detected ball and frame is the image that the balls were found in
34      # based off code written by Adrian Rosebrock
35      # found at: https://www.pyimagesearch.com/2015/09/14/ball-tracking-with-opencv/
36
37      while 1:
38
39          cap.set(cv2.CAP_PROP_AUTO_EXPOSURE, 3)
40          cap.set(cv2.CAP_PROP_EXPOSURE, -2)
41          # Read camera
42          ret, frame = cap.read()
43
44          # convert to HSV
45          hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
46
47          # apply colour filter
48          ball_mask = cv2.inRange(hsv_frame, HSVball_Low, HSVball_High)
49          ball = cv2.bitwise_and(frame, frame, mask=ball_mask)
50
51          # show the image
52          cv2.imshow("ball", ball)
53          cv2.waitKey(20)
54
55          # specify buffer for detected ball radius to be within
56          min_rad = int(ball_rad_pixels - ball_rad_pixels*0.15)
57          max_rad = int(ball_rad_pixels + ball_rad_pixels*0.15)
58
59          # find contours in the mask and initialize the
60          cnts = cv2.findContours(ball_mask, cv2.RETR_EXTERNAL,cv2.CHAIN_APPROX_SIMPLE)
61          cnts = imutils.grab_contours(cnts)
62          center = None
63          output = frame
64
65          # only proceed if at least one contour was found
66          if len(cnts) > 0:
67              # find the largest contour in the mask, then use
68              # it to compute the minimum enclosing circle and
69              # centroid
70              c = max(cnts, key=cv2.contourArea)
71              ((x, y), r) = cv2.minEnclosingCircle(c)
72
73              x = int(x)
74              y = int(y)
75              r = int(r)
76
77              M = cv2.moments(c)
78              center = (int(M["m10"] / M["m00"]), int(M["m01"] / M["m00"]))
79
80              # only proceed if the radius meets a minimum size
81              if r in range(min_rad,max_rad):
82                  # draw the circle and centroid on the frame,
83                  # then update the list of tracked points
84                  output = cv2.circle(output, (x,y), r,(0, 255, 255), 2)
85                  cv2.imshow('output',output)
86                  cv2.waitKey(20)
87
88                  break
89
90      cap.set(cv2.CAP_PROP_EXPOSURE, -2)
91      return x,y,r,frame
92
93  def Robot_Baring_Location(low_red, high_red,low_yellow, high_yellow):
```

```
94      # A fuction to determine a xy coordinates and a directional baring of a ball robot, ...
            from the LEDs within the robot
95      # Given the following inputs:
96      #   low_red is the lower red HSV threshold for detecting red LEDs
97      #   high_red is the high red HSV threshold for detecting red LEDs
98      #   low_yellow is the low yellow HSV threshold for detecting yellow LEDs
99      #   high_yellow is the high yellow HSV threshold for detecting yellow LEDs
100
101     # This function returns x,y,baring and frame which are the x and y pixel coords of the ...
            centre of the detected ball,
102     # baring is the direction that robot is oreneted, defined as clockwise from the ...
            vertical being +ve
103
104
105     print('\n Finding ball position')
106
107
108     # Enter loop until ball found and baring extracted
109     ball_rad_pixels = 52
110
111     print('\n ball rad_pixels:',ball_rad_pixels)
112
113     while 1:
114         # the try is needed because the image detection for yellow is dodgy so sometimes fails
115         try:
116             #print('\n Ball Coordinates:',x1,',',y1,'\n')
117             # get baring and location from LED poisitions
118             x_red,y_red,r_red,frame = ball_location(low_red,high_red,ball_rad_pixels/4)
119
120             x_yellow,y_yellow,r_yellow,frame = ...
                    ball_location(low_yellow,high_yellow,ball_rad_pixels/4)
121
122             # filtering out false results by checking the distance between the red and ...
                    yellow LEDs
123             dist_xy =math.sqrt((x_red-x_yellow)**2+(y_red-y_yellow)**2)
124             print(dist_xy)
125
126             # check that the soloution found is reasonable
127             if dist_xy< 2*ball_rad_pixels:
128
129                 # calculate the robot's baring defined as +ve clockwise from Yaxis
130                 baring = math.atan2(y_red-y_yellow,x_red-x_yellow)
131
132                 # approximate robot centre
133                 x1 = int((x_red + x_yellow)/2)
134                 y1 = int((y_yellow + y_red)/2)
135
136                 # approximate radius into pixels and draw onto frame the detected results
137                 r1 = ball_rad*real2pixels
138                 output_red = cv2.circle(frame, (x_red,y_red), r_red,(0, 0, 255), 2)
139                 output_yellow = cv2.circle(frame, (x_yellow,y_yellow), r_yellow,(0, 255, ...
                        255), 2)
140                 output = cv2.circle(frame, (x1,y1), 50, (0,255 ,0 ), 2)
141                 cv2.imshow('Centre point',output)
142                 cv2.waitKey(20)
143
144
145
146                 # If a baring is found, then break the loop
147                 if baring is not None:
148
149                     print('\n Baring:', baring*180/math.pi)
150                     break
151         except:
152             print('unable to find suitable image solution, trying again')
153
154
155     return x1,y1,baring
```

