|  |  |  |
| --- | --- | --- |
| University of Sussex Internal | **Candidate Number** | 237707 |
| **Module** | 825G5 |
| **Word Count** | 2223 |

**Airtags For Self-Navigating Robot in Restaurants Environment**



**Table of Content**

1. **Introduction ---------------------------------------------------------------------- 2**
2. **Methods --------------------------------------------------------------------------- 2**

2.1 Layout -------------------------------------------- 2

2.2 Disturbance ------------------------------------- 3

2.3 Routing Algorithm ------------------------------ 4

**3. Results and Analysis ---------------------------------------------------------- 4**

3.1Horizontal and Vertical Behaviour ---------------------- 4

3.2 Change in Distance in respect to Angle --------------- 5

3.3 Disturbance vs Time ---------------------------------------- 6

3.4. Routing Accuracy --------------------------------------------- 7

**4. Discussions ------------------------------------------------------------------------- 8**

**Appendix (Includes code but best check python file) -----------------------9**

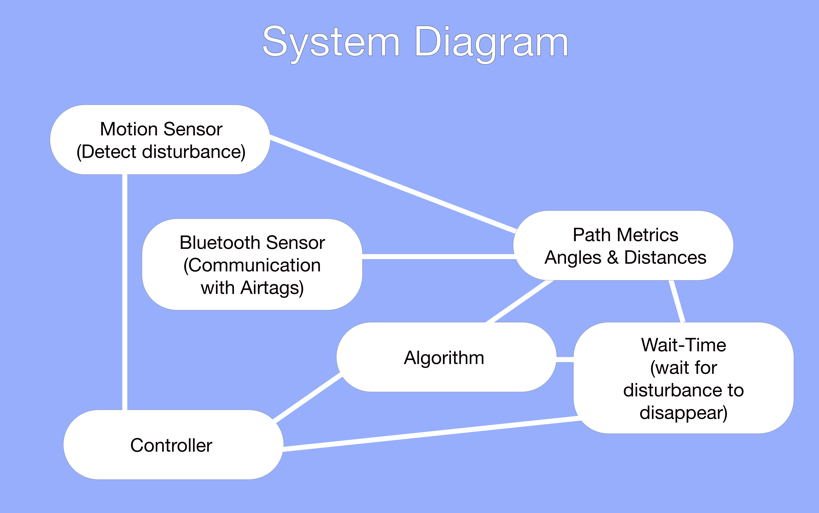
**References ------------------------------------------------------------------------------- 22**

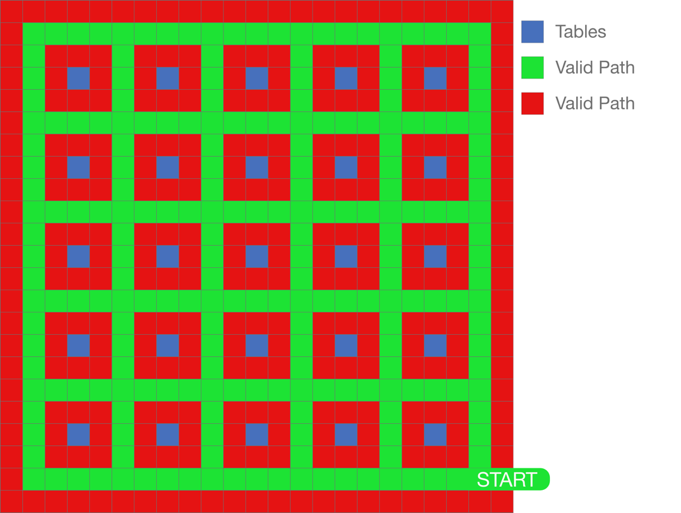
**1 - Introduction**

One of the major challenges of robotics is the fulfilment of a navigation architecture. Pfeiffer (2017) identified obstacle avoidance and safety distance to be among the many challenges in motion planning. Meanwhile, current recent studies have seen a boost with breakthroughs in fields of computer vision and machine learning with models such as deep neural networks (Frakiadaki, 2016) among others. In this respect, experiments often require heavy image and video analysis and are heavily reliant on data as well as computing power to make quick and effective decisions (Sunderhauf, 2018). As opposed to using expensive sensors and powerful CPU’s coupled with sophisticated models that can identify various features, our robot waitress will be limited to a simple motion sensor to identify disturbance as well as a bluetooth sensor designed to identify current distance and angle from the target table.

In an adaptive system, “some of the system parameters, based on some specific criterion are adjusted iteratively over time so as to make the system operate in, or as close to, optimum fashion” (Grami, 2016). Respectively, the aim of this report is to develop a simple yet efficient algorithm to understand the importance of the two parameters over time with the aim of achieving an adaptive routing capability. Our experiment will be limited to a controlled environment with the assumption that Airtag’s are placed on the restaurant tables. Although our experiment will be limited in scale, we aim to identify how our parameters can be scaled towards a more general-purpose routing mechanism in robotics.

