

# Duckietown – Simulation und Visualisierung von Wegfindung

Janko Varga, Jan Berchtold, Animesh Sharma, Julian  
Rittweger, Jing Qian

Konstanz, 10.10.2020

## Teamprojekt

# Ehrenwörtliche Erklärung

Wir versichern hiermit an Eides Statt, dass die vorliegende Arbeit mit dem Titel “Duckietown – Simulation und Visualisierung Autonomer Wegfindung” selbstständig und ohne unzulässige fremde Hilfe erbracht wurde. Es wurden keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erklären wir hiermit, dass die schriftliche und die elektronische Form vollständig übereinstimmen.

Ich bin mir bewusst, dass eine falsche Erklärung rechtliche Folgen haben wird.

Konstanz, 7.10.20

---

Janko Varga, Jan Berchtold, Animesh Sharma, Julian Rittweger, Jing Qian

# Inhaltsverzeichnis

<b>Ehrenwörtliche Erklärung</b>	<b>1</b>
<b>Abbildungsverzeichnis</b>	<b>3</b>
<b>Tabellenverzeichnis</b>	<b>4</b>
<b>1 Einleitung</b>	<b>5</b>
<b>2 Duckie Town</b>	<b>6</b>
2.1 Gym-Duckietown . . . . .	6
2.2 Duckiebots . . . . .	7
<b>3 Projektarchitektur</b>	<b>8</b>
3.1 Übersicht . . . . .	9
<b>4 Installation</b>	<b>10</b>
<b>5 Lokalisierung</b>	<b>11</b>
5.1 Monte-Carlo Lokalisierung . . . . .	11
5.1.1 Initialisierung der Partikel . . . . .	12
5.2 Probleme und Problemlösungen . . . . .	19
<b>6 Steuerung</b>	<b>20</b>
6.1 Simulation integrierte Steuerung . . . . .	20
6.2 Steuerbefehl ermitteln . . . . .	21
<b>7 Planung und GUI</b>	<b>23</b>
7.1 Visualisierung . . . . .	23
7.1.1 TkInter . . . . .	23
7.1.2 Mapping und Koordination . . . . .	25
7.1.3 A*-Algorithmus . . . . .	28
7.1.4 Probleme und Problemlösungen . . . . .	31
<b>8 Fazit</b>	<b>33</b>
8.1 Zielvorgaben . . . . .	33
8.2 Rückblick und Verbesserungen . . . . .	33
<b>Literaturverzeichnis</b>	<b>34</b>
<b>Anhang</b>	<b>35</b>

# Abbildungsverzeichnis

2.1	Gym-Duckietown Simulator . . . . .	6
2.2	Komponenten und visuelle Repräsentation des Duckiebot . . . . .	7
2.3	Oberflächen der Duckietown-Umgebung . . . . .	7
3.1	Verwendete Technologien . . . . .	8
3.2	Projekt Übersicht . . . . .	9
5.1	Partikel Initialisierung . . . . .	13
5.2	Partikel zerstreuen sich . . . . .	14
5.3	Roulette rad . . . . .	16
5.4	Ende der Route, Monte Carlo nach mehreren Schritten . . . . .	18
7.1	GUI . . . . .	24
7.2	Coordinate system . . . . .	24
7.3	Canvas Anzeigeliste . . . . .	25
7.4	Rotate Duckie . . . . .	28
7.5	Gesucht ist der schnellste Weg von A nach B mit Ausnahme über die Rot markierten Felder . . . . .	29
7.6	Gewichtung der Kosten . . . . .	29
7.7	Beispiel für eine Zusammensetzung der Kosten . . . . .	29
7.8	Auswahl des günstigsten Knoten und Ermitteln neuer Nachbarschaft . . .	30
7.9	Bei mehreren Kandidaten wählt der Algorithmus den Kürzesten zu B aus	30
7.10	Die Aufweichung an der Kurve . . . . .	31
7.11	Ein gerades Stück Fahrbahn . . . . .	32

# Tabellenverzeichnis

2.1	Duckietown Umgebung. . . . .	6
7.1	Canvas widgets. . . . .	25
7.2	Messtabelle. . . . .	27

# 1 Einleitung

Dieser Bericht dokumentiert den Verlauf unseres Teamprojektes, damit verbundene Aufgaben- und Aufgabenstellungen sowie aufgetretene Probleme und deren Problemlösung. Dabei wird zuerst ein Einblick in das Fundament verwendeter Technologien und zusammenhängender theoretischer Komponenten gewährt um anschließend das Projekt in seinen einzelnen Komponenten (Steuerung, Lokalisierung, Planung und GUI) zu erläutern. In der Auswertung werden Erfahrungen, Erkenntnisse und Ausblick in einem Fazit gebündelt, sämtlicher mit dem Projekt verbundener Programmiercode findet sich im Anhang.

## 2 Duckie Town

Das Duckietown-Projekt ist eine vom Massachusetts Institute of Technology (MIT) entwickelte Umgebung und beschreibt ein berechenbares Ökosystem mit dem ermöglicht wird Forschungsansätze und -konzepte der Robotik als auch der Entwicklung Künstlicher Intelligenz rund um das Autonome Fahren testbar und verifizierbar zu machen. Dabei unterteilt sich das Duckietown-Projekt in die sogenannten “Duckiebots” (autonome Roboter) und die “Duckietown”, eine urbane Umgebung mit Strassen, Ampeln und Hindernissen, durch die die Bots navigieren müssen.

### 2.1 Gym-Duckietown

Gym-Duckietown bezeichnet einen virtuellen Simulator auf Basis von Python3 und OpenAI für das Duckietown-Universum und erlaubt sämtliche physische Gegebenheiten der realen Testumgebung in einer flexiblen und anpassbaren Simulation darzustellen um mit Hilfe dieser diverse Algorithmen rund um das Autonome Fahren, zu trainieren und zu testen. Dabei gibt es verschiedene pre-definierte Umgebungen welche alle über eine eigene Map-Datei als YAML-File geladen werden können.

Genutzte Umgebungen	Verfügbare Umgebung
4way, udem1	Straight-road, 4way, udem1, small-loop, small-loop-cw, zigzag-dist, loop-obstacles, loop-pedestrians

Tabelle 2.1: Duckietown Umgebung.



Abbildung 2.1: Gym-Duckietown Simulator

Ein via Gym-Duckietown realisierter Algorithmus kann problemlos über die sogenannte Domain-Randomization API auf reale Roboter übertragen werden.

## 2.2 Duckiebots

Duckiebots selbst sind minimale autonome Plattformen welche Duckies (dt. Gummienten) transportieren. Sämtliche Berechnungen werden on-board erledigt, eine Kamera und die dazugehörige Sensorik erlaubt die Erfassung der eigenen Umgebung. In Gym-Duckietown wird der Duckiebot mit Hilfe einer fixierten Kamera aus der Ego-Perspektive simuliert. Geschwindigkeit, Dreh-Winkel, Größe und weitere Variablen werden dabei im Verhältnis 1:1 von der Realität in die Simulation übersetzt.

Duckiebots nutzen zur Orientierung primär Computer-Vision um an Fahrspuren entlang zu navigieren, andere “Verkehrsteilnehmer” (Duckies) zu vermeiden und über Kreuzungen zu gelangen.

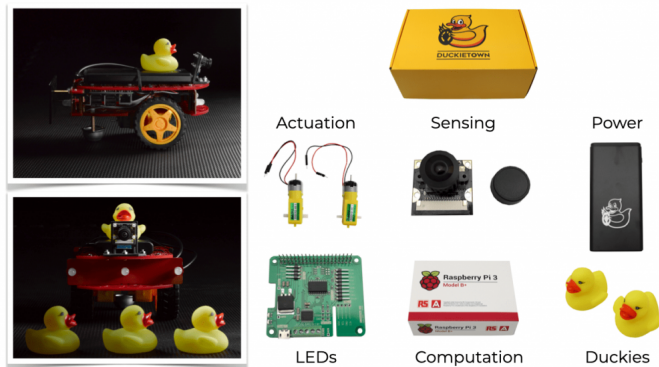


Abbildung 2.2: Komponenten und visuelle Repräsentation des Duckiebot

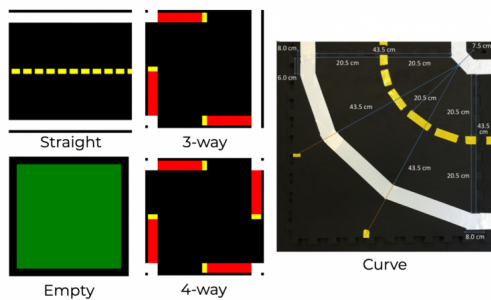


Abbildung 2.3: Oberflächen der Duckietown-Umgebung

Der Bot bewegt sich dabei auf einer von fünf möglichen Oberflächen: Straight, 3-way, 4-way, Curve und Empty. Die Oberfläche Empty kann dabei verschiedene Texturen annehmen (Grass, Asphalt, Haus) und darf vom Bot nicht befahren werden.



### 3 Projektarchitektur

Unsere Motivation für das Projekt war die Verbindung zum Thema Robotik und dem Autonomen Fahren. Eine grobe theoretische Idee der Realisierung war von Prof. Dr. Bittel bereits vorgeben. Damit lässt sich das Projekt in folgende Teilbereiche gliedern:

Bereich Beschreibung Perzeption Erzeugen von Input-Parametern anhand eines Neura-len Netzwerkes (nicht Bestandteil des Teamprojekts)

1. Lokalisierung Lokalisierung des Duckiebot auf der virtuellen Karte anhand des Monte-Carlo-Algorithmus 2. GUI und Planung Rendern einer Karte in einem 2D-Raum, Implementierung einer Wegfindung anhand des A\*-Algorithmus 3. Steuerung Aus der Wegfindung abgeleitete Steuerbefehle für den Duckiebot

Für die Kommunikation wurde Discord sowie TeamSpeak3 eingesetzt. Das Projektmanagement und auf die Gruppen (1-3) aufgeteilte Aufgaben wurden über die Plattform Trello formuliert und bearbeitet. Als Integrated Development Environment (IDE) wählte man das IntelliJ Pycharm IDE, für die Code-Versionierung und -Commitment kam Git in Verbindung mit Github zum Einsatz.

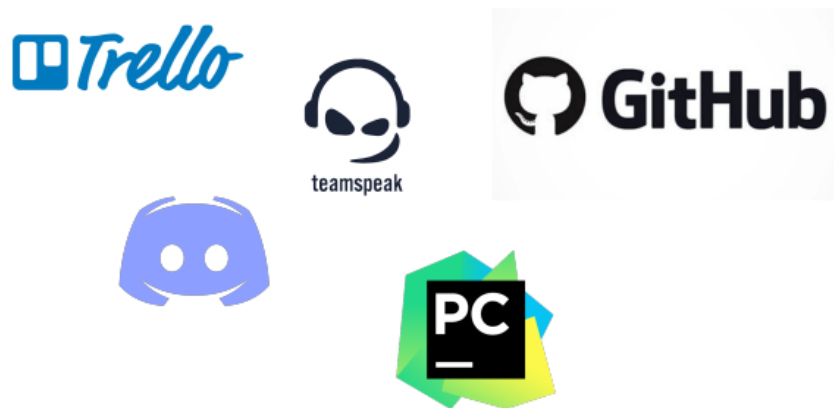


Abbildung 3.1: Verwendete Technologien

### 3.1 Übersicht

Ein Duckiebot als Bestandteil des Simulators liefert die Attribute: Geschwindigkeit, Winkel und Position (Z) an die Lokalisierung und in Form eines Bildes (über die eigene Kamera) an die Perzeption (Neurales Netzwerk). Die resultierenden Input-Parameter in Kombinationen mit den originalen Daten Z liefern Position und Winkel (Xg). Eine 2-dimensionale Karte erzeugt aus den Projektdateien und der Ausgabe der Monte-Carlo-Simulation ermöglicht die Planung von Strecken (A nach B) und die Berechnung einer schnellsten Strecke (von A nach B).

Die resultierende Strecke wiederum kann in Form ihrer physischen Attribute als Steuerungsvektor für den Duckiebot genutzt werden. Der Duckiebot navigiert so autonom auf Basis des Planungstools durch die Duckietown-Umgebung.