```
156
157
158 class Node():
159     # A node class for A* Pathfinding
160     # credit to Nicholas Swift 2017
161     # Taken from https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2
162
163     def __init__(self, parent=None, position=None):
164         self.parent = parent
165         self.position = position
166
167         self.g = 0 # G is the distance between the current node and the start node
168         self.h = 0 # H is the estimated distance from current node to end node
169         self.f = 0 # F is the total cost of the node f = g + h
170
171     def __eq__(self, other):
172         return self.position == other.position
173
174 def astar(maze, start, end):
175     # Based off code by Nicholas Swift 2017
176     # can be found at ...
            https://medium.com/@nicholas.w.swift/easy-a-star-pathfinding-7e6689c7f7b2
177
178     # Returns a list of tuples as a path from the given start to the given end in the ...
            given binary maze
179
180
181
182     # Create start and end node
183     start_node = Node(None, start)
184     start_node.g = start_node.h = start_node.f = 0
185     end_node = Node(None, end)
186     end_node.g = end_node.h = end_node.f = 0
187
188     # Initialize both open and closed list
189     open_list = [] #
190     closed_list = []
191
192     # Add the start node
193     open_list.append(start_node)
194
195     # Loop until you find the end
196     while len(open_list) > 0:
197
198         # Get the current node
199         current_node = open_list[0]
200         current_index = 0
201         for index, item in enumerate(open_list):
202             if item.f < current_node.f:
203                 current_node = item
204                 current_index = index
205
206         # Pop current off open list, add to closed list
207         open_list.pop(current_index)
208         closed_list.append(current_node)
209
210         # Found the goal
211         if current_node == end_node:
212             path = []
213             current = current_node
214             while current is not None:
215                 path.append(current.position)
216                 current = current.parent
217             return path[::-1] # Return reversed path
218
219         # Generate children
220         children = []
```

```
221         for new_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), ...
                (1, 1)]: # Adjacent squares
222 
223             # Get node position
224             node_position = (current_node.position[0] + new_position[0], ...
                    current_node.position[1] + new_position[1])
225             #print(node_position)
226 
227 
228             # Make sure within range
229             if node_position[0] > (len(maze) - 1) or node_position[0] < 0 or ...
                    node_position[1] > (len(maze[len(maze)-1]) -1) or node_position[1] < 0:
230                 continue
231 
232             # Make sure the current node isnt already in the closed list
233             if Node(current_node, node_position) in closed_list:
234                 continue
235 
236             # Make sure walkable terrain
237             if maze[node_position[0]][node_position[1]] != 0:
238                 continue
239 
240 
241             # Create new node
242             new_node = Node(current_node, node_position)
243 
244             # Append
245             children.append(new_node)
246 
247         # Loop through children
248         for child in children:
249 
250             # Child is on the closed list
251             for closed_child in closed_list:
252                 if child == closed_child:
253                     break
254 
255             # Create the f, g, and h values
256             child.g = current_node.g + 1
257             child.h = ((child.position[0] - end_node.position[0]) ** 2) + ...
                    ((child.position[1] - end_node.position[1]) ** 2)
258             child.f = child.g + child.h
259 
260             # Child is already in the open list
261             for open_node in open_list:
262                 # check if the new path to children is worst or equal
263                 # than one already in the open_list (by measuring g)
264                 if child == open_node and child.g ≥ open_node.g:
265                     break
266 
267             # Add the child to the open list
268             open_list.append(child)
269 
270 def simplify_path(path_old, maze):
271     # A function to simplify a path through a binary maze
272 
273 
274     length = len(path_old)
275 
276     #print('length:',length)
277     path_new = path_old
278     n = 0
279     i = 0
280 
281     # loop throuogh each point
282     while i<length-2:
283 
284         # define the points that are being analysed
```

```
285             x1,y1 = path_new[n]
286             x2,y2 = path_new[n+1]
287             x3,y3 = path_new[n+2]
288 
289             # look at the gradient betweeen consecutive points
290             if x1 == x2:
291                 grad1 = math.inf
292             else:
293                 grad1 = (y2-y1)/(x2-x1)
294 
295             if x2 == x3:
296                 grad2 = math.inf
297             else:
298                 grad2 = (y3-y2)/(x3-x2)
299 
300 
301             # remove the point if it is on the same line as its predecessor
302             if grad1 == grad2:
303                 del path_new[n+1]
304 
305             # otherwise mve onto the next point
306             else:
307                 n = n+1
308             i = i+1
309 
310         print('\n Path New 1:',path_new)
311 
312 
313 
314         # now analyse whether or not it is possible to further simplify consecutive points ...
                further by skipping points
315 
316 
317         # Find the big contours/blobs on the binary image:
318         contours, hierarchy = cv2.findContours(maze,cv2.RETR_CCOMP, cv2.CHAIN_APPROX_SIMPLE)
319         length = len(path_new)
320         i = 0
321         n = 0
322 
323         # loop through each point in the new path list
324         while n<len(path_new)-2:
325 
326 
327             # create a connecting line between current point and the evaluated point
328             path_line = LineString([Point(path_new[n]),Point(path_new[n+2])]) # create polygoy ...
                    line which skips a point
329 
330             # initally set intersection to 0
331             intersection = 0
332 
333             # evaluate whether skipping this point causes you to hit the maze walls
334             # loop through each  maze contour
335             for j in range(0, len(contours)):
336 
337                 # create a polygon from the given contour
338                 contour = np.squeeze(contours[j])
339                 maze_poly = Polygon(contour)
340 
341                 # check whether the line created above intersects with current polygon
342                 if maze_poly.intersects(path_line):
343 
344                     # if there is an inersection make intersection 1
345                     intersection = 1
346 
347 
348 
349             # if the proposed path does not intersect the maze polygons,
350             # make this the new path by deleting any points between there and point n
```