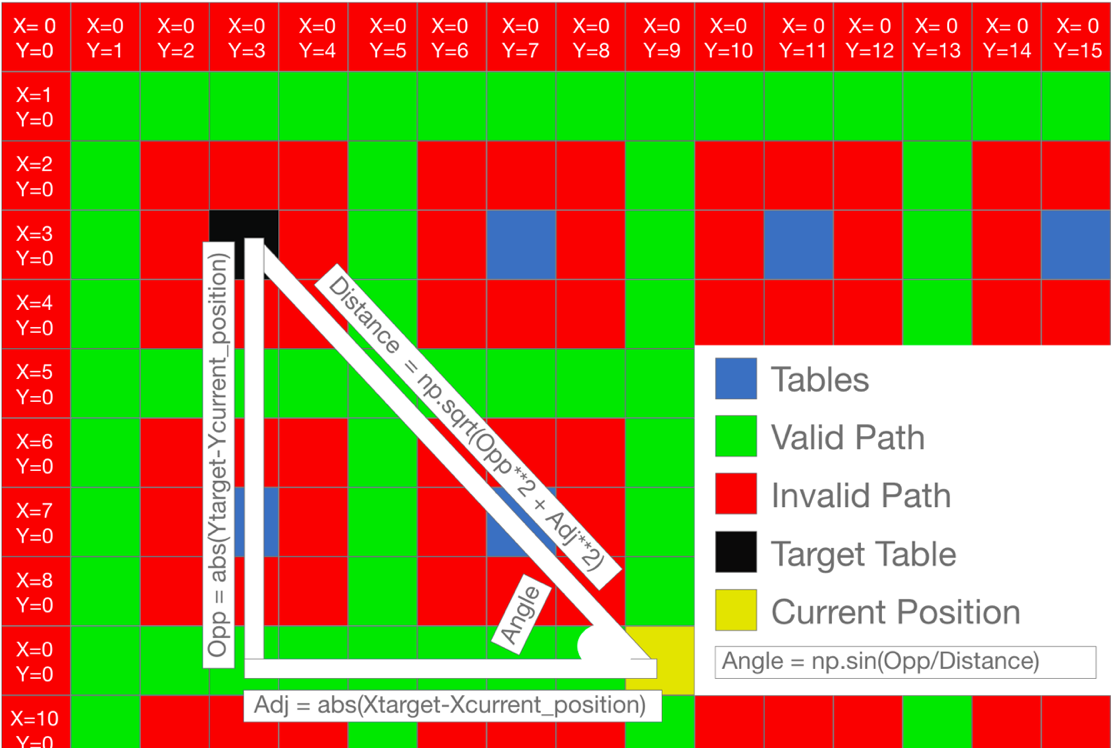
**2 - Methods**

**2.1 – Layout**



We have considered a medium sized restaurant with a total of 25 tables *[appendix 1]*. In our experiment our map will include pre-determined pathways where the robot is allowed to maneuver around. The robot is tasked with finding the optimal route by selecting paths after every one-meter movement based on the number of available valid paths as seen in the layout map above.

Our sensor can identify our target table from ranges of up to 100 meters with the Airtag’s placed on the tables. With this assumption, we can compute distances and angles for every available path for which our robot can take a step *[appendix 2, 4]*. These parameters will then run through our algorithm which aims to determine the best available path to take to get to the target table.



* The X and Y positions of each block are fitted to match the indexing of a python list.
* We can compute distance to target table using Pythagoras theorem:
  + Using the values absolute values: | and the distance can be calculated
* Using the trigonometry rule we can calculate our angle from the target table:
* Having found the key parameter provided by the Airtags, we can find the distances for the following pathways:
  + Left:
  + Right:
  + Up:
  + Down:

2.2 – Disturbance

For every 1-meter step that is taken, one of the available paths may face a disturbance. This disturbance is determined at random using distribution *[appendix 3]*. In the event that the disturbance has left us with only one pathway, we will trigger our deadlock avoidance mechanism. Initially, we will run the wait time mechanism before triggering the deadlock avoidance algorithm to effectively reroute.

|  |
| --- |
| **Wait-Time Mechanism** *[appendix 7]* |
| This algorithm waits up to 30 seconds for the disturbance to disappear. If the disturbance disappears, the robot may continue its path. Otherwise, the robot will opt to reroute its path to the target table. |

|  |  |
| --- | --- |
| **Deadlock Avoidance Algorithm** *[appendix 9]* | |
|  | Once the deadlock avoidance algorithm is triggered. The robot will keep going in the only available direction, until it is presented with the option to make a horizontal or vertical shift. Subsequently, it will revert back to the original algorithm which is tasked to take the robot to the target table. |

**2.3 – Routing Algorithms**

In our experiment, we will employ two different algorithms designed to find the optimal and updates for every 1-meter step that is taken. The aim is to test both algorithms against different disturbance distributions to better understand the value behind the parameters that are updated in communication with the Airtags. Movement directions are represented with digits from 0 to 3. Respectively, 0 represents a movement to the left, 1 to the right, 2 upwards and 3 downwards.

* **Algorithm 1 - Angle and Distance Minimizer** *[appendix 5]***:**

By finding the values of the minimum distance and angle among the available pathways at the given position, we are able to find the pathways with minimum distance or angle. Our condition favourably selects pathways with both the minimum angle and distance if they both correspond the same direction. Otherwise, it selects the path with the minimum angle.