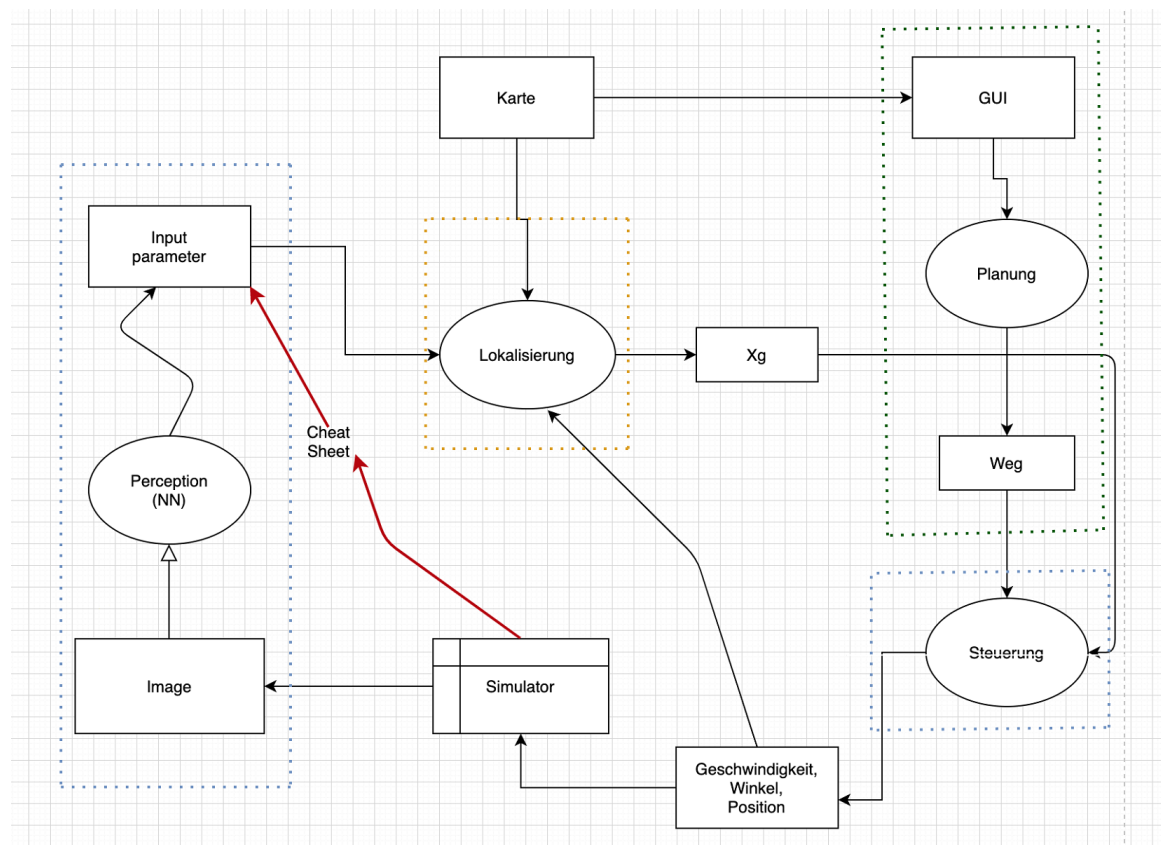


Abbildung 3.2: Projekt Übersicht

## 4 Installation

Zum Ausführen des Projekts benötigt man zunächst die gym-Duckietown selbst. Diese kann man von der offiziellen Duckietown-Projekt Github Repository herunterladen <https://github.com/duckietown/gym-duckietown>. Sämtliche Pythonpakete können mit dem folgendem Befehl installiert werden:

```
1 git clone https://github.com/duckietown/gym-duckietown.git
2 cd gym-duckietown
3 pip3 install -e .
```

Das Projekt selber kann man unter diesem Github-Link herunterladen.

```
1 git clone https://github.com/An571sha/GUI
```

Zum Ausführen muss man nun in dem GUI Repository und das Pythonskript gui.py ausführen

```
1 cd GUI
2 python gui.py
```

oder man verwendet eine IDE( Wir haben uns für PyCharm entschieden). Im GUI Repository befindet sich ebenfalls ein Video(GUIShowcase.mp4) in dem man die erstellte GUI in Aktion sehen kann.

## 5 Lokalisierung

Die Lokalisierung in der mobilen Robotik unterteilt sich in die globale- und lokale Selbstlokalisierung. Bei der globalen Selbstlokalisierung ist die initiale Position des Roboters unbekannt, der Roboter wird an einen beliebigen Ort auf einer Karte platziert und muss über seine Bewegung und die jeweiligen Sensorwerte herausfinden wo er sich auf der Karte befindet ohne zusätzliche Informationen über seine eigene Position zu kennen.

Bei der lokalen Selbstlokalisierung hingegen ist die initiale Position des Roboters schätzungsweise bekannt. Sobald sich der Roboter bewegt muss seine Position über die jeweiligen Sensorwerte neu berechnet werden. Im Folgenden wird die Implementierung der Selbstlokalisierung mit Hilfe des Monte-Carlo Algorithmus beschrieben.

### 5.1 Monte-Carlo Lokalisierung

Beim Verfahren der Monte-Carlo Lokalisierung werden beliebig viele Partikel die über die selbe Sensorik wie der Roboter verfügen, zufällig auf einer Karte erzeugt. Im Fall des Duckiebot liefert die Sensorik den Abstand und Winkel zu der jeweiligen Mittellinie. Neben diesen Werten besitzt jeder Partikel ein Gewicht, welches die wahrscheinliche Stellung des Roboters (Duckiebot) darstellt. Ist ein Gewicht sehr hoch, so repräsentiert dieser Partikel sehr wahrscheinlich die aktuelle Stellung des Roboters.

Bewegt sich nun der Roboter, erben die Partikel seine Bewegung und verhalten sich identisch zu diesen. Somit bewegen sich auch die Partikel, wenn sich der Roboter bewegt, nur an anderen zufälligen Positionen mit anderen zufälligen Winkeln. Endet die Bewegung des Roboters, werden die Sensorwerte des Roboters mit den Sensorwerten der Partikel verglichen und die Gewichte der Partikel neu berechnet. Partikel mit einem geringen Gewicht werden daraufhin über das Roulette-Verfahren gefiltert. Da ein lokales Lokalisierungsverfahren verwendet wird, erzeugen sich die Partikel in der näheren Umgebung des Roboters und versuchen dadurch die genaue Position herauszufinden. Dies haben wir anfangs über den Mittelwert der Position aller Partikel die in der jeweiligen Partikelmenge existieren erreicht. Jedoch haben wir einen noch genaueren Weg gefunden um die geschätzte Position zu berechnen, in dem wir die einzelnen Koordinaten mit einem Faktor multiplizieren. Dieser Faktor wird mittels einer Division zwischen dem jeweiligen eigenen Gewicht und dem insgesamten Gewicht aller Partikel berechnet. Somit verwenden wir kein arithmetischen Mittelwert, sondern einen gewichteten Mittelwert.

$$\vec{x} = \sum_{i=1}^n w_i x_i = w_1 x_1 + \dots + w_n x_n$$

### 5.1.1 Initialisierung der Partikel

```
1 def __init__(self, p_x, p_y, weight, name, map_, env, angle=-1):
2     self.p_x = p_x
3     self.p_y = p_y
4     self.tile: Tile = None
5     self.weight = weight
6     self.name = name
7     self.angle = angle
8     self.tilesizesize = 0.61
9     self.map_ = map_
10    self.env = env
```

Listing 5.1: Initialisierung der Partikel

“Tile” bezeichnet den aktuelle Feld-Typ der Karte, also wo sich momentan der Partikel befindet (ein Tile kann z.B. vom Typ “Gerade Strasse” sein). “weight” ist das Gewicht des Partikels, der Standard-Wert ist 1. “angle” beschreibt den globalen Winkel des Partikels. Der eigentliche Monte-Carlo Algorithmus wird in fünf Schritte aufgeteilt.

1. Über die Funktion `spawn_particle_list` werden beliebig viele Partikel in der Nähe des Roboters generiert

```
1 def spawn_particle_list(self, robot_pos, robot_angle):
2     self.p_list = list()
3     p_x, p_y = (robot_pos[0], robot_pos[2])
4     i = 0
5     while i < self.p_number:
6         randX = random.uniform(p_x - 0.25, p_x + 0.25)
7         randY = random.uniform(p_y - 0.25, p_y + 0.25)
8         rand_angle = random.uniform(robot_angle - np.deg2rad(15),
9                                     robot_angle + np.deg2rad(15))
10        a_particle = Particle(randX, randY, 1, i, self.map_, self.env, angle=
            rand_angle)
11        a_particle.set_tile()
12        if a_particle.tile.type not in ['floor', 'asphalt', 'grass']:
13            self.p_list.append(a_particle)
14        i = i + 1
```

Listing 5.2: `spawn_particle_list`

2. Nachdem die Liste an Partikeln erzeugt wurde, muss sichergestellt werden, dass sich die Partikel nur auf Flächen befinden die befahrbar sind (da sich der Roboter nur auf befahrbaren Boden befinden kann), deshalb werden diese Partikel mithilfe der “`filter_particle`” Funktion herausgefiltert, um Partikel auszuschließen die sich nicht auf einer Straße befinden. Partikel die sich auf Asphalt, Boden oder Gras befinden werden heraus gefiltert.

```
1 def filter_particles(self):
2     self.p_list = list(filter(lambda p: p.tile.type not in
```

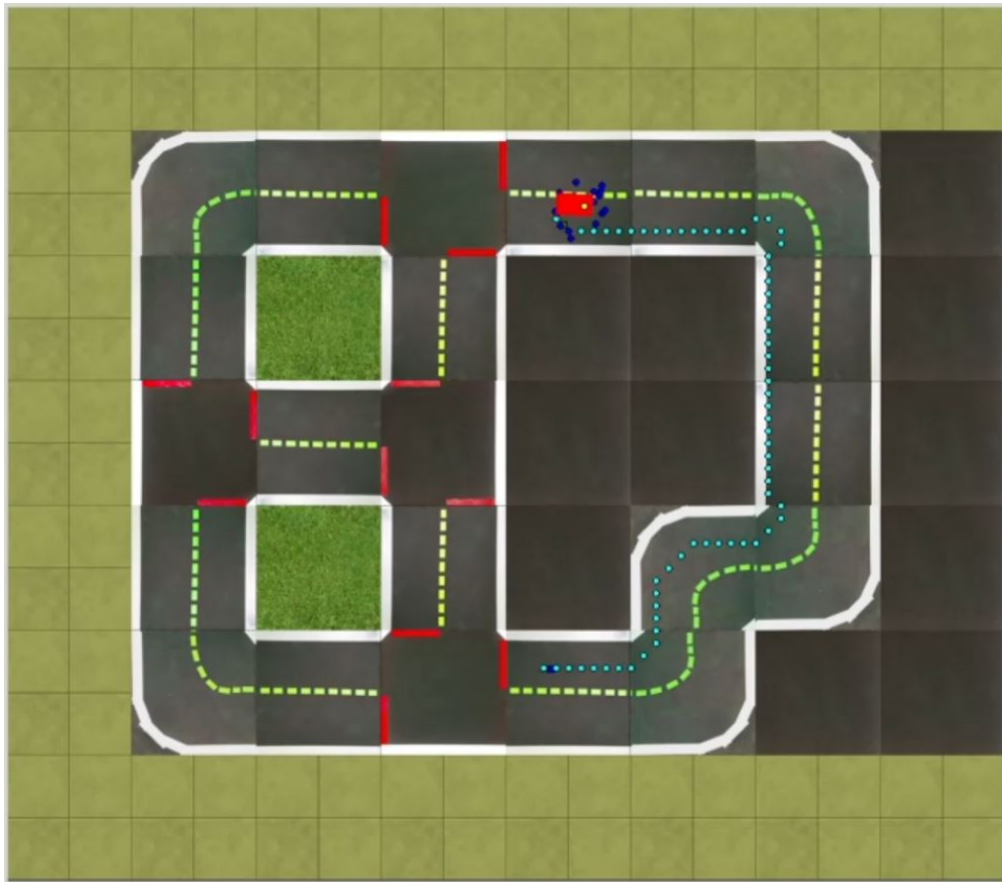


Abbildung 5.1: Partikel Initialisierung  
Partikel werden in der Nähe des Duckies initialisiert.  
Die geschätzte Pose ist als gelber Punkt markiert.