```
351         if intersection == 0:
352
353             del path_new[n+1]
354
355
356
357         else : # otherwise increment n to move onto the next point
358             n = n+1
359
360
361
362
363     print('\n Path New 2:',path_new)
364     return path_new
365
366
367
368     n=0
369     while True:
370         #Read data
371         data_in, addr = s_UDP.recvfrom(1024) # buffer size is 1024 bytes
372
373         # Print data
374         print ("received bytes:", list(data_in)) # As byte values
375
376         if data_in == b'\x01':
377             n = n+1
378             break
379
380 def GimmeAone():
381     #Function which waits until a UDP signal of a 1 is recieved.
382     # This is required to integrate with robot as the stae machine returns a after every ...
          action is completed
383     n=0
384     while True:
385         #Read data
386         data_in, addr = s_UDP.recvfrom(1024) # buffer size is 1024 bytes
387
388         # Print data
389         print ("received bytes:", list(data_in)) # As byte values
390
391         if data_in == b'\x01':
392             n = n+1
393             break
394
395
396 ##      Claibrate camera       ###
397 os.getcwd() # Get current Working Directory
398
399 #Load the ArUco Dictionary Dictionary 4x4_50 and set the detection parameters
400 aruco_dict = aruco.Dictionary_get(aruco.DICT_4X4_50)
401 pa = aruco.DetectorParameters_create()
402
403 # Select the camera that is connected to the machine
404 cap = cv2.VideoCapture(0)
405
406 # Set the width and heigth of the camera to desired values
407 width = 1280         # in pixels
408 height = int(width*9/16)        # in pixels
409 cap.set(3,width)
410 cap.set(4,height)
411
412 # ## Set the exposure of the camera
413 cap.set(cv2.CAP_PROP_AUTO_EXPOSURE, 3)  # disable auto exposure
414 cap.set(cv2.CAP_PROP_EXPOSURE, -2)
415
416 # DEFINE INPUT DATA
417 ball_rad = 100 # ball radius in mm
```

```
418 # define the spacing between the aruco markers 0 and 1 that outline the area
419 Real_dist = 600   # in mm
420
421 # Define corner marker values
422 Desired_Ids = [0,1,2]
423
424 # Defining HSV threshold values use the script on the following link to help tune these ...
        values:
425 # ...
        https://medium.com/programming-fever/how-to-find-hsv-range-of-an-object-for-computer-vision-applications
426
427 low_yellow = np.array([8,93, 100])
428 high_yellow = np.array([31, 255, 255])
429
430 low_red = np.array([160,140, 140])
431 high_red = np.array([180, 255, 255])
432
433 low_green = np.array([40, 80, 80])
434 high_green = np.array([90, 255, 255])
435
436
437 Ids_Index = [0]*len(Desired_Ids) # Create an index to map the array locations of the ArUco ...
        codes
438 Ids_Location = [[0]*2]*len(Desired_Ids)
439
440 # Search camera image for aruco codes 0-3, these mark the 4 corners of the playing field
441 while(True):
442
443     # Start the performance clock
444     start = time.perf_counter()
445
446     # Capture current frame from the camera
447     ret, frame = cap.read()
448
449     # Convert the image from the camera to Gray scale
450     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
451
452     # Blur the gray scale image and compare the results
453     blur = cv2.GaussianBlur(gray,(5,5),0)
454
455     # Detect edges in the blurred scale image from the camera
456     canny_blur = cv2.Canny(blur,100,110)
457
458     # Run the ArUco detection function
459     corners, ids, rP = aruco.detectMarkers(gray, aruco_dict)
460
461     #Draw the detected markers as an overlay on the original frame
462     out = aruco.drawDetectedMarkers(frame, corners, ids)
463
464     try:
465
466         n = 0
467         # Loop through all the desired ArUco codes
468         for i in range(0,len(Desired_Ids)):
469             # Loop through all of the  detected aruco markers
470             for j in range(0,len(ids)):
471                 # Map the detected marker location to the correct array position
472                 if Desired_Ids[i]==ids[j]:
473                     #Ids_Index[i] = j
474                     Ids_Location[i] = corners[j][0][0]
475                     n = n+1
476                     print('n = ', n)
477
478         # Break the while loop when all of the desired codes are detected in one go
479         if n == len(Desired_Ids):
480             break
481     except:
482         print('No ArUco codes detected')
```

```
483
484     ### Display the images ###
485     # use for debugging purposes only
486
487     cv2.imshow('image-with-ArUco',out)
488
489     cv2.waitKey(20)
490
491
492
493
494 #print('Calibration ArUco Locations: ' , Ids_Location)
495 ArUco_0 = Ids_Location[:][0]
496 ArUco_1 = Ids_Location[:][1]
497
498 # Find the distance between ArUco codes 0 and 1
499 Pixels_dist = math.sqrt((ArUco_0[:][0]-ArUco_1[:][0])**2 + (ArUco_0[:][1]-ArUco_1[:][1])**2)
500
501 # work out the scalling factor to go from pixels to real life and vice versa
502 pixels2real = Real_dist/Pixels_dist
503 real2pixels = Pixels_dist/Real_dist
504
505
506 ## Test to see whether the scalling is accurate
507 Test_length = 270 # mm
508 ArUco_2 = Ids_Location[:][2] # this is the marker for where the ball ends up
509
510 Test_lenght_pixels  =  math.sqrt((ArUco_1[:][0]-ArUco_2[:][0])**2 + ...
        (ArUco_1[:][1]-ArUco_2[:][1])**2)
511 Test_lenght_real = Test_lenght_pixels * pixels2real
512 print('\n pixels lenght 1 to 2: ',Test_lenght_pixels)
513 print('\n Real length: ',Test_length,' Calculated length: ',Test_lenght_real,'mm \n')
514
515
516
517
518
519 ###############################################################################
520 #       Read the current position and orientation of the robot
521
522 x1,y1,baring = Robot_Baring_Location(low_red, high_red,low_yellow, high_yellow)
523
524
525 ###############################################################################
526 #       Observe the environment and determine the path through it to desired location
527
528
529 # capture a frame and convert to HSV
530 ret, frame = cap.read()
531 hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
532
533
534 # Apply green mask
535 Green_mask = cv2.inRange(hsv_frame, low_green, high_green)
536 #cv2.imshow('maassk',Green_mask)
537
538
539 # Pixelate the image to specified size
540 Pixelate_width = 72
541 Pixelate_height = int(Pixelate_width*9/16)
542
543 # calculate the conversion factors from the hd image to the pixelated realm
544 pixelate2HD = width/Pixelate_width
545 HD2pixelate = Pixelate_width/width
546
547 # pixelate the image to the specified size - mkaes computinf the path much faster
548 maze = cv2.resize(Green_mask,(Pixelate_width,Pixelate_height),interpolation=cv2.INTER_LINEAR)
549
```