* **Algorithm 2 - Vertical and Horizontal Distance Minimizer** *[appendix 6]***:**

Our second algorithm firstly identifies paths that would work towards reducing the difference between our X and Y parameters for both the target table and current position. This is done by passing conditions to ensure the path is firstly valid and secondly to ensure that our moments are either reducing the vertical or horizontal distance from the target. For every valid path that meets our condition, a random sample is selected as the next move. Otherwise, a random sample of either the minimum angle or distance is selected.

**3 – Results and Analysis**

**3.1 – Horizontal and Vertical Behaviour**

* **Iterations:** 300
* **Table Selection:** random
* **Aim:** Identifying the behaviour within the minimizing vertical and horizontal distances from target as well changing disturbance rates.
* **Method:** For each iteration the lower quartile, median and upper quartile will be stored. Once 300 iterations are completed the same 3 data points will be selected for each category and plotted against a of 0 – 15 representing the journey *[appendix 10, 11]*.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Disturbance Rates** | | | | | | | |
|  | 20% |  | 10% |  | 5% |  | 0% |

|  |  |
| --- | --- |
| **Algorithm 2** | **Algorithm 1** |
|  |  |
| **Algorithm 2** | **Algorithm 1** |
|  |  |

Both algorithms appear to adapt in the face of disturbances on average after 300 iterations. We can see losses in our vertical distance increasing as larger disturbance rates are applied. While both algorithms appear to begin by taking a vertical shift before minimizing the horizontal distance toward the end of the journey, it is noteworthy that algorithm 2 incurs greater losses when minimizing its horizontal distance (specially at 20% disturbance). However, both algorithms appear to behave in a more consistent manner when minimizing their vertical distance. This can be articulated as both algorithms undertaking an effective route for the first half of the journey before algorithm 2 faces a decline in its performance*.* Although both algorithms appear to behave somewhat consistently, it is perhaps the element of randomness in algorithm 2 along with focus on angles minimization that gives algorithm 1 a small edge.

**3.2 – Change in Distance in Respect to Angle**

* **Iterations:** 150
* **Table Selection:** X=3, Y=3 (furthest table)
* **Aim:** Identifying relation between distance and angle as well as adaptivity in response to different disturbance distributions.
* **Method:** For each iteration min, lower quartile, median, upper quartile and will be stored. Once 150 iterations are completed the same 5 data points will be selected for each category and plotted against a of 0 – 15 representing the journey *[appendix 10, 11].*

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Disturbance Rates** | | | | | | | |
|  | 20% |  | 10% |  | 5% |  | 0% |

|  |  |
| --- | --- |
| **Algorithm 2** | **Algorithm 1** |
|  |  |

The plot above shows that both algorithms are capable of adapting to reach their target table despite disturbances. Although algorithm 2 is not particularly designed to minimize its angle, both algorithms decrease distance before sharply decreasing angles in normal conditions. This displays a correlation in respect to the reduction of distance and angles in the paths taken by both algorithms. Subsequently, when disturbance is increased to 5% algorithm 2 slightly diverges possibly due the random selection of vertical/horizontal distances which impacts the angles. On the other hand, algorithm 1 continues to follow the pattern. However, when disturbance is increased to 10% or 20% both algorithms face heavy divergence before minimizing their distance and angles from the target table highlighting their abilities to re-route and eventually adapt over time.

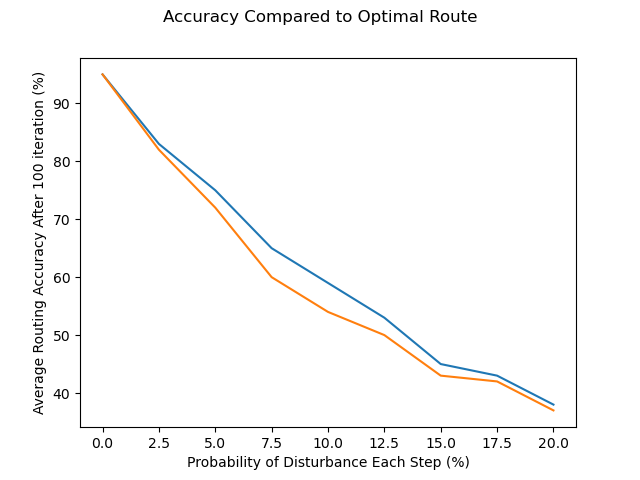
**3.2 – Disturbance vs Time**

* **Iterations:** 100
* **Table Selection:** X=3, Y=3 (furthest table)
* **Aim:** Determine how disturbance impacts the delivery time and how the system adapts to different disturbance distributions and compare the algorithms.
* **Method:** Time measured as 1 second for every straight path and 3 seconds for every time direction is altered. For each iteration total delivery time and total disturbances are stored to plot 100 dots on our scatter plot *[appendix 12]*.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Disturbance Rates** | | | | | | | |
|  | 20% |  | 10% |  | 5% |  | 0% |

|  |  |
| --- | --- |
| **Algorithm 2** | **Algorithm 1** |
|  |  |

As exemplified in the scatter plot above, it is evident that higher disturbance rates call for longer delivery times. Both algorithms appear to perform similarly with low disturbance rates with most deliveries completed in less than 50 seconds without disturbance and 100 second with 5% disturbance. Subsequently, when disturbance is increased to 10% and 20%, a small but noticeable difference starts to emerge. Algorithm 2 appears to have encountered more disturbances suggesting longer routes have been undertaken in reaching its target. Respectively, the delivery times appear to have been impacted due the extra steps taken. Therefore, algorithm 1 can be seen to be slightly more adaptive.