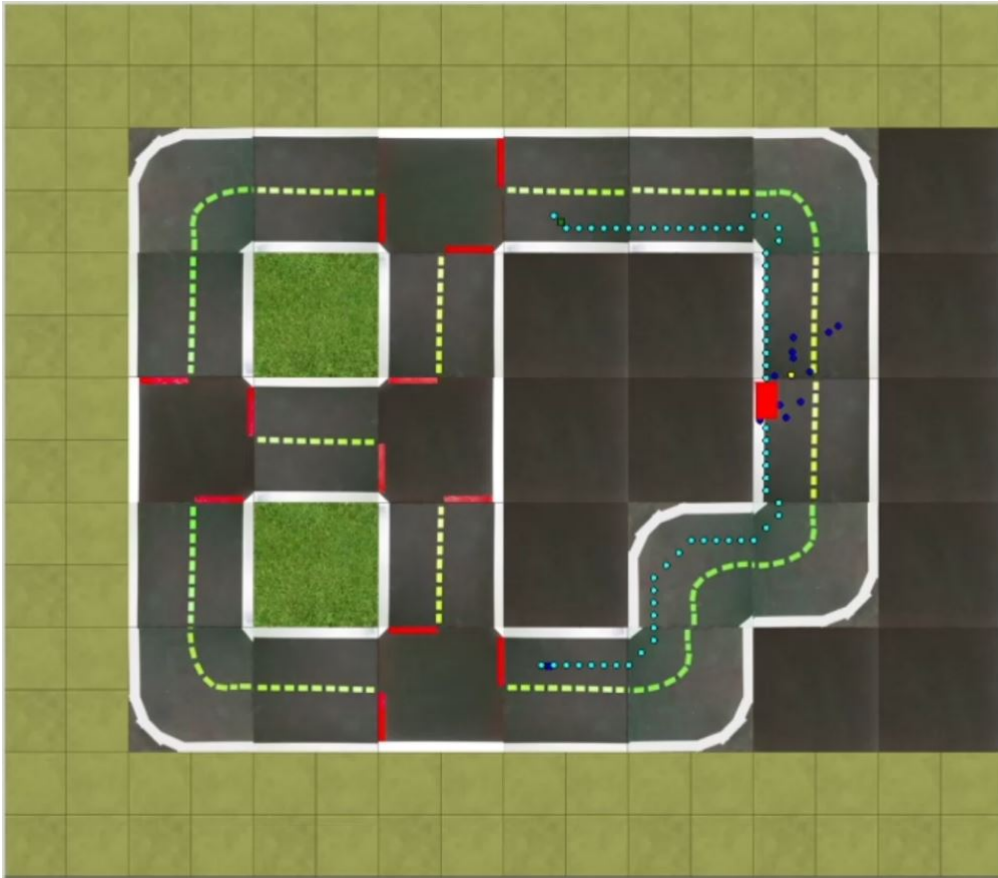


Abbildung 5.2: Partikel zerstreuen sich  
Die Partikel zerstreuen sich, aufgrund mehrerer möglicher Positionen.  
Die geschätzte Pose ist als gelber Punkt markiert.

```
3 ['floor', 'asphalt', 'grass'], self.p_list))
```

Listing 5.3: filter particles

3. Die Partikel befinden sich jetzt in der befahrbaren Umgebung um den Roboter. Wenn sich der Roboter bewegt wird diese als “integrate movement” bezeichnete Aktion an die Partikelmenge weitergegeben. In dieser Aktion bekommt jeder Partikel die Bewegung des Roboters übergeben, außerdem wird nach der Regung der einzelnen Partikel erneut die “filter\_particles” Funktion verwendet um Partikel die sich außerhalb der Straße bewegt haben zu entfernen.

```
1 def integrate_movement(self, action):
2 for p in self.p_list:
3 p.step(action)
4 self.filter_particles()
```

Listing 5.4: integrate movement

4. Nach jeder Bewegung des Roboters, sprich nach jedem Aufruf von “integrate\_movement”, muss das Gewicht jedes einzelnen Partikels neu berechnet werden. Diesen Vorgang bezeichnet man als “integrate\_measurement”.

```
1 def integrate_measurement(self, distance_duckie, angle_duckie):
2 for p in self.p_list:
3 p.weight_calculator(distance_duckie, angle_duckie)
```

Listing 5.5: integrate measurement

Das Gewicht eines Partikels ergibt sich durch die Differenz zwischen den Daten des Abstand-Sensors und denen des Partikels. Sowohl der Differenz zwischen dem Winkel des Roboters und dem Winkel des Partikels. Diese beiden Werte werden dann jeweils mittels der Gaußschen Glockenfunktion als Annäherung der aktuellen Position des Roboters.

Beide Wahrscheinlichkeiten werden zusammen multipliziert und ergeben letztendlich das Gewicht des Partikels. Dieses Gewicht wird daraufhin mit dem vorherigen Gewichten multipliziert um die vergangenen Gewichte/Schritte des Partikels mit in die Positionsschätzung einbinden zu können. Beim späteren Resampling werden die vergangenen, gespeicherten Schritte immer wieder zurückgesetzt.

```
1
2 def weight_calculator(self, distance, angle):
3 cur_pos = [self.p_x, 0, self.p_y]
4 lane_pos = self.env.get_lane_pos2(cur_pos, self.angle)
5 distance_p = lane_pos.dist
6 angle_p = lane_pos.angle_rad
7 self.weight = self.weight * norm.pdf(distance - distance_p) * norm.
   pdf(angle - angle_p)
8 return self.weight
```

Listing 5.6: Gewicht-Rechner

Nachdem die Bewegung des Roboters als auch die Neugewichtung der Partikel durchgeführt wurde, muss entschieden werden welche Partikel eine plausible Stellung des Roboters repräsentieren.

```
1
2 def resampling(self):
3 arr_particles = []
4 for i in range(0, len(self.p_list)):
5 idx = self.roulette_rad()
6 arr_particles.append(self.p_list[idx])
7 sum_py = 0
8 sum_px = 0
9 sum_angle = 0
10 for x in arr_particles:
11 sum_px = sum_px + x.p_x
```



## Roulette-Rad-Verfahren

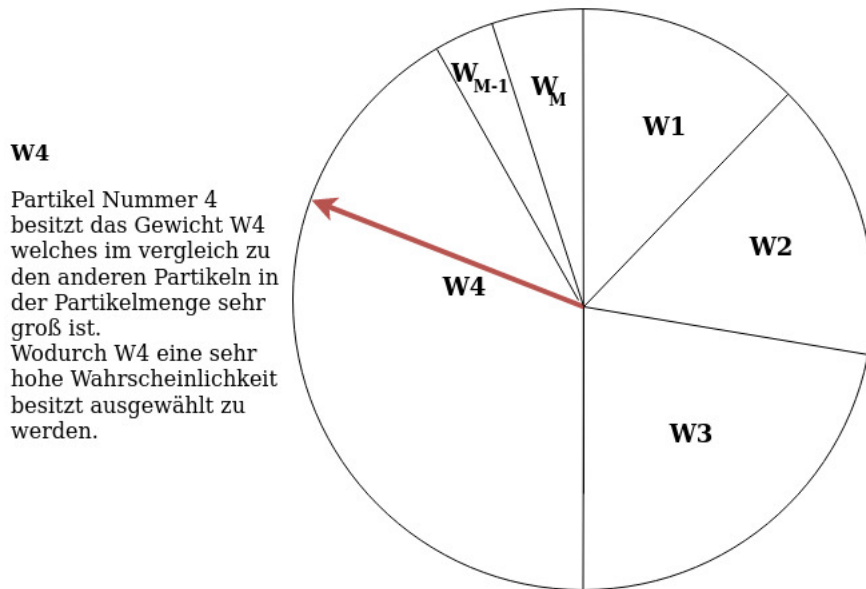


Abbildung 5.3: Roulette rad  
source by:draw.io

```

12 sum_py = sum_py + x.p_y
13 sum_angle += x.angle
14
15 possible_location = [sum_px / len(arr_particles), 0, sum_py / len(
    arr_particles)]
16 possible_angle = sum_angle / len(arr_particles)
17 return arr_particles, possible_location, possible_angle

```

Listing 5.7: Resampling

Um entscheiden zu können, welche Partikel als “chosen\_one” übrig bleiben und eine plausible Stellung des Roboters repräsentieren, wird das Roulette-Verfahren (“roulette\_rad”) verwendet. Dabei wird eine zufällige Zahl gewählt, die sich innerhalb der Menge der Summe aller Gewichte befindet um somit einem Partikel mit hohem Gewicht eine große Wahrscheinlichkeit zu ermöglichen, als einer der “chosen\_ones” gezogen zu werden. Dadurch existieren immer gleich viele Partikel nach dem Resampling.

```

1
2 def roulette_rad(self):
3     weight_arr = []
4     weight_of_particle = 0
5     for p in self.p_list:
6         weight_of_particle += p.weight
7     weight_arr.append(weight_of_particle)

```

```

8 the_chosen_one = random.uniform(0, weight_of_particle)
9 idx_particle = bisect_left(weight_arr, the_chosen_one)
10 return idx_particle

```

Listing 5.8: Roulette rad

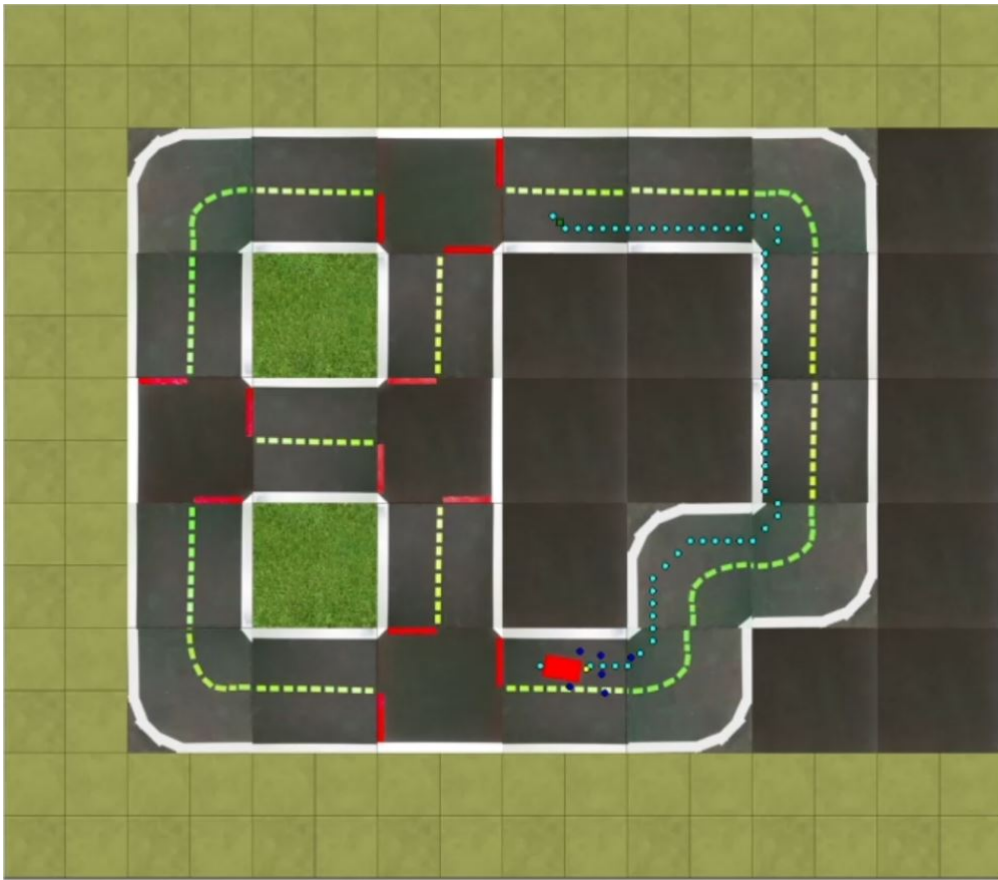
5. Um ein möglichst genaues Ergebnis zu erzielen, wird die Bewegung als auch die Berechnung der Gewichte zehnmal durchgeführt. Dabei werden vergangene Gewichte des jeweiligen Partikels miteinander multipliziert um auf Dauer ein präziseres Gewicht berechnen zu können. Nach jedem zehnten Schritt wird das Resampling durchlaufen und das multiplizierte Gewicht zurückgesetzt. Diese Verfahren zeigte sich als leistungseffizienter. Der “possible\_angle” (dt. Mögliche Winkel) und die “possible\_location” (dt. Mögliche Lokalisierung) des Roboters resultiert aus dem gewichteten Mittelwert aller Partikel und ihrer Positionen/Winkel. Wir haben die geschätzte Position des Duckies als gelben Punkt auf der GUI markiert.

```

1
2 obs, reward, done, info = env.step([speed, steering])
3 mcl.integrate_movement([speed, steering])
4 step_counter += 1
5 mcl.integrate_measurement(distance_to_road_center,
6                             angle_from_straight_in_rads)
7 if step_counter % 10 == 0:
8     arr_chosenones, possible_location, possible_angle = mcl.resampling()
9     print("posloc and robot position", possible_location, env.cur_pos)
10    print('possible angle and robot angle', possible_angle, env.
11           cur_angle)
12 mcl.weight_reset()

```

Listing 5.9: Reward



## 5.2 Probleme und Problemlösungen

- Performance

Anfangs wurde versucht jeden einzelnen Partikel als Klon des Duckiebots zu erzeugen, da allerdings min. einhundert bis gar tausende Partikel erzeugen werden müssen, kam es zu starken Performance-Einbußen.