```
550
551 # create an array to buffer new values in
552 buffer = maze*0
553
554 # work out how much the buffer needs to be offset by
555 ball_rad_maze = math.ceil(ball_rad*real2pixels*HD2pixelate)
556
557 # turn the colour mask into a binary maze with an added buffer
558 threshold = 160
559 for i in range(0,Pixelate_width):
560     for j in range(0,Pixelate_height):
561         if maze[j,i] > threshold:
562             maze[j,i] = 1
563
564             ## add a buffer to the map
565             if  i > ball_rad_maze and i < Pixelate_width-ball_rad_maze-1:
566                 if j > ball_rad_maze and j < Pixelate_height-ball_rad_maze-1:
567                     # make the boundries grow by the ball radius
568                     for k in range(0,ball_rad_maze):
569                         for l in range(0,ball_rad_maze):
570                             buffer[j+k,i+l] = 1
571                             buffer[j-k,i+l] = 1
572                             buffer[j+k,i-l] = 1
573                             buffer[j-k,i-1] = 1
574
575
576         else:
577             maze[j,i] = 0
578
579
580 # save a copy of the old maze then add buffer to new maze
581
582 maze_old = maze
583 maze = maze + buffer
584
585
586 # turn the 1s and 0s into black and white for display purposes
587 pixelate =(1- maze_old)*255
588 display = cv2.resize(pixelate,(width,height),interpolation=cv2.INTER_NEAREST)
589 cv2.imshow('mazaaae',display)
590 cv2.waitKey(20)
591
592
593 # determine the coordinates in the pixelated frame
594 Current_x_pixelated = int(x1*HD2pixelate)
595 Current_y_pixelated = int(y1*HD2pixelate)
596
597 endx_pixelated = int(ArUco_2[0]*HD2pixelate)
598 endy_pixelated = int(ArUco_2[1]*HD2pixelate)
599
600 maze[Current_y_pixelated,Current_x_pixelated] = 0
601
602
603 # check that goal is reachable:
604 if maze[Current_y_pixelated,Current_x_pixelated] ==1:
605     print('\n start location unreachable')
606 if maze[endy_pixelated,endx_pixelated]==1:
607     print('\n end location unreachable')
608
609
610 # determine the path using the astar function
611 path_old = astar(maze, ...
        (Current_y_pixelated,Current_x_pixelated),(endy_pixelated,endx_pixelated))
612
613
614
615 # convert the pixelated image back to rgb for adding plot features to it
616 pixelate = cv2.cvtColor(pixelate, cv2.COLOR_GRAY2BGR)
```

```
617
618 # create a green squares showing the path
619 for i in range(0,len(path_old)):
620     rows,columns = path_old[i]
621     pixelate[rows,columns][:] = [0,255,0]
622
623
624 # simplify the path
625 path_new = simplify_path(path_old, maze_old)
626 # add red points for the simplified path points
627 for i in range(0,len(path_new)):
628     rows,columns = path_new[i]
629     rows_new = int(rows*pixelate2HD)
630     columns_new = int(columns*pixelate2HD)
631     pixelate[rows,columns][:] = [0,0,255]
632     cv2.circle(frame, [columns_new,rows_new], 5,(255, 0, 0), 2)
633
634 # resize image and plot it
635 pixelate = cv2.resize(pixelate,(width,height),interpolation=cv2.INTER_NEAREST)
636 cv2.imshow('Output_image', frame)
637 cv2.imshow('Path',pixelate)
638 cv2.waitKey(20)
639
640
641
642 ######################################
643 ## communicate with robot:
644 ####################################
645 # This code is not included in the
646 # Setup UDP and TCP links
647 # First we need to get the LOCAL IP address of your system
648 UDP_IP = socket.gethostbyname(socket.gethostname())
649
650 # This is the IP address of the machine that the data will be send to
651 TCP_IP = "138.38.203.188"
652
653 # This is the LOCAL port I am expecting data (on the sending machine this is the REMOTE port)
654 TCP_PORT_OUT = 25000
655 UDP_PORT_IN = 50002
656
657 # # Create the socket for the UDP communication
658 # s_UDP = socket.socket(socket.AF_INET,    # Family of addresses, in this case IP ...
      (Internet Protocol) family
659 #                       socket.SOCK_DGRAM) # What protocol to use, in this case UDP (datagram)
660 # logging.info('UDP Socket successfully created')
661 # # Create the socket for the UDP communication
662 # s_TCP = socket.socket(socket.AF_INET,        # Family of addresses, in this case IP type
663 #                       socket.SOCK_STREAM)    # What protocol to use, in this case TCP ...
      (streamed communications
664 # logging.info('TCP Socket successfully created')
665
666 # # Bind to the socket and wait for data on this port
667 # s_UDP.bind((UDP_IP, UDP_PORT_IN))
668
669 # # Establish the connection with the remote Server using their IP and the port
670 # s_TCP.connect((TCP_IP, TCP_PORT_OUT))
671 # logging.info('Connected')
672
673
674 n = 0
675 ########################################
676 # COMMENCE DATA TRANSFER
677
678 # loop through all data points
679 for i in range(0,len(path_new)-1):
680     # load the current path point
681     (x1,y1) = path_new[:][i]
682     (x2,y2) = path_new[:][i+1]
```