**3.4 – Routing Accuracy**

* **Iterations:** 300
* **Table Selection:** X=3, Y=3 (furthest table)
* **Aim:** Understand how accurate our algorithm are without disturbance how that accuracy is impacted by disturbance.
* **Method:** Find accuracy for each iteration using and find the mean accuracy for different disturbance rates.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Disturbance rates** | **0%** | **2.5%** | **5%** | **7.5%** | **10%** | **12.5%** | **15%** | **17.5%** | **20%** |
| Algorithm 1 | ﻿95% | ﻿ ﻿83% | 75% | 65% | 59% | 53% | 45% | 43% | 38% |
| Algorithm 2 | ﻿95% | 82% | 72% | 60% | 54% | 50% | 43% | 42% | 37% |
| **Disturbance Rates** | | | |
|  | Algorithm 1 |  | Algorithm 2 |

Our accuracy scores further add weight to our previous conclusion that algorithm 1 performs better in response to disturbance. Although both algorithms appear to have similar and yet high accuracies with no disturbance, we see a sharp drop in both algorithms. However, the decline rate appears to stabilize a bit from disturbance rates of 15% - 20%. Despite the poor performance with higher disturbance rates, one must remember that our optimal route is a route where no disturbance is faced and solely only used for testing purpose. Moreover, in a normal environment it is not logical that our robot will face disturbance rate higher than 7.5%. In this respect, both algorithms demonstrated their abilities to adapt.

**3 – Results and Analysis**

Although the simulation was unable to definitively compare the routing accuracies with the application of disturbance, both algorithms scored 95% without disturbance. The deadlock avoidance algorithm also appeared to be effective considering both algorithms managed to find a route over time. In this respect, one can assume that our accuracy score would be somewhat consistent with 95% minus effectiveness of the deadlock algorithm in the precedence of disturbance. Therefore, the simulation demonstrated that our algorithm “adjusted” our key “parameters [angle & distance] to make the system operate… as close to as optimum fashion” meeting Grami (2016) definition of an adaptive system.

Our results also fulfilled our objective of demonstrating the importance our core parameters in angle and distance. While both algorithms had consistent results with one another, Algorithm 1 which focused on minimizing the angle appeared have a slight edge with an average accuracy of 61% (with disturbance ranging from 0-20%) compared to optimal route (without disturbance). On the other hand, algorithm 2 which focused randomly minimizing horizontal or vertical distance (where possible) scored 59%. In this respect, Airtags can be simple, cheap and yet useful implementations for in robot navigation in controlled environments (e.g. restaurant with Airtags). Although, we were not able to relate to any past research particularly to the use of Airtags, our experiment relates from a different angle to wider studies in robot navigation.

Despite our simulation being very limited in both scope and scale, our results indicate that our parameters are effective, and our implementation can be scaled in various ways. Our restaurant map was very straight forward and only allowed for a maximum of 4 directions in the most ideal circumstance. Moreover, the simulation only communicated with the Airtag placed on the target table rather than all tables. In this respect, one area of interest to explore would be using all 25 Airtags for the robot to better understand its environment and expand the extent to which the robot is able to manoeuvre.