- Dokumentation

Um die genauen Abmessungen der einzelnen Kartenbestandteile (z.B. einer Straße oder Kurve) herauszufinden, wurde die offizielle Dokumentation konsultiert. Hierbei kam es jedoch zu dem Problem, dass die Spezifikationen für den realen Duckiebot genauer sind und sich von den Spezifikationen der Simulation unterscheiden. Beispielsweise ist Kurve als reales Element eckig, in der Simulation allerdings rund.

- Kommunikationsschwierigkeiten

Das Projekt musste öfters umgebaut werden, weil es z.B. nicht gesichert war, wie bestimmte Daten in Verbindung mit der KI-Gruppe verarbeitet werden sollen.

# 6 Steuerung

## 6.1 Simulation integrierte Steuerung

Um einen Duckiebot in der Duckietown-Simulation zu bewegen, wird die Integrierte Simulationsmethode `step((Velocity, Angle))` verwendet. Die Simulation besitzt bereits die Ausmaßen des Duckiebots, wie z.B. den Abstand der Räder, um den Duckiebot in der Simulation genau so zu steuern, wie ein echter Duckiebot.

```
1
2  def step(self, action):
3      vel, angle = action
4
5      # Distance between the wheels
6      baseline = self.unwrapped.wheel_dist
7
8      # assuming same motor constants k for both motors
9      k_r = self.k
10     k_l = self.k
11
12     # adjusting k by gain and trim
13     k_r_inv = (self.gain + self.trim) / k_r
14     k_l_inv = (self.gain - self.trim) / k_l
15
16     omega_r = (vel + 0.5 * angle * baseline) / self.radius
17     omega_l = (vel - 0.5 * angle * baseline) / self.radius
18
19     # conversion from motor rotation rate to duty cycle
20     u_r = omega_r * k_r_inv
21     u_l = omega_l * k_l_inv
22
23     # limiting output to limit, which is 1.0 for the duckiebot
24     u_r_limited = max(min(u_r, self.limit), -self.limit)
25     u_l_limited = max(min(u_l, self.limit), -self.limit)
26
27     vels = np.array([u_l_limited, u_r_limited])
28
29     obs, reward, done, info = Simulator.step(self, vels)
```

Listing 6.1: Integrierte Steuerung

## 6.2 Steuerbefehl ermitteln

Für die Berechnung des Winkels für den Steuerbefehl, werden zwei verschiedene Methoden implementiert.

1. Lokale Fortbewegung: Bei der lokalen Fortbewegung sind keine Informationen der Position des Roboters erforderlich. Und wird auch nur zu Beginn der Simulation verwendet, falls keine globale Position von der Lokalisierung kommt. Für die Bestimmung des Winkels wird lediglich der Abstand des Duckiebots zur Mittellinie benötigt. Diese Information ist immer vorhanden. Selbst wenn der Duckiebot sich noch nicht bewegt hat. Angenommen der Duckiebot befindet sich auf einem Straßenelement. Zur Berechnung des zu fahrenden Winkels wird immer ein Punkt angesteuert der 5 cm orthogonal von der Mittellinie entfernt vor dem Duckiebot liegt. Diese Art der Steuerung sollte nur dann verwendet werden, wenn die Lokalisierung keine globale Position des Duckiebots bestimmen kann. Dieser Fall sollte nur zum Beginn der Simulation passieren.

```
1     def local_angle(rad, dist):
2         w = 10
3         tile_size = 1
4         return rad - (tile_size / 20 - dist) * w
```

Listing 6.2: Lokale Steuerung

2. Globale Fortbewegung: Die globale Fortbewegung dient dazu Koordinaten auf der Karte im Simulator anzufahren. Es kann ein einzelner Punkt oder eine Menge an Punkten übergeben werden. Mit dieser Methode wird der, vom A-Stern, berechneter Weg abgefahren. Die Punkte müssen sich auf der Fahrbahn befinden. Der Duckibot steuert den nächst gelegenen Punkt an bis zur einer Toleranz von 2 cm.

```
1     def global_angle_arr(self, point, i):
2         t = self.cur_angle
3         x = self.cur_pos[0]
4         y = self.cur_pos[2]
5         print("\n chster punkt", point[i][1] / 10, point[i][0] / 10)
6         rot = np.arctan2((y - point[i][0] / 10), (point[i][1] / 10 -
7             x))
8         if t is None:
9             t = 0
10        a = rot - t
11        a = (a / abs(a)) * (abs(a) % (2 * np.pi))
12        print(i, "t = ", t, "rot = ", rot, "a = ", a)
13        if (a >= np.pi):
14            a = -2 * np.pi + a
15            return a
16        if (a < -np.pi):
17            a = 2 * np.pi + a
18            return a
19        return a
```

Listing 6.3: Globale Steuerung

## 7 Planung und GUI

Der Aufgabenbereich im Teilbereich “Planung und GUI” des Projektes, war die Erstellung einer grafischen, 2-dimensionalen Oberfläche zur Visualisierung des Duckiebot und der Duckietown-Umgebung. Daneben sollte ein Wegfindungs-Werkzeug entwickelt werden, dass den schnellsten Weg sowie die aus der Monte-Carlo-Simulation gewonnen Partikel berechnet und visualisiert.

### 7.1 Visualisierung

Der Simulator und die damit verbundenen Komponenten funktionieren auf Basis der Programmiersprache Python, entsprechend entschied man sich auch bei der Wahl eines geeigneten Grafik-Frameworks für die Entwicklung der grafischen Oberfläche (GUI), für eine Python-Bibliothek.

Die Auswahlmöglichkeiten für ein GUI-Framework in der Python-Umgebung sind ziemlich dünn. Nach einiger Recherche fiel die Auswahl auf Pygame, eine Bibliothek die primär für Spiele-Programmierung genutzt wird. Der Grund: Die gewünschte GUI stellt nur ein sich bewegendes Element dar (den Duckiebot), die Karte ist statisch aufgebaut, Interaktion findet kaum statt und die Möglichkeit zu physikalischen Simulationen der Pygame-Library kann sich als nützlich erweisen. Die Wahl erwies sich leider als problematisch. Pygame führt unter Python3 zu Installationsproblemen und enthält keine Module für die Erzeugung von Navigationselementen wie Buttons und Menüs. Noch dazu erwies sich die Skalierung der Karte aufgrund mangelhafter Dokumentation des Grid-Modules als äußerst kompliziert und führte in unseren Render-Versuchen zu deren Performancebrüchen.

#### 7.1.1 TkInter

TkInter ist ein Standard-Interface für das Tk GUI Toolkit (ein Python-Framework für die Erstellung grafischer Oberflächen). Es ist kompatibel mit allen Plattformen (Windows, Linux, Mac-OS) und bietet die Möglichkeit zur Erzeugung von Canvas-Elementen als auch die Generierung statischer sowie dynamischer Buttons und Menü-Elementen. Diese Elemente werden in Form von sogenannten Widgets verwaltet die nach einer Kapsel-Logik beliebig miteinander verknüpft werden können. Zusätzlich bietet TkInter ein Koordinaten-System (welches sich als praktikabel für unsere Idee zur Erstellung der Map erwies), die Möglichkeit diverse Maus-Events abzufangen (zur Interaktion) sowie Methoden um Canvas-Elemente in beliebiger Form (Polygon, Oval, Rectangle) zu zeichnen.