```
683
684
685         ##############################
686         # Wait until 1 recieved from UDP initally
687         GimmeAone()
688
689
690
691         ################################
692         # Orient robot into correct baring
693
694         # Calculate the baring from the path
695
696         # find angle between neighboring points
697         if y1-y2 == 0:
698             angle = math.pi/2
699         else:
700             angle = math.atan2((x1-x2),(y1-y2))
701
702         # convert from radians to degrees
703         angle = angle*math.pi/180
704         baring = baring*math.pi/180
705
706         # work out the angle that the robot needs to turn
707         turn_angle = int((angle-baring)*180/math.pi)
708
709
710         # add 360 deg if turrn angle is negative
711         if turn_angle<0:
712             turn_angle = 360 + turn_angle
713
714
715         # put commands into the correct data format for comms with pi
716         if turn_angle> 180:
717             turn_data = turn_angle - 180
718             data = bytes([0,180,turn_data])
719             print('\ndata:',[0,180,turn_data])
720         else:
721             turn_data = turn_angle
722             data = bytes([0,turn_data,0])
723             print('\ndata:',[0,turn_data,0])
724
725         # check
726         # send data
727         s_TCP.send(data)
728         print('Angle Data sent')
729
730         ####################################
731         # Recieve a 1
732         # Wait for confermation from pi that action has been completed
733         GimmeAone()
734
735         ########################################
736         ## Send position data
737
738         # Find distance between neighboring points
739         dist = math.sqrt((x1-x2)**2 + (y1-y2)**2)
740
741         # convert into mm and then into number of turns
742         turns = dist*pixelate2HD*pixels2real*8.25/(math.pi*200)
743
744         # make into an integer for data transfer:
745         turns = int(turns) # scalling error with mm etc may be too low for it to be integerised
746         baring = 0
747         data = bytes([turns,baring,0])
748
749         # Send the message over the TCP socket
750         # send positional command
```

```
751
752         s_TCP.send(data)
753         print('PositionData sent')
754
755         ####################################
756         # Recieve a 1
757         # Wait for confermation from pi that action has been completed
758         GimmeAone()
759
760         ####################################
761         # Re-evaluate robot's status
762         x1,y1,baring = Robot_Baring_Location(low_red, high_red,low_yellow, high_yellow)
763
764         # Update the i+1 path location to the current location of the robot:
765         path_new[:][i+1] = (x1*HD2pixelate,y1*HD2pixelate)
766
767         # Redo the path planning:
768         # make the current robot
769         # Note to idiot, check the angle after angle command is executed
770
771     cv2.waitKey()
```

C Road crossing script

```
# Script to allow a robot to cross the road of with ArUco codes driving along it without being hit
# Created by Rob Borrett, University of Bath, Nov-2021
# Using parts of code supplied by I Georgilas


import cv2              # This is the vision library OpenCV
import numpy as np      # This is a library for mathematical functions for python (used later)
import time             # This is a library to get access to time-related functionalities
import os                # This is for finding the working directory
import cv2.aruco as aruco   # This library allows the detection of aruco codes
import math               # This library contains lots of mathematical functions
import matplotlib.pyplot as plt # This library is for plotting nice graphs
import logging           # This library will allow us to access the system clock for pause/sleep/delay actions
import socket            # This library will allow you to communicate over the network


def Find_ArUco(Desired_Ids):
    # A function to Search camera image for ArUco codes specified in the input Desired_Ids
    # This function returns the pixel coordinates of the detected ArUco codes, and the frame in which the codes werer detected

    Ids_Index = [0]*len(Desired_Ids) # Create an index to map the array locations of the ArUco codes
    Ids_Location = [[0]*2]*len(Desired_Ids)

    while(True):

        # Start the performance clock
        start = time.time()

        # Capture current frame from the camera
        ret, frame = cap.read()

        # Convert the image from the camera to Gray scale
        gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

        # Blur the gray scale image and compare the results
        blur = cv2.GaussianBlur(gray,(5,5),0)

        # Detect edges in the blurred scale image from the camera
        canny_blur = cv2.Canny(blur,100,110)

        # Run the ArUco detection function
        corners, ids,rp = aruco.detectMarkers(gray, aruco_dict)

        #Draw the detected markers as an overlay on the original frame
        out = aruco.drawDetectedMarkers(frame, corners, ids)



        try:

            n = 0
            # Loop through all the desired ArUco codes
            for i in range(0,len(Desired_Ids)):
                # Loop through all of the  detected aruco markers
                for j in range(0,len(ids)):
                    # Map the detected marker location to the correct array position
                    if Desired_Ids[i]==ids[j]:
                        #Ids_Index[i] = j
                        Ids_Location[i] = corners[j][0][0]
                        n = n+1
```