**Appendix (Check .py file for code references)**

|  |
| --- |
| **Appendix 1:** Set up of restaurants mapping and block features |
| ﻿**class** **Blocks**(**object**):    **def** **\_\_new\_\_**(cls, \*args, \*\*kwargs):  instance = **super**(Blocks, cls).\_\_new\_\_(cls)  **return** instance    **def** **\_\_init\_\_**(**self**, position\_X, position\_Y, valid\_path, table):  **self**.position\_X = position\_X  **self**.position\_Y = position\_Y  **self**.valid\_path = valid\_path # Boolean to validate if the robot can travel on the path  **self**.table = table # Boolean to validate if there is a table on the block  **self**.disturbance = False   **class** **Create\_Maze**:    **def** **\_\_init\_\_**(**self**):  **self**.maze = []    **def** **create\_invalid\_path**(**self**, idx):  **for** i **in** **self**.maze[idx]:  i.valid\_path = False  i.table = False    **def** **create\_valid\_path**(**self**, idx):  a = 0  **for** i **in** **self**.maze[idx]:  a += 1  i.valid\_path = True  i.table = False  **if** a == 1 **or** a == 23:  i.valid\_path = False    **def** **create\_simi\_valid\_path**(**self**, idx):  a = 0  **for** i **in** **self**.maze[idx]:  a += 1  i.valid\_path = False  i.table = False  **if** a **in** (2, 6, 10, 14, 18, 22):  i.valid\_path = True    **def** **create\_tables**(**self**, idx):  a = 0  **for** i **in** **self**.maze[idx]:  a += 1  i.table = False  **if** a **in** (2, 6, 10, 14, 18, 22):  i.valid\_path = True  **if** a **in** (4, 8, 12, 16, 20):  i.table = True    **def** **add\_positions**(**self**):  x = 0  y = 0  **for** i **in** **self**.maze:    y = 0  **for** a **in** i:   a.position\_X = x  a.position\_Y = y  y += 1  x += 1    **def** **maze\_setup**(**self**):  **self**.add\_positions()  **for** i **in** range(23):  **if** i **in** (1, 23):  **self**.create\_invalid\_path(i-1)  **if** i **in** (2, 6, 10, 14, 18, 22):  **self**.create\_valid\_path(i-1)  **if** i **in** (3, 5, 7, 9, 11, 13, 15, 17, 19, 21):  **self**.create\_simi\_valid\_path(i-1)  **if** i **in** (4, 8, 12, 16, 20):  **self**.create\_tables(i-1)    **def** **show\_maze**(**self**, idx):  maze = **self**.maze  **for** i **in** maze[idx]:  **if** i.table == True:  print("TABLE" ,i.position\_X-1, i.position\_Y-1)  **if** i.valid\_path == True:  print("VALID" ,i.position\_X-1, i.position\_Y-1)  **if** i.valid\_path == False **and** i.table == False:  print("INVALID" ,i.position\_X-1, i.position\_Y-1) |
| ﻿﻿mze = Create\_Maze()  **lis** = []  **for** i **in** **range**(24):  **if** i > 0:  mze.maze.**append**(**lis**)  **lis** = []  **for** a **in** **range**(23):  **lis**.**append**(Blocks(0, 0, False, False))   mze.maze\_setup() **run** = Run\_Maze(mze.maze) data = Data\_Collection(**run**) **run**.select\_table() **print**("TARGET TABLE IS", **run**.target\_table\_X, **run**.target\_table\_Y)  best\_route = **run**.return\_optimal\_route()  **run**.dist\_prob = 0 |

|  |
| --- |
| **Appendix 2:** Calculating the distance and angle from a given position number (0-3) or if its for current position using False parameter. |
| ﻿﻿def find\_dist\_ang(self, idx):  current\_X = self.current\_position\_X  current\_Y = self.current\_position\_Y    **if** idx == False:  pass  elif idx == 0:  current\_X -= 1  elif idx == 1:  current\_X += 1  elif idx == 2:  current\_Y -= 1  elif idx == 3:  current\_Y -= 1    width = abs(current\_Y - self.target\_table\_Y)  height = abs(current\_X - self.target\_table\_X)  distances\_squared = height\*\*2 + width\*\*2  distance = np.sqrt(distances\_squared)  angle = np.sin(height/distance)    return distance, angle |

|  |
| --- |
| **Appendix 3:** Setting disturbance. |
| ﻿﻿**def** **has\_disturbance**(**self**):  **if** **self**.move\_num == 0:  **return** False  val = [True, False]  prob\_1 = **self**.dist\_prob  prob\_2 = 1 - prob\_1  distribution = [prob\_1, prob\_2]  has\_disturbance = random.choices(val, distribution)[0]    **return** has\_disturbance |

|  |
| --- |
| **Appendix 4:** Removing pathway with disturbance and finding the distances and angles as well as direction number (0-3) for every available path at from current position. |
| ﻿﻿ **def** **paths**(**self**):  left\_path = **self**.maze[**self**.current\_position\_X][**self**.current\_position\_Y - 1]  right\_path = **self**.maze[**self**.current\_position\_X][**self**.current\_position\_Y + 1]  up\_path = **self**.maze[**self**.current\_position\_X - 1][**self**.current\_position\_Y]  down\_path = **self**.maze[**self**.current\_position\_X + 1][**self**.current\_position\_Y]  **return** left\_path, right\_path, up\_path, down\_path       **def** **valid\_paths**(**self**):   left\_path, right\_path, up\_path, down\_path = **self**.paths()  paths = [left\_path, right\_path, up\_path, down\_path]  val = **self**.has\_disturbance()  count = -1   idx = 100   valid\_paths = []   **for** i **in** paths:  count += 1   **if** i.valid\_path == True:  valid\_paths.append(count)    random\_item = random.choice(valid\_paths)  **if** val == True:  **self**.disturbance = True   valid\_paths.remove(random\_item)  idx = random\_item     **return** valid\_paths, len(valid\_paths), idx    **def** **path\_metrics**(**self**):  valid\_paths, \_, \_ = **self**.valid\_paths()  distances = []  angles = []   **for** idx **in** valid\_paths:  dist, ang = **self**.find\_dist\_ang(idx)  distances.append(dist)  angles.append(ang)    **return** valid\_paths, distances, angles |