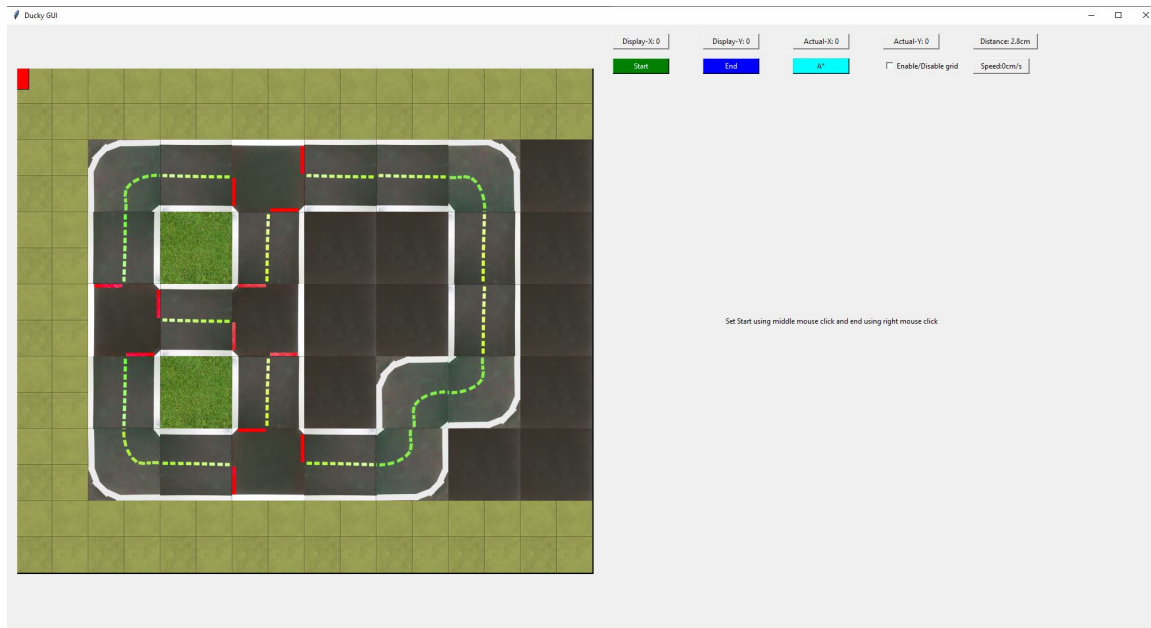


Abbildung 7.1: GUI

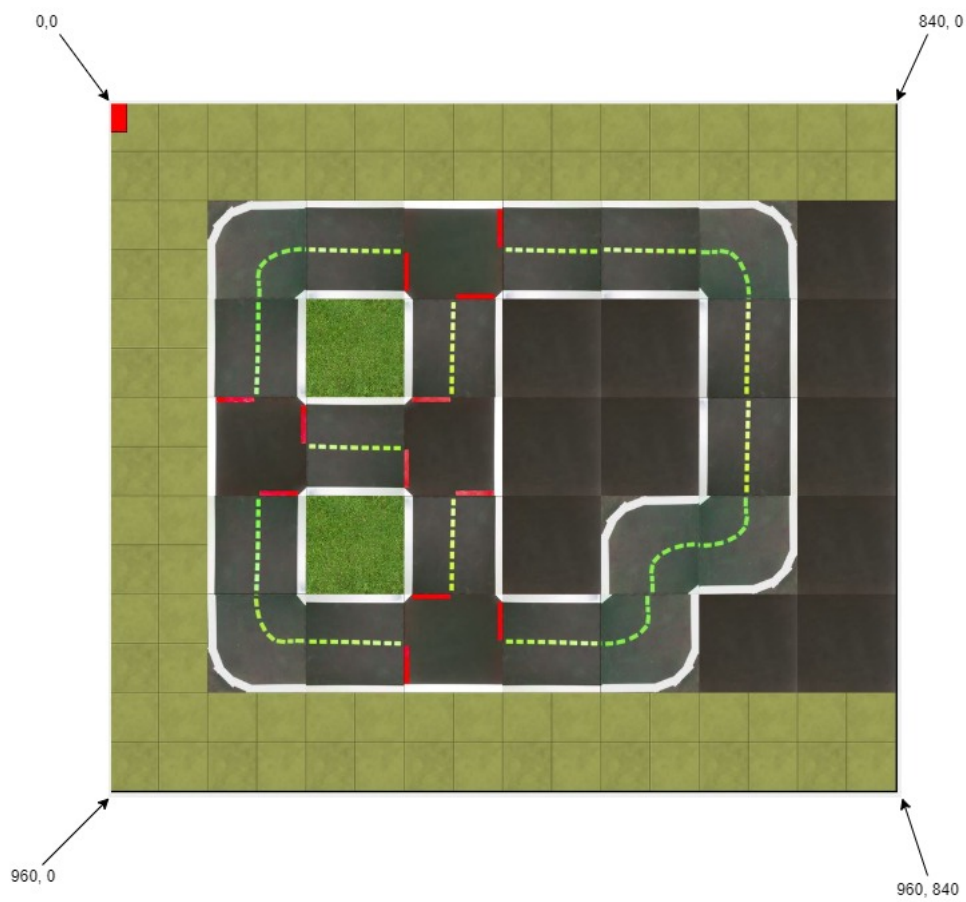


Abbildung 7.2: Coordinate system  
source by:draw.io

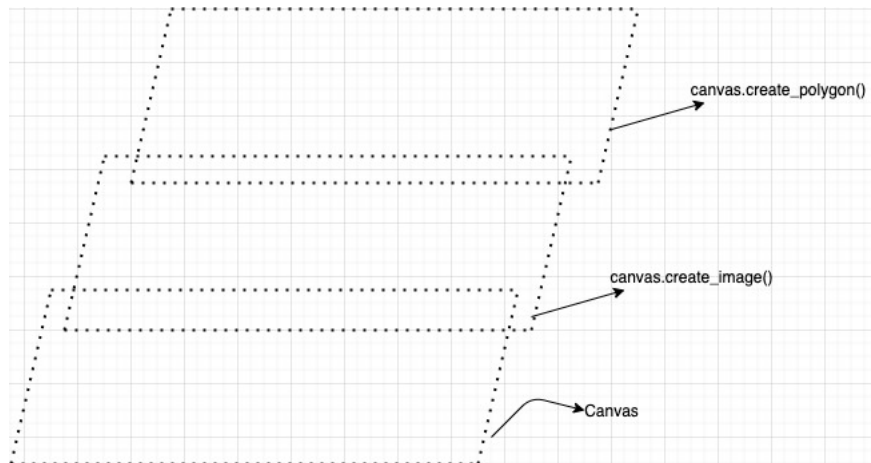


Abbildung 7.3: Canvas Anzeigeliste  
source by:draw.io

#### 7.1.1.1 Canvas

Der Canvas besitzt eine Anzeigeliste, die sich auf der Reihenfolge aller Objekte auf dem Canvas bezieht. Vom Hintergrund (unten in der Anzeigeliste) bis zum Vordergrund (Oben in der Anzeigeliste). Die Canvas-Elementen wie Duckie, das Grid und die Buttons werden in dieser Liste eingefügt. Zudem werden Canvas-Elementen mit einem Tag bezeichnet, damit man eine Referenz auf diesen Objekt hat.

Widget	Wo?	Wie?
Image	Anzeige der YAML-Tiles	canvas.create_image
Oval	Anzeige A*/Partikel	canvas.create_oval
Polygon/Rectangle	Anzeige Duckiebot	canvas.create_rectangle
Checkbox	Gitter an/aus	tk.Checkbutton(self.root, ..)
Button	Bedienung	tk.Button(self.root, ..)

Tabelle 7.1: Canvas widgets.

### 7.1.2 Mapping und Koordination

#### 7.1.2.1 YAML

Der grundlegende Aufbau sämtlicher virtueller Umgebungen wird im Gym-Duckietown über sogenannte YAML Dateien repräsentiert. YAML (Yet Another Markup Language) ist eine vereinfachte Auszeichnungssprache und dient zur Datenserialisierung ähnlich XML. Die Interaktion mit YAML Dateien erfolgt in Python3 über die Programmierbibliothek “yaml” (diese muss installiert werden und ist nicht Teil der Standard-Installation).

Beispiel einer assoziativen Liste als YAML Repräsentation:

```

--- # Block
name: John Smith
age: 33
--- # Inline
{name: John Smith, age: 33}

```

Karte “udem1.yml”:

```

tiles:
- [floor , floor , floor , floor , floor , floor , floor , floor]
- [floor , curve_left/W , straight/W, 3way_left/W , straight/W, straight/W ..
- [floor , straight/S , grass , straight/N , asphalt , asphalt , straight/N , asphalt]
- [floor , 3way_left/S , straight/W, 3way_left/N , asphalt , asphalt ..
- [floor , straight/S , grass , straight/N , asphalt , curve_right/N ..
- [floor , curve_left/S , straight/E, 3way_left/E , straight/E ..
- [floor , floor , floor , floor , floor , floor , floor , floor]

```

Da die Karten in Form von Kacheln (tiles) und deren Maßeinheit (10\*10) bereits als YAML Dateien existierten, mussten diese lediglich geladen und skaliert werden.

### 7.1.2.2 Skalierung

Jede einzelne Kachel wurde mit der Konstante 0,585 skaliert, einer frei gewählten globale Referenz für die Skalierung sämtlicher Karten-Abschnitte. Die Idee war: Jede der 10\*10 Kacheln um den Faktor 12 zu vergrößern, damit jeder Kartenabschnitt durch ein Bild der Größe 120x120 Pixel repräsentiert werden kann. Das Resultat ist eine Kachel-Größe von 58,5 cm.

Um eine visuelle und navigierbare Repräsentation der 3D-Umgebung zu erhalten wurde die Idee des folgenden Algorithmus umgesetzt:

- Die Karte in Form einer YAML Datei importieren und als Variable speichern
- Ein TkInter Canvas Widget der Größe 120x120 erzeugen
- Mit Hilfe der PIL Library (eine Python-Bibliothek für Bild-Manipulation) jeder Kachel das passende Bild im Canvas zu ordnen

Für eine symmetrische Karte und einheitlichen Kacheln mussten die vorliegenden und aus dem Duckietown-Projekt extrahierten Texturen manuell zu 256x256 Pixeln abgeändert werden.

Anschließend wurde ein Gitter (Grid) über den Canvas gezeichnet um die Kacheln in eine kleinere Dimension (6 Einheiten = 2.8 cm) zu unterteilen und so eine präzisere Wegfindung zu ermöglichen. Da die Größe des Duckiebot in der Spezifikation nicht angegeben ist, musste dieser gemessen und umgerechnet werden:

Wertetabelle
2,8cm = 6 Einheiten
21cm · 13cm ( <i>gemessen</i> )
1cm = 6/2,8 (2,14) Einheiten
21cm = 45 (2,14·21) <i>Einheiten</i>
13cm = 27 (2,14 ·13) <i>Einheiten</i>
45 · 27 <i>Einheiten</i>

Tabelle 7.2: Messtabelle.

Der Duckiebot selbst wurde als 36\*21 Einheiten( da, 45 \* 27 Einheiten der Duckie als zu groß dargestellt) in Form von roten Pixeln, als einzelnes Canvas in das Koordinatensystem gezeichnet und über die KeyPress-API der Tkinter-Bibliothek um x+n und y+n bewegbar gemacht (manuelle WSAD-Steuerung).

### 7.1.2.3 Bewegung des Duckiebot

Damit die Wegfindung des Duckiebot über die generierte 2D-Umgebung auch visuell korrekt dargestellt wird, der Duckiebot also bei Kurven oder vertikalen Richtungsänderungen seine optische Form verändert (Rotation des Canvas-Widgets um 45\*) musste neben dem globalen Koordinatensystem auch ein lokales, nur den Duckiebot betreffendes Koordinatensystem implementiert werden.

Dieses lokale Koordinatensystem wird repräsentiert durch einen Tupel, bestehend aus vier Punkten und einem Winkel, der sich je nach Bewegung neu generiert. Der Winkel berechnet sich mit Bezug auf die globalen Koordinaten, in dem der Arctan auf die alten Koordinaten in Differenz mit den neuen Koordinaten der Position des Duckiebots angewandt wird:

```
1 np.arctan2(ducky_pos[1] - old_ducky_pos[1], ducky_pos[0] - old_ducky_pos[0]);
```

Listing 7.1: rotate ducky

Ist der Winkel bestimmt, wird die neue Position des Duckiebot anhand von vier Koordinaten berechnet:

$$neue\_position_x = -alte\_position_x + ((\cosinus\_von\_Winkel * offset_x) \pm (\sinus\_von\_Winkel * \pm offset_y))$$

$$neue\_position_y = -alte\_position_y + ((\sinus\_von\_Winkel * \pm offset_x) \pm (\cosinus\_von\_Winkel * \pm offset_y))$$

Das Offset gibt in diesem Zusammenhang jeweils die Breite (x) und die Größe (y) des Duckiebot an. Anhand des neuen Wertepaares (x,y) der vier Koordinaten können dann neue Punkt bestimmt und diese über die Funktion “canvas.create\_polygon” als Duckiebot gerendert werden.

Dieser Prozess wird ad infinitum wiederholt um nach jedem Schritt (x+1) anhand der Koordinaten des Duckiebots dessen Position korrekt zu visualisieren.

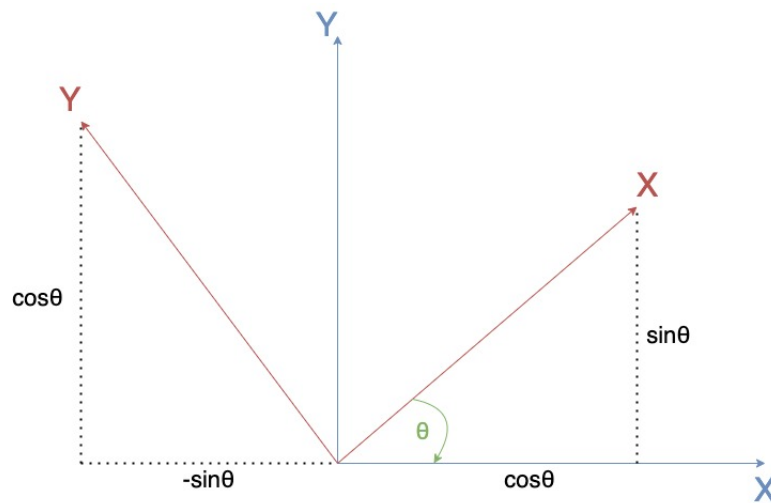


Abbildung 7.4: Rotate Duckie  
source by:draw.io

### 7.1.3 A\*-Algorithmus

Der A-Stern-Algorithmus ist ein Suchalgorithmus und wird in der Informatik verwendet um den kürzesten Weg zwischen zwei Knoten in einem Graphen zu berechnen. Er gilt als Erweiterung des Dijkstra-Algorithmus und verwendet eine Schätzfunktion (Heuristik) um zielgerichtet suchen zu können und die mit dem Suchvorgang verbundene Laufzeit zu verringern.

In unserem Projekt fand der A-Stern-Algorithmus Anwendung im Rahmen der grafischen Wegfindung des Duckiebot auf der jeweils ausgewählten Karte. Die Knoten werden dabei von einzelnen Kacheln (Ziel- und Endkachel) repräsentiert.

#### 7.1.3.1 Prinzip

Gegeben sei die oben dargestellte Situation. Es existieren zwei Knoten A, B, (Start- und Endknoten) sowie eine Folge an Hindernissen (rote Kacheln; “Gesperrt”) die im Rahmen der Navigation nicht betreten werden dürfen (in unserem Fall: Wiese-, Asphalt- und Sandkacheln).