```
61                          print('n = ', n)
62
63              # Break the while loop when all of the desired codes are detected in one go
64              if n == len(Desired_Ids):
65                  break
66
67          except:
68              print('No ArUco codes detected')
69
70          ### Display the images ###
71
72          cv2.imshow('image-with-ArUco',out)
73          cv2.waitKey(20)
74          #end = time.time()
75      return Ids_Location, frame
76
77  def ...
        lane_check(top_left,bottom_right,Car_Ids,Ball_cross_time,direction,mode,Position_old,Position_new,ball_e
78      # A function to either detect when is a suitable time to cross the road, or to check ...
            whether a proposed time is suitable.
79      # Apologies for all the inpts
80      # top_left = coordinates of top left cropping point
81      # bottom_right =  coordinates of bottom right cropping point
82      # CarIds = the ArUco Ids of the cars entering from the direction that is being analysed
83      # Ball_cross_time = the time it will take the robot to cross the road
84      # direction = the diretction in which the aruco cars are travelling
85      # mode = either 'check' or 'discover' where check checks the feasibility of crossing ...
            the road at a given time,
86      #           and discover porposes new times until it finds a vlalid solution
87      # Position_old = the previous position of the ArUco codes
88      # Position_new = the current position of the ArUco codes
89      # ball_entry_time =  time robot enters the crossing zone
90      # Velocity  =  current velocity of ArUco cars
91      # Exit_Entry = current prediceted entry and exit times for the detected ArUco Codes
92
93      # This function returns the following information: ...
            ball_entry_time,Position_new,Position_old,Entry_Exit,Velocity
94      # ball_entry_time = None if no valid solution found or a specified time if a valid ...
            solution is found
95      # Position_new, Position_old are the old and new positions for the aruco cars. these ...
            are output to be re-input for the next iteeration, acting as a temporary memory
96      # Entry_Exit, Velocity = the newely calculated entry and exit times, and to the ...
            crossing zone, and car velocities. Again they are ouptut to be re input into the ...
            fuinction next time, acting as a sort of memory
97
98      # Capture current frame from the camera
99      ret, frame = cap.read()
100     # crop the frame to road section
101     ################################
102
103     # Convert the image from the camera to Gray scale
104     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
105
106     # Crop for the left frame
107
108     frame1 = gray[ top_left[1]: bottom_right[1], top_left[0]: bottom_right[0]] # crop ...
            between top left and end point
109
110     ## ArUco ###
111     # Run the ArUco detection function
112     corners1, ids1, rP1 = aruco.detectMarkers(frame1, aruco_dict)
113     #print(corners1)
114     #Draw the detected markers as an overlay on the original frame
115     out1 = aruco.drawDetectedMarkers(frame, corners1, ids1)
116     cv2.imshow('out1',out1)
117     cv2.waitKey(5)
118     # wait till cars have travelled a certian distance
119     int_clock_old = time.time()
```

```
120
121
122     time.sleep(0.5)
123
124     ret, frame = cap.read()
125     # crop the frame to road section
126     ################################
127
128     # Convert the image from the camera to Gray scale
129     gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
130
131     # Crop for the frame
132
133     frame2 = gray[ top_left[1]: bottom_right[1], top_left[0]: bottom_right[0]] # crop ...
            between top left and end point
134
135     #run the Aruco Detection function again
136     corners2, ids2, rP2 = aruco.detectMarkers(frame2, aruco_dict)
137
138     #print(corners2)
139     int_clock_new = time.time()
140     out2 = aruco.drawDetectedMarkers(frame2, corners2, ids2)
141     cv2.imshow('out2',out2)
142     cv2.waitKey(5)
143
144     # Process ArUcos if present
145     if (ids2 is not None)and(np.sum(ids1) == np.sum(ids2)):
146         # loop through all the specified ids
147         for i in range(0,len(Car_Ids)):
148             # loop through all of the measured ids
149             for j in range(0,len(ids2)):
150
151                 # this statement is needed in case there is a code detected in the first ...
                        instance but not the second
152                 # only proceed if
153                 if Car_Ids[i]==ids2[j]:
154                     # find the coresponding id index for the old aruco code
155                     i_old = np.where(ids1==Car_Ids[i])
156                     i_old = np.squeeze(i_old)
157
158                     # get the arcuco loctaion
159                     Position_old[i] = corners1[i_old[0]][0][0].astype(int)
160                     Position_new[i] = corners2[j][0][0].astype(int)
161                     #print('position new:',Position_new,'position old:',Position_old)
162                     # calculate the distance moved in pixels
163
164                     dist = math.sqrt((Position_new[i][0]-Position_old[i][0])**2 + ...
                            (Position_new[i][1]-Position_old[i][1])**2)
165                     #print('dist', dist)
166
167
168                     Velocity[i] = dist/(int_clock_new-int_clock_old)
169                     #print('Velocity',Velocity[i])
170
171                     # ALso calculate the distance btween current pos and the crossing point
172                     # currently approximated to the y distance to the midddle edge
173                     if direction == 'Left':
174                         to_crossing = len(frame1[:]) - Position_new[i][0]
175                         #to_crossing =  ...
                                math.sqrt((Position_new[i][0]-crossing_point[0])**2 + ...
                                (Position_new[i][1]-crossing_point[1])**2)
176                         #print(len(frame1[:]),'-',Position_new[i][0],'=',to_crossing)
177                         #print('to_crossing:',to_crossing)
178                     else:
179
180                         # if direction is right assign the y coordinate
181                         to_crossing = Position_new[i][0]
182
```