|  |
| --- |
| **Appendix 5:** Algorithm 1 |
| ﻿ ﻿ **def** path\_selections(self):  valid\_paths, distances, angles = self.path\_metrics()  min\_distance = valid\_paths[(distances.index(min(distances)))]  min\_angle = valid\_paths[(angles.index(min(angles)))]  max\_angle = valid\_paths[(angles.index(max(angles)))]  **return** min\_distance, min\_angle, max\_angle, len(valid\_paths)    **def** algorithm1(self):  \_, \_, idx = self.valid\_paths()  min\_distance, min\_angle, max\_angle, num\_paths = self.path\_selections()  timer, timer\_val = self.timer\_algorithm()    **if** idx != 100 and timer\_val == False:  self.disturbance = True   self.wait\_time += timer  self.update\_position(idx)  elif num\_paths == 1:  self.disturbance = True   self.reroute\_val = True  self.reroute()  self.wait\_time += timer  elif min\_distance == min\_angle:    self.update\_position(min\_distance)  **return** min\_distance  **else**:  self.update\_position(min\_angle)  **return** min\_angle    self.reroute\_val = False |

|  |
| --- |
| **Appendix 6:** Algorithm 2 |
| ﻿﻿ **def** XY\_minimizer(**self**):  valid\_paths, \_, \_ = **self**.path\_metrics()  val = []    **if** 0 **in** **tuple**(valid\_paths) **and** **self**.current\_position\_Y > **self**.target\_table\_Y:  val.append(0)  **if** 1 **in** **tuple**(valid\_paths) **and** **self**.current\_position\_Y > **self**.target\_table\_Y:  val.append(1)  **if** 2 **in** **tuple**(valid\_paths) **and** **self**.current\_position\_X > **self**.target\_table\_X:  val.append(2)  **if** 3 **in** **tuple**(valid\_paths) **and** **self**.current\_position\_X > **self**.target\_table\_X:  val.append(3)  return val  **def** random\_selector(self):  valid\_paths, distances, angles = self.path\_metrics()  min\_dist = valid\_paths[distances.index(min(distances))]  min\_ang = valid\_paths[angles.index(min(angles))]  val = self.XY\_minimizer()  if len(val) == 0:    return random.sample([min\_dist, min\_ang, min\_ang],1)[0]    else:    return min(val)   **def** algorithm2(**self**):  \_, \_, idx = **self**.valid\_paths()  min\_distance, min\_angle, max\_angle, num\_paths = **self**.path\_selections()  timer, timer\_val = **self**.timer\_algorithm()   **if** idx != 100 **and** timer\_val == **False**:  **self**.disturbance = **True**  **self**.wait\_time += timer  **self**.update\_position(idx)  elif num\_paths == 1:  **self**.disturbance = **True**   **self**.reroute\_val = **True**  **self**.reroute()  **self**.wait\_time += timer  **else**:  random = **self**.random\_selector()  **self**.update\_position(random)    **self**.reroute\_val = **False** |

|  |
| --- |
| **Appendix 7:** 30 Second wait-time algorithm |
| ﻿  **def** **timer\_algorithm**(self):  val = [False, True]  prob\_1 = 0.1  prob\_2 = 1 - prob\_1  distribution = [prob\_1, prob\_2]    **for** i **in** range(30):  wait = random.choices(val, distribution)[0]  **if** wait == False:     **return** i, False    **return** 33, True |

|  |
| --- |
| **Appendix 8**: Random wait-time algorithm |
| ﻿﻿**def** **random\_algorithm**(self):  val = [True, False]  prob\_a1 = 0.02  prob\_a2 = 0.03  prob\_b1 = 1 - prob\_a1  prob\_b2 = 1 - prob\_a2  distribution\_a = [prob\_a1, prob\_b1]  distribution\_b = [prob\_a2, prob\_b2]  wait = random.choices(val, distribution\_a)[0]  leave = random.choices(val, distribution\_b)[0]    **for** i **in** range(30):  wait  leave  **if** wait == True:      **return** i, False  **elif** leave == True:  self.wait\_time += i + 3      **return** 33, True |

|  |
| --- |
| **Appendix 9:** Rerouting |
| ﻿ **def** **change\_direction**(**self**, vertical):  valid\_paths, distances, angles = **self**.path\_metrics()  vertical\_idx = []  horizontal\_idx = []  vert\_dist = []  hor\_dist = []  **for** i **in** valid\_paths:  **if** i **in** (0, 1):  vertical\_idx.append(i)  vert\_dist.append(distances[valid\_paths.index(i)])  **if** i **in** (2, 3):  horizontal\_idx.append(i)  hor\_dist.append(distances[valid\_paths.index(i)])    **if** vertical == True **and** len(vertical\_idx) != 0:  **return** vertical\_idx[vert\_dist.index(min(vert\_dist))]  elif vertical == False **and** len(horizontal\_idx) != 0:  **return** horizontal\_idx[hor\_dist.index(min(hor\_dist))]  else:  **return** random.sample(valid\_paths, 1)[0]            **def** **is\_vertical**(**self**):   **for** i **in** **self**.path\_mapping:  **if** i.get("move\_num") == **self**.move\_num - 1 **and** i.get("position\_X") == **self**.current\_position\_X:  **return** True  **return** False     **def** **previous\_position**(**self**, num):  **for** i **in** **self**.path\_mapping:  **if** i.get("move\_num") == **self**.move\_num - num:  **return** i.get("position\_X"), i.get("position\_Y")  **return** **self**.previous\_position\_X, **self**.previous\_position\_Y      **def** **reroute**(**self**):  print()  print("DISTURBANCE!!!!!")  is\_vertical = **self**.is\_vertical()   num = 1    **for** i **in** range(10):  num += 1  val = **self**.horizontal\_vertical()  paths, \_, \_ = **self**.path\_metrics()  **if** val == False:  **self**.current\_position\_X, **self**.current\_position\_Y = **self**.previous\_position(num)  **self**.append\_data()  num += 1  elif val == True:  new\_direction = **self**.change\_direction(is\_vertical)  **self**.update\_position(new\_direction)  **self**.append\_data()  **self**.update\_position(new\_direction)  **break** |