Als erstes wird ermittelt wie weit entfernt die Knoten voneinander liegen. Man bedient sich dabei der Nachbarschaft und gewichtet diese ausgehend von A mit:

Diese Gewichtungen werden als G-Kosten bezeichnet mit  $G = \text{Distanz von Startknoten A}$ . Umgekehrt wird via Heuristik die Distanz vom Endknoten B zur jeweiligen Kachel (Knoten) berechnet, die sogenannten H-Kosten mit  $H = \text{Distanz vom Endknoten B}$ . Für die Heuristikfunktion wird die Luftlinie zwischen den zwei Knoten genommen.

Die beiden entstandenen Kosten werden als Summe in den sogenannten F-Kosten mit  $F = G + H$  zusammengefasst. Der Algorithmus geht also zum Startpunkt A und wählt abhängig von der gewichteten Nachbarschaft immer den Weg mit den günstigsten

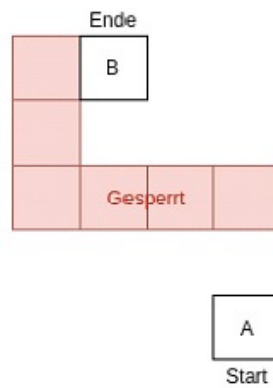


Abbildung 7.5: Gesucht ist der schnellste Weg von A nach B mit Ausnahme über die Rot markierten Felder

source by:draw.io

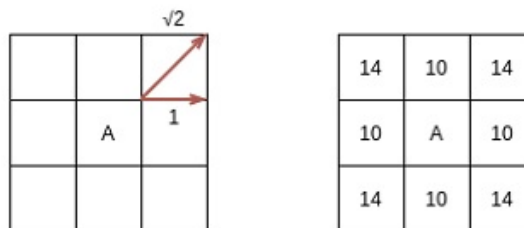
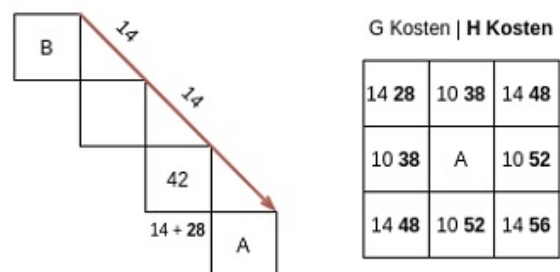


Abbildung 7.6: Gewichtung der Kosten

source by:draw.io



G Kosten   H Kosten					
14	28	10	38	14	48
10	38	A	10	52	
14	48	10	52	14	56

Abbildung 7.7: Beispiel für eine Zusammensetzung der Kosten

source by:draw.io

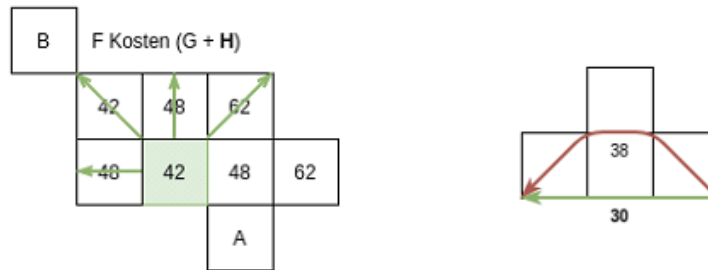


Abbildung 7.8: Auswahl des günstigsten Knoten und Ermitteln neuer Nachbarschaft  
source by:draw.io

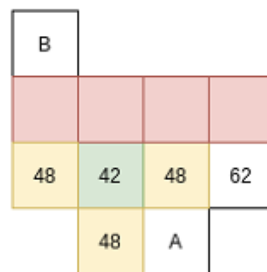


Abbildung 7.9: Bei mehreren Kandidaten wählt der Algorithmus den Kürzesten zu B aus

source by:draw.io

F-Kosten aus, in dem er diesen auswählt und anschließend erneut die Nachbarschaft ermittelt.

Dabei werden bereits ermittelte Nachbarschaften auf Basis neuer Erkenntnisse (in Form zusätzlich “gesichteter” Knoten), neu gewichtet. Es kann also durchaus vorkommen, dass sich ein ehemals teurerer Weg als günstig(er) herausstellt. Sollten mehrere Nachbarn das gleiche Gewicht (F) aufweisen, wählt der Algorithmus den Nachbarn mit den geringsten H-Kosten (Distanz zur Endkachel) aus.

### 7.1.4 Probleme und Problemlösungen

#### 1. Performance des A\*-Algorithmus

Die Berechnung des A\*-Algorithmus auf Basis von 1-dimensionalen Arrays (Nullen und Einsen, der jeweils nicht-betretbaren/betretbaren Felder) erzeugte starke Performance-Einbrüche. Das Problem konnte durch eine Repräsentation der Kacheln als 2-dimensionalen Array gelöst werden.

#### 2. Vermeidung von Kanten

Die Punkten an der Kanten werden bei der A Stern Algorithmus nicht berücksichtigt, da sobald in der Simulation der Duckiebot den Farhbahnrand erreicht, die Simulation anhält.



Abbildung 7.10: Die Aufweichung an der Kurve



3. Gerichteter Graphen Damit der Duckiebot nicht auf der falschen Fahrbahn fährt, dürfen die Knoten nur jeweils in eine Richtung zeigen und zwar in die Fahrtrichtung. Hierzu wird ein 2-dimensionales Array für jede Kachel angelegt, die bestimmt in welche Himmelsrichtung ein jeweiliger Knoten verbunden ist.

Ein Knoten 'N' ist mit Knoten 'W', 'E', 'N.W', 'N', und 'N.E' verbunden.

Ein Knoten 'S' ist mit Knoten 'W', 'E', 'S.W', 'S', und 'S.E' verbunden.

Ein Knoten 'W' ist mit Knoten 'W', 'S', 'N.W', 'N', und 'S.W' verbunden.

Ein Knoten 'E' ist mit Knoten 'S', 'E', 'N.E', 'N', und 'S.E' verbunden.

```

1      N.W   N   N.E
2      \    |   /
3      \    |   /
4      W----Knoten----E
5          / | \
6          / | \
7      S.W   S   S.E

```

Listing 7.2: gerichteter Graph

Ein gerades Stück Fahrbahn würde als gerichteter Graph wie folgt aussehen.

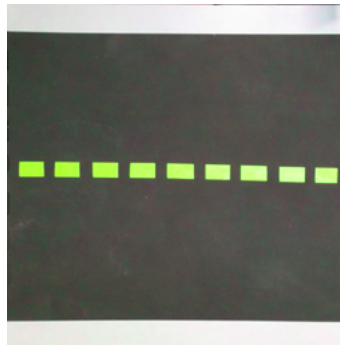


Abbildung 7.11: Ein gerades Stück Fahrbahn

```

1 straight_w =
2 [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
3 ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
4 ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
5 ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
6 ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
7 ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
8 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
9 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
10 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
11 ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]

```

Listing 7.3: Gerichteter Graph eines seitlichen Geraden Stück

# 8 Fazit

:

## 8.1 Zielvorgaben

1. Zwei-dimensionale Darstellung der Gym-Duckietown Umgebung in einer GUI
2. Lokalisierung des Duckiebot auf Basis einer Monte-Carlo-Simulation sowie eine damit verbundene Visualisierung der errechneten Partikel
3. Parallellisierung von Simulator und GUI, Implementierung eines Shortest-Path Algorithmus ( $A^*$ )

## 8.2 Rückblick und Verbesserungen

Mit unserer Projektarbeit glauben wir, alle geforderten Zielvorgaben erfolgreich umgesetzt zu haben. Die 2-dimensionale Darstellung (1) wurde mit Hilfe des TkInter-Framework realisiert und über interaktive Steuermöglichkeiten zu einem User-Interface ausgebaut. Dem Benutzer ist es möglich eine Route zwischen zwei Wegpunkten zu definieren, sowie Einsicht in diverse Informationen (Winkel, Position, Geschwindigkeit) den Duckiebot betreffend, zu erhalten. Potenzielle Verbesserungen wären z.B. eine variable Skalierung/Zoom-Funktion und die Möglichkeit mehrere Wegpunkte anzugeben. Die Monte-Carlo-Simulation (2) generiert eine variable Anzahl an Partikeln die sich korrekt um den Duckiebot verteilen und in Form von dunkelblauen Punkten dargestellt werden, aufgrund unterschiedlicher Sensorwerte verhält sich die Schätzung des Monte Carlos in Kurven genauer als auf einer geraden Straße. Hierbei würden weitere Sensorwerte, beispielsweise das miteinbeziehen von Straßenschildern oder die Entfernung zu einem in der Ferne liegenden Hindernisses helfen, besser die Unterschiedlichen möglichen Posen zu differenzieren. Die Parallellisierung von Simulator und GUI konnte durch die Implementierung konkurrenenter Module (Multi-Threading) erreicht werden. Das Projekt wurde auf der Distribution Ubuntu LTS 20.04 umgesetzt und getestet. Auf Windows 10 und Mac IOS gibt es Kompilationsprobleme mit tkinter und pygame. Die Umsetzung des  $A^*$ -Algorithmus war dabei die größte Herausforderung, vor allem in Verbindung mit dem Einhalten der korrekten Fahrspur.

Äußerst hilfreich war das Team-orientierte Arbeiten sowie die wöchentlichen Meetings mit Prof. Dr. Bittel, welche umfangreiche Möglichkeiten für Feedback und Verbesserungsvorschläge boten.

Insgesamt haben wir durch das Teamprojekt viel gelernt und erfolgreich unser, durch das Studium erworbene Fachwissen unter Beweis gestellt. Dafür bedanken wir uns!

# Literaturverzeichnis

- [Cormen, 2009] Cormen, T. H. (2009). *Introduction to Algorithms, third edition (Mit Press) (Englisch) Gebundene Ausgabe*.
- [Github, 2020] Github (2020). Gym-duckietown for openai gym.
- [gym duckietown, 2019] gym duckietown (2019). *Appearance Specifications*.
- [Python, ] Python. Graphical user interfaces with tk.
- [Senn, 2009] Senn, M. (2009). *Using L<sup>A</sup>T<sub>E</sub>X for Your Thesis*.
- [Swift, 2017] Swift, N. (2017). *Easy A\* (star) Pathfinding*.
- [tutorials point, 2020] tutorials point (2020). *Python - Multithreaded Programming*.

# Anhang

- gui.py

```
1  # The following class defines GUI.
2  import threading
3  import tkinter as tk
4  import numpy as np
5  from lokalisierung.duckietown_env import DuckietownEnv
6  from modules import astar, navigation, grid, mapping, findpath
7  import time
8  from Observer import Subscriber
9  from threading import Thread
10
11
12 class GUI(threading.Thread, Subscriber):
13
14     def __init__(self):
15         threading.Thread.__init__(self)
16         self.root = tk.Tk()
17         self.root.title('Ducky GUI')
18         self.root.geometry('1920x1080')
19         self.gboll = True
20         self.initialise_map = mapping.Mapping(tk)
21         self.canvas_and_ducky = self.initialise_map.gen_map('udem1')
22         self.canvas = self.canvas_and_ducky[0]
23         self.ducky = self.canvas_and_ducky[1]
24         self.generated_grid = grid.Grid(self.canvas)
25
26         self.number_of_clicks = 0
27         self.text_y_display = tk.StringVar()
28         self.text_x_display = tk.StringVar()
29         self.text_x_actual = tk.StringVar()
30         self.text_y_actual = tk.StringVar()
31         self.distance_traveled_display = tk.StringVar()
32         self.start_point = tk.StringVar()
33         self.end_point = tk.StringVar()
34         self.display_speed = tk.StringVar()
35         self.label_explain = tk.StringVar()
36
37         self.myduckietown = DuckietownEnv(GUI=self, domain_rand=
False, draw_bbox=False, map_name="udem1")
38         self.START = [0, 0]
39         self.END = [0, 0]
40         self.start = []
41         self.end = []
42         self.ovals = []
43
```

```

44         self.speed = 2
45         self.sleep_time = 0.2
46         self.counter = 0
47
48         self.start_set = False
49         self.end_set = False
50
51         self.line = astar.main(self.START, self.END)
52         self.path = findpath.Findpath(self.canvas)
53
54         self.coordinate_elements = {'display_x': 0, 'display_y': 0,
55                                     'actual_x': 0, 'actual_y': 0,
56                                     'distance_traveled': 0, '
57
58         number_clicks': 0}
59
60     def update(self, message):
61         print("Wird update auch während der simulation aufgerufen?"
62 )
63         self.draw_particles(message[0])
64         self.draw_ducky_bot(message[1], message[2])
65         self.draw_expected_pos(message[3])
66
67     def draw_particles(self, parti):
68         for x in range(0, len(parti)):
69             self.canvas.delete('parti' + str(x))
70
71             for x in range(0, len(parti)):
72                 print("Particle", x, "is=", parti[x].p_x, parti[x].p_y)
73                 self.canvas.create_oval(parti[x].p_x * 120, parti[x].p_y
74 * 120, parti[x].p_x * 120 + 6,
75                                     parti[x].p_y * 120 + 6, fill="
76 #0000ff",
77                                     tags='parti' + str(x))
78
79     def draw_expected_pos(self, pos):
80         self.canvas.delete('mean_pos_particle')
81         self.canvas.create_oval(pos[0] * 120, pos[2] * 120, pos[0] *
82 120 + 6, pos[2] * 120 + 6, fill="yellow",
83                                     tags='mean_pos_particle')
84
85     def set_gui_label_buttons(self):
86
87         """
88
89         :return:
90
91         """