```
183
184                     # Determine Entry time to crossing zone
185                     if Velocity[i] == 0:
186                         #print('Break for zero velocity')
187                         break
188
189                     Entry_Exit[i][0] = to_crossing/Velocity[i] + int_clock_new ...
                            -crossing_width/(2*Velocity[i])
190                     print('\n ArUco',Car_Ids[i],' Predicted entry time:',Entry_Exit[i][0])
191                     #print('Current_time:',time.time())
192
193                     # Exit time to crossing zone
194                     Entry_Exit[i][1] = Entry_Exit[i][0] + crossing_width/Velocity[i]
195
196
197         # # Display the grey image in another window
198         #     out = aruco.drawDetectedMarkers(frame, corners1, ids1)
199         #     cv2.imshow('image-with-ArUco',out)
200
201         if mode == 'discover':
202             # loop through the next 10 s to find a time when it is suitable for the robot ...
                    to cross
203             for i in range (0,int(10/t_step)):
204
205                 # loop through each time step
206                 ball_entry_time = time.time() + 4 + i*t_step # + 4 included to give ...
                        checking time
207                 ball_exit = ball_entry_time + Ball_cross_time
208                 #print('\n Ball Entry time:',ball_entry_time )
209
210                 # loop through each exit and entry time
211                 total_feasibiity = 0
212                 for j in range(0, len(Entry_Exit)):
213                     feasibility = 0
214                     # make feasibilty 1 if this time instance is feasible for ecach car
215                     # if car has fully crossed by the time the ball enters the crossing zone
216                     if ball_entry_time≥ Entry_Exit[j][1]:
217                         feasibility = 1 # this time is feasible
218                     # if the ball leaves before the car enters the crossing zone
219                     if ball_exit≤ Entry_Exit[j][0]:
220                         feasibility = 1 # this time is feasible
221
222
223
224                     # create a running total
225                     total_feasibiity  = total_feasibiity + feasibility
226
227                 #print('\n Total Feasiblity:', total_feasibiity)
228
229                 # if all of the cars are feasible break the loop and use that time
230                 if total_feasibiity == len(Entry_Exit):
231                     #print('\n POTENTIAL SOLUTION FOUND AT TIME =',ball_entry_time)
232                     break
233
234                 else:
235                     ball_entry_time = None
236
237
238
239         elif mode == 'check':
240
241             ball_exit = ball_entry_time + Ball_cross_time
242              # if car has fully crossed by the time the ball enters the crossing zone
243             if ball_entry_time≥ Entry_Exit[j][1]:
244                 feasibility = 1 # this time is feasible
245             # if the ball leaves before the car enters the crossing zone
246             if ball_exit≤ Entry_Exit[j][0]:
247                 feasibility = 1 # this time is feasible
```

```
248
249             else:
250                 #print('check for feasability unsuccessfull')
251                 ball_entry_time = None
252
253
254
255     return ball_entry_time,Position_new,Position_old,Entry_Exit,Velocity
256
257
258
259 ######################################
260 ## Setup comms with robot:
261 ####################################
262
263 # Setup UDP and TCP links
264 # First we need to get the LOCAL IP address of your system
265 UDP_IP = socket.gethostbyname(socket.gethostname())
266
267 # This is the IP address of the machine that the data will be send to
268 TCP_IP = "138.38.203.188"
269
270 # This is the LOCAL port I am expecting data (on the sending machine this is the REMOTE port)
271 TCP_PORT_OUT = 25000
272 UDP_PORT_IN = 50002
273
274 # Create the socket for the UDP communication
275 s_UDP = socket.socket(socket.AF_INET,    # Family of addresses, in this case IP (Internet ...
        Protocol) family
276                       socket.SOCK_DGRAM) # What protocol to use, in this case UDP (datagram)
277 logging.info('UDP Socket successfully created')
278 # Create the socket for the UDP communication
279 s_TCP = socket.socket(socket.AF_INET,        # Family of addresses, in this case IP type
280                       socket.SOCK_STREAM)    # What protocol to use, in this case TCP ...
                            (streamed communications
281 logging.info('TCP Socket successfully created')
282
283 # Bind to the socket and wait for data on this port
284 s_UDP.bind((UDP_IP, UDP_PORT_IN))
285
286 # Establish the connection with the remote Server using their IP and the port
287 s_TCP.connect((TCP_IP, TCP_PORT_OUT))
288 logging.info('Connected')
289
290
291 n = 0
292 ##########################################
293 # COMMENCE DATA TRANSFER
294
295 # loop through all data points
296
297
298
299 os.getcwd() # Get current Working Directory
300
301
302 ### SETUP CAMERA ###
303
304
305
306 #Load the ArUco Dictionary Dictionary 4x4_50 and set the detection parameters
307 aruco_dict = aruco.Dictionary_get(aruco.DICT_4X4_50)
308 pa = aruco.DetectorParameters_create()
309
310 # Select the camera that is connected to the machine
311 # For an external camera this will be a number n>0
312 cap = cv2.VideoCapture(0)
313
```

```
314
315 # Set the width and heigth of the camera to desired values
316 width = 1280         # in pixels
317 height = int(width*9/16)        # in pixels
318 cap.set(3,width)
319 cap.set(4,height)
320
321 # ## Set the exposure of the camera
322 cap.set(cv2.CAP_PROP_AUTO_EXPOSURE, 0.25)
323 cap.set(cv2.CAP_PROP_EXPOSURE, -8)
324
325
326
327 ##      Find Geometric ArUco corner markers      ####
328
329 Geom_Ids = [3,4,5,6]
330
331 Geom_Ids_Location,frame = Find_ArUco(Geom_Ids)
332 ##############################################
333 # croppping and shit
334
335 start_point = Geom_Ids_Location[0].astype(int)      # Set ArUco 3 as start marker
336 end_point =  Geom_Ids_Location[1].astype(int)       # set ArUco 4 as end marker
337 crossing_point_left =  ...
        [int(end_point[0]+abs((start_point[0]-end_point[0]))*3/4),int(end_point[1])]  # ...
        crossing point midway between start and finish point
338 crossing_point_right =  ...
        [int(end_point[0]+abs((start_point[0]-end_point[0]))/4),int(end_point[1])]   # ...
        crossing point midway between start and finish point
339 top_left = Geom_Ids_Location[2].astype(int)         # set aruco 5 to top left coords
340 bottom_right = Geom_Ids_Location[3].astype(int)     #set aruco 6 to bottom right marker
341
342 Left_frame = frame[ top_left[0]:  end_point[0], top_left[1]:  end_point[1]] ...
                        # crop between top left and end point
343 Right_frame = frame[ start_point[0]:bottom_right[0] ,  start_point[1] : bottom_right[1]] ...
          # crop between srart point and top left
344
345 ##############################################
346
347
348 # Define car marker values
349
350
351
352 Car_Ids_left = [15,16,17,18,19]             # Cars entering from the left can have any of ...
        these values
353 Car_Ids_right = [10,11,12,13,14]            # Cars entering from the right can have any of ...
        these values
354 Ids_Index_left = [0]*len(Car_Ids_left)      # Create an index to map the array locations ...
        of the ArUco codes
355 Ids_Index_right = [0]*len(Car_Ids_right)    # Create an index to map the array locations ...
        of the ArUco codes
356
357 Velocity_left = [0]*len(Car_Ids_left)               # Store Aruco Velocity for cars ...
        entering from the left here
358 Velocity_right = [0]*len(Car_Ids_right)            # Store Aruco Velocity for cars ...
        entering from the right here
359 Position_old_left = [[0]*2]*len(Car_Ids_left)      # Store old Aruco positions here for ...
        cars entering from the left
360 Position_new_left = [[0]*2]*len(Car_Ids_left)      # Store new ArUco positions here for ...
        cars entering from the left
361 Position_old_right = [[0]*2]*len(Car_Ids_right)    # Store old ArUco positions here for ...
        cars entering from the right
362 Position_new_right = [[0]*2]*len(Car_Ids_right)    # store new ArUco positions here for ...
        cars entering from the right
363 Entry_Exit_left = [[0]*2]*len(Car_Ids_right)       # store Entry and exit times to the ...
        crossing zone for for cars entering from the left here
```