|  |
| --- |
| **Appendix 10:** Resetting |
| ﻿ **def** **append\_data**(**self**):    is\_vertical = **self**.is\_vertical()  X = abs(**self**.current\_position\_X - **self**.target\_table\_X)  Y = abs(**self**.current\_position\_Y - **self**.target\_table\_Y)  move\_data = {"move\_num": **self**.move\_num,  "position\_X": **self**.current\_position\_X,   "position\_Y": **self**.current\_position\_Y,   "range\_X": X,  "range\_Y": Y,  "distance": **self**.current\_distance,   "angle": **self**.current\_angle,  "disturbance": **self**.disturbance,  "is\_vertical": is\_vertical,  "reroute": **self**.reroute\_val}    **self**.disturbance = False  **self**.path\_mapping.append(move\_data)  print(move\_data)  **self**.move\_num += 1  **self**.optimal\_delivey\_time = **self**.return\_optimal\_route()     **def** **update\_position**(**self**, new\_position\_index):  **if** new\_position\_index == 0:  **self**.current\_position\_Y -= 1  **if** new\_position\_index == 1:  **self**.current\_position\_Y += 1  **if** new\_position\_index == 2:  **self**.current\_position\_X -= 1  **if** new\_position\_index == 3:  **self**.current\_position\_X += 1    **self**.update\_ang\_dist()    **def** **update\_ang\_dist**(**self**):  **self**.previous\_distance = **self**.current\_distance  **self**.previous\_angle = **self**.current\_angle  **self**.current\_distance, **self**.current\_angle = **self**.find\_dist\_ang(False) |

|  |
| --- |
| **Appendix 11:** Collecting data for “Horizontal and Vertical Behavior” and “Change in Disturbance in Respect to Angle.” |
| ﻿﻿﻿﻿ ﻿**class** **Data\_Collection**:  **def** **\_\_init\_\_**(**self**, run):  **self**.run = run  **self**.path\_mapping = **self**.run.path\_mapping    # Data Collection  **self**.velocities = []  **self**.disturbances = []  **self**.times = []  **self**.angles = []  **self**.distances = []  **self**.reroutes = []  **self**.range\_X = []  **self**.range\_Y = []    **self**.target\_table\_X = **self**.run.target\_table\_X  **self**.target\_table\_Y = **self**.run.target\_table\_Y    **self**.optimal\_route = **self**.run.optimal\_delivey\_time  **self**.total\_disturbance = 0  **self**.wait\_time = **self**.run.wait\_time   **self**.disturbance = 0  **self**.count\_reroute = 0  **self**.distance\_covered = 0  **self**.time = 0  **self**.velocity = 0    **self**.move\_num = []  **self**.position\_X = []  **self**.position\_Y = []  **self**.distance = []  **self**.angle = []     **def** **return\_data**(**self**):  move\_num = []  range\_X = []  range\_Y = []  distance = []  angle = []  print()  **for** i **in** **self**.path\_mapping:  move\_num.append(i.get("move\_num"))  range\_X.append(i.get("range\_X"))  range\_Y.append(i.get("range\_Y"))  distance.append(i.get("distance"))  angle.append(i.get("angle"))     **self**.move\_num = move\_num  **self**.position\_X = range\_X  **self**.position\_Y = range\_Y  **self**.distance = distance  **self**.angle = angle  ﻿﻿. **def** **x\_against\_y**(**self**):  array = np.asarray(**self**.position\_X)  array1 = np.percentile(array, [0, 25, 50, 75, 100])  array1 = array1.tolist()  **self**.range\_X.append(array1)    arr = np.asarray(**self**.position\_Y)  array2 = np.percentile(arr, [0, 25, 50, 75, 100])  array2 = array2.tolist()  **self**.range\_Y.append(array2)    arr = np.asarray(**self**.distance)  array3 = np.percentile(arr, [0, 25, 50, 75, 100])  array3 = array3.tolist()  **self**.distances.append(array3)    arr = np.asarray(**self**.angle)  array4 = np.percentile(arr, [0, 25, 50, 75, 100])  array4 = array4.tolist()  **self**.angles.append(array4)  ﻿ ﻿ **def** **return\_x\_y**(**self**, data):    x0 = [i[0] **for** i **in** data]  x0\_min = statistics.median\_low(x0)  x0\_mean = statistics.mean(x0)  x0\_max = statistics.median\_high(x0)    x25 = [i[1] **for** i **in** data]  x25\_min = statistics.median\_low(x25)  x25\_mean = statistics.mean(x25)  x25\_max = statistics.median\_high(x25)    x50 = [i[2] **for** i **in** data]  x50\_min = statistics.median\_low(x50)  x50\_mean = statistics.mean(x50)  x50\_max = statistics.median\_high(x50)    x75 = [i[3] **for** i **in** data]  x75\_min = statistics.median\_low(x75)  x75\_mean = statistics.mean(x75)  x75\_max = statistics.median\_high(x75)    x100 = [i[4] **for** i **in** data]  x100\_min = statistics.median\_low(x100)  x100\_mean = statistics.mean(x100)  x100\_max = statistics.median\_high(x100)   X = [max(x100), x100\_max, x100\_mean, x100\_min, min(x100),  max(x75), x75\_max, x75\_mean, x75\_min, min(x75),  max(x50), x50\_max, x50\_mean, x50\_min, min(x50),  max(x25), x25\_max, x25\_mean, x25\_min, min(x25),  max(x0), x0\_max, x0\_mean, x0\_min, min(x0)]    ﻿ **def** **update\_data**(**self**):  **self**.total\_time()  **self**.count\_reroutes()  **self**.optimal\_route = **self**.run.optimal\_delivey\_time  **self**.wait\_time = **self**.run.wait\_time   **self**.distance\_covered = **self**.run.move\_num  **self**.find\_velocity()  **self**.times.append(**self**.time)  **self**.reroutes.append(**self**.count\_reroute)  **self**.count\_disturbance()   # Returns time incured due to turns taken   **def** **count\_turns**(**self**):  count\_turns = 0   **for** i **in** range(len(**self**.path\_mapping)):  **if** **self**.path\_mapping[i].get("is\_vertical") != **self**.path\_mapping[i-1].get("is\_vertical"):  count\_turns += 1  **return** count\_turns/2   **def** **total\_time**(**self**):  count = **self**.count\_turns()  turns = count \* 2  wait\_time = **self**.run.wait\_time  total\_moves = **self**.distance\_covered  **self**.time = turns + total\_moves + wait\_time      # Returns the number of disturbances faced   **def** **count\_disturbance**(**self**):  count = 0   **for** i **in** **self**.path\_mapping:  **if** i.get("disturbance") == True:  count += 1  **self**.disturbances.append(count)  **self**.total\_disturbance = count      # Returns the number of reroutes faced   **def** **count\_reroutes**(**self**):  reroute = 0   **for** i **in** range(len(**self**.path\_mapping)):  **if** **self**.path\_mapping[i].get("reroute") != **self**.path\_mapping[i-1].get("reroute"):  reroute += 1    **self**.count\_reroute = reroute/2   **return** X |

|  |
| --- |
| **Appendix 12:** Collecting data for “Disturbance vs Time” and “Reroutes vs Time and Comparison of Wait-time Algorithms.” |
| ﻿ ﻿ **def** **update\_data**(**self**):  **self**.total\_time()  **self**.count\_reroutes()  **self**.optimal\_route = **self**.run.optimal\_delivey\_time  **self**.wait\_time = **self**.run.wait\_time   **self**.distance\_covered = **self**.run.move\_num  **self**.find\_velocity()  **self**.times.append(**self**.time)  **self**.reroutes.append(**self**.count\_reroute)  **self**.count\_disturbance()   # Returns time incured due to turns taken   **def** **count\_turns**(**self**):  count\_turns = 0   **for** i **in** range(len(**self**.path\_mapping)):  **if** **self**.path\_mapping[i].get("is\_vertical") != **self**.path\_mapping[i-1].get("is\_vertical"):  count\_turns += 1  **return** count\_turns/2   **def** **total\_time**(**self**):  count = **self**.count\_turns()  turns = count \* 2  wait\_time = **self**.run.wait\_time  total\_moves = **self**.distance\_covered  **self**.time = turns + total\_moves + wait\_time      # Returns the number of disturbances faced   **def** **count\_disturbance**(**self**):  count = 0   **for** i **in** **self**.path\_mapping:  **if** i.get("disturbance") == True:  count += 1  **self**.disturbances.append(count)  **self**.total\_disturbance = count      # Returns the number of reroutes faced   **def** **count\_reroutes**(**self**):  reroute = 0   **for** i **in** range(len(**self**.path\_mapping)):  **if** **self**.path\_mapping[i].get("reroute") != **self**.path\_mapping[i-1].get("reroute"):  reroute += 1    **self**.count\_reroute = reroute/2 |

**Reference List (APA Style)**

Grami, A. (2016). *Introduction to digital communications*. Amsterdam: Elsevier.

Fragkiadaki, Et. Al. (2015). Learning visual predictive models of physics for playing billiards. *arXiv preprint arXiv:1511.07404*.

Pfeiffer, Et. Al. (2016). From Perception to Decision: A Data-driven Approach to End-to-end Motion Planning for Autonomous Ground Robots. *arXiv e-prints*, arXiv-1609.

Sünderhauf, Et Al. (2018). The limits and potentials of deep learning for robotics. *The International Journal of Robotics Research*, *37*(4-5), 405-420.