```
364 Entry_Exit_right = [[0]*2]*len(Car_Ids_right)       # store Entry and exit times to the ...
        crossing zone for for cars entering from the right here
365
366
367
368
369 # Define time step
370 t_step = 0.5
371
372 # Define and the time it takes the robot to cross the road in seconds
373 Ball_cross_time = 10
374
375 # define the pixel width of the crossing zone
376 crossing_width = 50 # pixels
377
378
379
380 # Start the performance clock
381 start = time.time()
382
383 int_clock_old = 0
384
385 while(True):
386     while(True):
387
388         while(True):
389             # Run the lane check function in discover mode for the left side of the road.
390             ball_entry_time,Position_new_left,Position_old_left,Entry_Exit_left,Velocity_left ...
                    =  ...
                    lane_check(top_left,end_point,Car_Ids_left,Ball_cross_time,'Left','discover',Position_old_le
391             # break loop when valid time proposed
392             if ball_entry_time is not None:
393                 break
394
395         # check this in the right lane
396         ball_entry_time,Position_new_right,Position_old_right,Entry_Exit_right,Velocity_right ...
                =  ...
                lane_check(start_point,bottom_right,Car_Ids_right,Ball_cross_time,'Right','check',Position_old_r
397         # break the loop if teh entry time is still valid
398         if ball_entry_time is not None:
399             print('\n SOLUTION CONFIRMED')
400             break
401
402     # wait until the time is reached and check that solution is still feasible
403
404     abort = 0 # initialise abourt variable as 0- currently crossing time is still valid
405
406     print('time',time.time())
407     while time.time()<ball_entry_time:
408         print('\n WAIT..... till:',ball_entry_time,' Current time: ',time.time())
409         # check that route is still feasible
410         # Capture current frame from the camera
411         ret, frame = cap.read()
412
413         # Convert the image from the camera to Gray scale
414         gray = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)
415
416         # crop the frame to road sections
417         Left_frame = gray[ top_left[1]:  end_point[1], top_left[0]:  end_point[0]] # crop ...
                between top left and end point
418         # cv2.imshow('\n left_frame',Left_frame)
419
420         # run the checking fucntion for the left side
421         ball_entry_time,Position_new_left,Position_old_left,Entry_Exit_left,Velocity_left ...
                =  ...
                lane_check(top_left,end_point,Car_Ids_left,Ball_cross_time,'Left','check',Position_old_left,Pos
422
423         # Entry time will be set to None if it is unfeasible
```

```
424         # break if unfeasible
425         if ball_entry_time is None:
426             print('\n MISSION ABORTED')
427             abort = 1
428             break
429         print('\n check 1 complete')
430 
431 
432         # crop the frame to road sections
433         Right_frame = gray[ start_point[1]:bottom_right[1] ,  start_point[0] : ...
                bottom_right[0]] # crop between start point and top left
434 
435 
436         # cv2.imshow('Right_frame',Right_frame)
437         # cv2.waitKey(20)
438 
439         # run the checking function for the right side
440         ball_entry_time,Position_new_right,Position_old_right,Entry_Exit_right,Velocity_right ...
                =  ...
                lane_check(start_point,bottom_right,Car_Ids_right,Ball_cross_time,'Right','check',Position_old_r
441 
442         # Abort mission if unfeasible
443         if ball_entry_time is None:
444             print('\n MISSION ABORTED')
445             abort = 1
446             break
447 
448         print('\n check 2 complete')
449 
450 
451 
452     # if by the end of the loop abort is still 0, then it is safe to cross the road!
453     if abort ==0:
454         print('\n LETS GO!!!!')
455         break
456 
457     # Otherwise, return to the start
458     print('\nReturning to start')
459 
460 
461 ########################################
462 # Communicate with robot
463 ########################################
464 
465 GimmeAone() # search for a 1 from the UDP signal
466 
467 print('sending data')
468 
469 # Tell the robot to move forward by sending the action for the ball to peform over TCP
470 position  = 115
471 baring1 = 0
472 baring2 = 0
473 
474 
475 data = bytes([position,baring1,baring2])
476 
477 
478 # Send the message over the TCP socket.
479 # send positional command
480 s_TCP.send(data)
481 print('Data sent')
482 
483 # Await confermation that the action has been completed
484 GimmeAone()
485 
486 # realease camera
487 cap.release()
```