```

```

87         tk.Button(self.root, textvariable=self.text_y_display, width
=12).grid(row=0, column=1, sticky='W', padx=15,
88
89             pady=15)
90         self.text_y_display.set('Display-X: 0')
91
92         tk.Button(self.root, textvariable=self.text_x_display, width
=12).grid(row=0, column=1, sticky='W', padx=165,
93
94             pady=15)
95         self.text_x_display.set('Display-Y: 0')
96
97         tk.Button(self.root, textvariable=self.text_x_actual, width
=12).grid(row=0, column=1, sticky='W', padx=315,
98
99             pady=15)
100        self.text_x_actual.set('Actual-X: 0')
101
102        tk.Button(self.root, textvariable=self.text_y_actual, width
=12).grid(row=0, column=1, sticky='W', padx=465,
103
104            pady=15)
105        self.text_y_actual.set('Actual-Y: 0')
106
107        tk.Button(self.root, textvariable=self.
distance_traveled_display, width=14).grid(row=0, column=1, sticky
='W',
108
109            padx=615,
110
111            pady=15)
112        self.distance_traveled_display.set('Distance: 0cm')
113
114        tk.Button(self.root, textvariable=self.display_speed, width
=12).grid(row=1, column=1,
115
116            sticky='NW',
117
118            padx=615,
119
120            pady=0)
121
122        self.label_explain.set("Set Start using middle mouse click
and end using right mouse click")
123
124        tk.Label(self.root, textvariable=self.label_explain).grid(
row=1, column=1,
125
126            sticky='W',

```

```

117     padx=200,
118
119     pady=0)
120
121     self.display_speed.set('Speed: ' + '0' + ' cm/s')
122
123     def keypress(self, event):
124
125         """
126         :param event:
127         :return:
128
129         """
130
131         x = 0
132         y = 0
133
134         if event.char == "a":
135             x = -6
136             display_x = self.coordinate_elements['display_x'] = self
.coordinate_elements['display_x'] - 1
137
138             self.text_x_display.set('Display-X: ' + str(display_x *
139 6))
140             self.text_x_actual.set('Actual-X: ' + str(display_x / 2)
141 )
142
143             print(display_x)
144
145         elif event.char == "d":
146             x = 6
147             display_x = self.coordinate_elements['display_x'] = self
.coordinate_elements['display_x'] + 1
148             self.text_x_display.set('Display-X: ' + str(display_x *
149 6))
150             self.text_x_actual.set('Actual-X: ' + str(display_x / 2)
151 )
152
153         elif event.char == "w":
154             y = -6
155             display_y = self.coordinate_elements['display_y'] = self
.coordinate_elements['display_y'] - 1
156             self.text_y_display.set('Display-Y: ' + str(display_y *
157 6))
158             self.text_y_actual.set('Actual-Y: ' + str(display_y / 2)
159 )

```

```

155         elif event.char == "s":
156             y = 6
157             display_y = self.coordinate_elements['display_y'] = self
.coordinate_elements['display_y'] + 1
158             self.text_y_display.set('Display-Y: ' + str(display_y *
159 6))
160             self.text_y_actual.set('Actual-Y: ' + str(display_y / 2)
161 )
162
163             distance_traveled = self.coordinate_elements['
164 distance_traveled'] = self.coordinate_elements[
165
166             'distance_traveled'] + 2.8
167             self.distance_traveled_display.set('Distance: ' + "{:.1f}".
168 format(distance_traveled) + 'cm')
169
170             self.canvas.move(self.ducky, x, y)
171
172         def set_start(self, event):
173
174             """
175
176             :param event:
177             :return:
178
179             """
180
181             if self.start_set:
182                 self.canvas.delete('start')
183
184             self.canvas.create_rectangle(event.x, event.y, event.x + 6,
185 event.y + 6, fill="green", tags="start")
186             self.start_set = True
187             x = int(round(event.x / 12))
188             y = int(round(event.y / 12))
189             self.start.append(x)
190             self.start.append(y)
191
192             self.start_point.set('{} , {}'.format(x, y))
193
194         def set_end(self, event):
195
196             """
197
198             :param event:
199             :return:
200
201             """

```



```

197         if self.end_set:
198             self.canvas.delete('end')
199
200             self.canvas.create_rectangle(event.x, event.y, event.x + 6,
201             event.y + 6, fill="blue", tags="end")
202             self.end_set = True
203             x = int(round(event.x / 12))
204             y = int(round(event.y / 12))
205             self.end.append(x)
206             self.end.append(y)
207
208             self.end_point.set('{} {}'.format(x, y))
209
210     def draw_ducky_bot(self, ducky_pos, ducky_angle):
211
212         """
213         :param ducky_pos:
214         :param ducky_angle:
215         :return:
216
217         """
218         ducky = []
219         ducky.append(ducky_pos[0] * 120)
220         ducky.append(ducky_pos[2] * 120)
221
222         print("ducky_pos in draw duckiebot", ducky_pos)
223
224         # angle = np.arctan2(ducky_pos[0], ducky_pos[1])
225         points = self.calc_points(ducky, ducky_angle - np.pi / 2)
226         self.canvas.delete("ducky")
227         self.ducky = self.canvas.create_polygon(points, fill="red",
228         tags='ducky')
229
230     def move_ducky(self, line):
231
232         """
233         :param line:
234         :return:
235
236         """
237
238         old_x = 0
239         old_y = 0
240
241         for coordinates in line:
242             # delete the ducky from _init_ and also after changing
243             position

```

```

243         self.canvas.delete("ducky")
244
245         y, x = coordinates
246         scaled_x = x * 12
247         scaled_y = y * 12
248         self.draw_ducky_bot([scaled_x, scaled_y], (old_x, old_y)
249     )
250
251     ducky_speed = (5.6 / self.sleep_time)
252
253     parti = []
254     for i in range(-25, 25):
255         parti.append((scaled_x + i, scaled_y + i))
256
257     # self.draw_particles(parti)
258
259     # print the coordinates of middle point of the ducky
260     # print("scaled_x = ", scaled_x, " scaled_y = ",
scaled_y)
261     self.text_x_display.set('Display-X: ' + str(scaled_x))
262     self.text_x_actual.set('Actual-X: ' + str(scaled_x / 12))
263 )
264     self.text_y_display.set('Display-Y: ' + str(scaled_y))
265     self.text_y_actual.set('Actual-Y: ' + str(scaled_y / 12))
266 )
267     self.display_speed.set('Speed: ' + "{:.1f}".format(
ducky_speed) + 'cm/s')
268     distance_traveled = self.coordinate_elements['
distance_traveled'] = self.coordinate_elements[
269         'distance_traveled'] + 5.6
270     self.distance_traveled_display.set('Distance: ' + "{:.1f}
".format(distance_traveled) + 'cm')
271
272     old_x = scaled_x
273     old_y = scaled_y
274
275     time.sleep(self.sleep_time)
276
277 def find_path(self):
278     """
279     :return:
280     """
281     if len(self.ovals) > 0:
282         for o in self.ovals:
283             self.canvas.delete(o)

```

```

284     line = []
285     line = astar.main(self.start, self.end)
286
287     if len(line) > 0:
288         for i, l in enumerate(line):
289             x = (4 + 4 + 4) * l[1] # 4 = 0,
290             y = (4 + 4 + 4) * l[0]
291
292             x1, y1 = (x - 3), (y - 3)
293             x2, y2 = (x + 3), (y + 3)
294             self.canvas.create_oval(x1, y1, x2, y2, fill="#00
ffff", tags='oval' + str(i))
295             self.ovals.append('oval' + str(i))
296             self.myduckietown.line = line
297             print('status of thread', self.myduckietown.is_alive())
298             if not self.myduckietown.is_alive():
299                 self.myduckietown.start()
300             print("wie viele threads laufen?", threading.active_count())
301             self.start = []
302             self.end = []
303
304     def draw_buttons(self):
305
306         """
307
308         :return:
309
310         """
311
312         tk.Button(self.root, textvariable=self.start_point, bg='
green', fg='white', width=12).grid(row=1, column=1,
313
314                                     sticky='NW',
315
316                                     padx=15,
317
318                                     pady=0)
319         self.start_point.set('Start')
320         tk.Button(self.root, textvariable=self.end_point, bg='blue',
321 fg='white', width=12).grid(row=1, column=1,
322
323                             sticky='NW',
324
325                             padx=165,
326
327                             pady=0)
328         self.end_point.set('End')

```

```

323         tk.Button(self.root, text='A*', width=12, command=self.
find_path, bg='#00ffff').grid(row=1, column=1,
324
325         sticky='NW',
326
327         padx=315,
328
329         pady=0)
330
331         tk.Checkbutton(self.root, text='Enable/Disable grid',
command=self.generated_grid.check_grid).grid(row=1,
332
333         column=1,
334
335         sticky='NW',
336
337         padx=465,
338
339         pady=0)
340
341     @staticmethod
342     def calc_points(pos, angle):
343
344         """
345         :param pos:
346         :param angle:
347         :return:
348
349         """
350         print("calc angle=", angle)
351         offx = 21 / 2
352         offy = 36 / 2
353
354         pointx1 = pos[0] + (np.cos(angle) * -offx - np.sin(angle) *
offy)
355         pointy1 = pos[1] - (np.sin(angle) * -offx + np.cos(angle) *
offy)
356
357         pointx2 = pos[0] + (np.cos(angle) * -offx - np.sin(angle) *
-offy)
358         pointy2 = pos[1] - (np.sin(angle) * -offx + np.cos(angle) *
-offy)
359
360         pointx3 = pos[0] + (np.cos(angle) * offx - np.sin(angle) * -
offy)
361         pointy3 = pos[1] - (np.sin(angle) * offx + np.cos(angle) * -
offy)

```

```

356     pointx4 = pos[0] + (np.cos(angle) * offx - np.sin(angle) *
357         pointy4 = pos[1] - (np.sin(angle) * offx + np.cos(angle) *
358             offy)
359
360         return [[pointx1, pointy1], [pointx2, pointy2], [pointx3,
361             pointy3], [pointx4, pointy4]]
362
363     def main(self):
364         self.set_gui_label_buttons()
365         self.draw_buttons()
366
367         self.root.bind("<Key>", self.keypress)
368         self.root.bind("<Button-2>", self.set_start)
369         self.root.bind("<Button-3>", self.set_end)
370
371         navigation.Navigation(self.canvas)
372         self.root.mainloop()
373
374 if __name__ == "__main__":
375     t = Thread(target=GUI().main(), args=())
376     t.start()
377

```

Listing 8.1: gui.py

- Observer.py

```

1  class Subscriber:
2      def __init__(self, name):
3          self.name = name
4
5      def update(self, message):
6          pass
7
8
9  class Publisher:
10     def __init__(self):
11         self.subscribers = set()
12
13     def register(self, who):
14         self.subscribers.add(who)
15
16     def unregister(self, who):
17         self.subscribers.discard(who)
18
19     def dispatch(self, message):
20         for subscriber in self.subscribers:

```

```

21 subscriber.update(message)
22
23

```

Listing 8.2: Observer.py

- astar.py

```

1  import numpy
2  import yaml
3
4  way_left_N = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
5                ['N', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
6                ['N', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
7                ['N', 'S', 'S', 'S', 'N', 'N', 'N', 'N', 'N', 'X'],
8                ['X', 'S', 'S', 'S', 'N', 'N', 'N', 'N', 'N', 'X'],
9                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
10               ['S', 'S', 'S', 'S', 'N', 'N', 'N', 'N', 'N', 'X'],
11               ['S', 'S', 'S', 'S', 'N', 'N', 'N', 'N', 'N', 'X'],
12               ['S', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
13               ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]
14
15  way_left_E = [['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'X', 'X'],
16                ['W', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'X', 'X'],
17                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
18                ['W', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'W', 'W'],
19                ['X', 'X', 'X', 'X', 'X', 'E', 'E', 'E', 'E', 'X'],
20                ['X', 'X', 'X', 'X', 'E', 'E', 'E', 'E', 'E', 'X'],
21                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
22                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
23                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
24                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]
25
26  way_left_S = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
27                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
28                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
29                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
30                ['X', 'S', 'S', 'S', 'X', 'X', 'X', 'X', 'X', 'X'],
31                ['X', 'S', 'S', 'S', 'S', 'S', 'N', 'N', 'N', 'N'],
32                ['X', 'S', 'S', 'S', 'S', 'S', 'N', 'N', 'N', 'S'],
33                ['X', 'S', 'S', 'S', 'S', 'S', 'N', 'N', 'N', 'S'],
34                ['X', 'S', 'S', 'S', 'S', 'S', 'N', 'N', 'N', 'X'],
35                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]
36
37  way_left_W = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
38                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
39                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
40                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
41                ['X', 'W', 'W', 'W', 'W', 'X', 'X', 'X', 'X', 'X'],
42                ['X', 'W', 'W', 'W', 'X', 'X', 'X', 'X', 'X', 'X'],

```

```

43         ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
44         ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
45         ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
46         ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X']]
47
48 curve_left_n = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
49                 ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'X'],
50                 ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'X'],
51                 ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'X'],
52                 ['X', 'X', 'X', 'X', 'X', 'X', 'N', 'N', 'N', 'X'],
53                 ['X', 'X', 'X', 'X', 'X', 'X', 'N', 'N', 'N', 'X'],
54                 ['S', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
55                 ['S', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
56                 ['X', 'X', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
57                 ['X', 'X', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]
58
59 curve_left_e = [['X', 'X', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X'],
60                 ['X', 'X', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X'],
61                 ['W', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X'],
62                 ['W', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X'],
63                 ['X', 'X', 'X', 'X', 'X', 'E', 'E', 'E', 'E', 'X'],
64                 ['X', 'X', 'X', 'X', 'E', 'E', 'E', 'E', 'E', 'X'],
65                 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'X'],
66                 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'X'],
67                 ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'X'],
68                 ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]
69
70 curve_left_s = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'X', 'X'],
71                 ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'X', 'X'],
72                 ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
73                 ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
74                 ['X', 'S', 'S', 'S', 'X', 'X', 'X', 'X', 'X', 'X'],
75                 ['X', 'S', 'S', 'S', 'S', 'S', 'X', 'X', 'X', 'X'],
76                 ['X', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S'],
77                 ['X', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S'],
78                 ['X', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S'],
79                 ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]
80
81 curve_left_w = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
82                 ['X', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
83                 ['X', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
84                 ['X', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
85                 ['X', 'W', 'W', 'W', 'W', 'X', 'X', 'X', 'X', 'X'],
86                 ['X', 'W', 'W', 'W', 'X', 'X', 'X', 'X', 'X', 'X'],
87                 ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
88                 ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
89                 ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'X', 'X'],
90                 ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'X', 'X']]
91

```

```

102 straight_w = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
103                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
104                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
105                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
106                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
107                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
108                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
109                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
110                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
111                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]
112
113 straight_n = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
114                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
115                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
116                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
117                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
118                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
119                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
120                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
121                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
122                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]
123
124 straight_s = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
125                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
126                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
127                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
128                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
129                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
130                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
131                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
132                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
133                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]
134
135 straight_e = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
136                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
137                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
138                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
139                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
140                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
141                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
142                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
143                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
144                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]
145
146 class Node:
147     def __init__(self, parent=None, position=None):
148         self.parent = parent
149         self.position = position

```



```

141         self.g = 0
142         self.h = 0
143         self.f = 0
144
145     def __eq__(self, other):
146         return self.position == other.position
147
148
149
150 class Vertex:
151     def __init__(self, position=None):
152         self.position = position
153
154     def getIndex(self):
155         return self.position[0] * 80 + self.position[1]
156
157     def __str__(self):
158         return str((self.position[0], self.position[1]))
159
160
161 def shortest(maze, start, end):
162     kl = []
163     d = []
164     p = []
165     number_of_vertices = (len(maze)) * (len(maze[0]))
166     print("number_of_vertices", number_of_vertices)
167     for x in range(0, number_of_vertices):
168         d.append(200000)
169         p.append(None)
170     d[start.getIndex()] = 0
171     kl.append(start)
172     print("start::", kl[0])
173
174     while len(kl) > 0:
175         v = find_min_h(kl, end, d)
176         kl.remove(v)
177         print("next vertex", v.position)
178         if v.getIndex() == end.getIndex():
179             print("find way")
180             return p
181         for x in get_adjacent_vertices(v, (len(maze) - 1), (len(maze)
182 [len(maze) - 1]) - 1), maze):
183             if d[x.getIndex()] == 200000:
184                 kl.append(x)
185                 cost = d[v.getIndex()] + 1 # maze[x.position[0]][x.
186 position[1]]
187                 if cost < d[x.getIndex()]:
188                     p[x.getIndex()] = v
189                     d[x.getIndex()] = cost

```

```

188     print("no way")
189     return p
190
191
192 def find_min_h(kl, end, d):
193     low = 20000000
194     v = None
195     for x in kl:
196         h = numpy.sqrt(((x.position[0] - end.position[0]) ** 2) + ((
197             x.position[1] - end.position[1]) ** 2))
198         if low > d[x.getIndex()] + h:
199             low = d[x.getIndex()] + h
200             v = x
201     return v
202
203 def get_adjacent_vertices(current_node, max_x, max_y, maze):
204     list_of_vertices = []
205
206     #
207     #
208     #   N.W   N   N.E
209     #   \    |   /
210     #   \    |   /
211     #   W----Cell----E
212     #   /  |  \
213     #   /  |  \
214     #   S.W   S   S.E
215
216     # for newPosition in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1)
217     # , (-1, 1), (1, -1), (1, 1)]:
218     adjacent_vertices = []
219     # (y,x)
220     print('maze value at y = ', current_node.position[0], 'x =',
221           current_node.position[1], 'value = ' , maze[current_node.position
222           [0]][current_node.position[1]])
223     if maze[current_node.position[0]][current_node.position[1]] == '
224     S':
225         adjacent_vertices = [(0, -1), (1, 0), (0, 1), (1, -1), (1,
226         1)]
227     elif maze[current_node.position[0]][current_node.position[1]] ==
228     'N':
229         adjacent_vertices = [(0, 1), (-1, 0), (0, -1), (-1, 1), (-1,
230         -1)]
231     elif maze[current_node.position[0]][current_node.position[1]] ==
232     'E':
233         adjacent_vertices = [(0, 1), (1, 0), (-1, 0), (1, 1), (-1,
234         1)]
235     elif maze[current_node.position[0]][current_node.position[1]] ==

```

```

227         'W':
228             adjacent_vertices = [(0, -1), (-1, 0), (1, 0), (-1, -1), (1,
229                                     -1)]
230
231     for newPosition in adjacent_vertices:
232         # for newPosition in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
233         node_position = (
234             current_node.position[0] + newPosition[0], # X
235             current_node.position[1] + newPosition[1]) # Y
236         v = Vertex(node_position)
237         # Range
238         range_criteria = [
239             node_position[0] > max_x,
240             node_position[0] < 0,
241             node_position[1] > max_y,
242             node_position[1] < 0,
243         ]
244
245         if any(range_criteria):
246             continue
247
248         # if maze[node_position[0]][node_position[1]] > 30:
249         #     continue
250         list_of_vertices.append(v)
251     return list_of_vertices
252
253 def create_path(list_of_vertices, start, end):
254     path = []
255     previous = end
256     print("number of element in p", len(list_of_vertices))
257     if list_of_vertices[end.getIndex()] is None:
258         return []
259     while start.getIndex() != previous.getIndex():
260         path.append(previous.position)
261         previous = list_of_vertices[previous.getIndex()]
262     path.append(start.position)
263     path.reverse()
264
265     return path
266
267 def load_map(name):
268     with open('maps/{}.yaml'.format(name), 'r') as map_file:
269         map_file = yaml.load(map_file, Loader=yaml.FullLoader)
270
271     return map_file['tiles']
272
273

```

```

274 x = load_map('udem1')
275
276
277 def create_map(mapdata):
278     print('start map')
279     data = []
280     mod = 1
281     for m in mapdata:
282         for i in range(0, 10):
283             row = []
284             for t in m:
285                 if t == 'floor' or t == 'asphalt' or t == 'grass':
286                     for o in range(10):
287                         row.append(200)
288             else:
289                 if t == 'straight/E':
290                     for l in straight_e[i]:
291                         row.append(l * mod)
292                 elif t == 'straight/W':
293                     for l in straight_w[i]:
294                         row.append(l * mod)
295                 elif t == 'straight/N':
296                     for l in straight_n[i]:
297                         row.append(l * mod)
298                 elif t == 'straight/S':
299                     for l in straight_s[i]:
300                         row.append(l * mod)
301                 elif t == 'curve_left/S':
302                     for l in curve_left_s[i]:
303                         row.append(l * mod)
304                 elif t == 'curve_left/N':
305                     for l in curve_left_n[i]:
306                         row.append(l * mod)
307                 elif t == 'curve_left/E':
308                     for l in curve_left_e[i]:
309                         row.append(l * mod)
310                 elif t == 'curve_left/W':
311                     for l in curve_left_w[i]:
312                         row.append(l * mod)
313                 elif t == '3way_left/E':
314                     for l in way_left_E[i]:
315                         row.append(l * mod)
316                 elif t == '3way_left/S':
317                     for l in way_left_S[i]:
318                         row.append(l * mod)
319                 elif t == '3way_left/W':
320                     for l in way_left_W[i]:
321                         row.append(l * mod)
322                 elif t == '3way_left/N':

```

```

323         for l in way_left_N[i]:
324             row.append(l * mod)
325         else:
326             for o in range(10):
327                 row.append(1 * mod)
328
329         print(row)
330         data.append(row)
331     return data
332
333
334 # star_map = create_areas(load_map('udem1'))
335
336
337 def main(start, end):
338     star_map = create_map(load_map('udem1'))
339
340     start_x = start[0]
341     start_y = start[1]
342
343     start = (start_y, start_x)
344
345     end_x = end[0]
346     end_y = end[1]
347
348     end = (end_y, end_x)
349
350     print('a star active')
351     print(star_map, start, end)
352
353     v = shortest(star_map, Vertex(start), Vertex(end))
354     print('v =', v)
355     s = Vertex(start)
356     e = Vertex(end)
357     if len(v) > 0:
358         p = create_path(v, s, e)
359         for x in p:
360             print(x)
361         print("length of my path", len(p))
362         return p
363     return []
364
365

```

Listing 8.3: astar.py

- findpath.py

```

1 class Findpath():
2     start_set = False

```

```

3     end_set = False
4     start = []
5     end = []
6     ovals = []
7
8     def __init__(self, canvas):
9         self.canvas = canvas
10
11     def set_start(self, event):
12         global start_set
13         global start_point
14         global start
15
16         if start_set:
17             self.canvas.delete('start')
18
19         self.canvas.create_rectangle(event.x, event.y, event.x + 6,
20                                     event.y + 6, fill="green", tags="start")
21         start_set = True
22         x = int(round(event.x / 12))
23         y = int(round(event.y / 12))
24         start.append(x)
25         start.append(y)
26
27         start_point.set('{} , {}'.format(x, y))
28

```

Listing 8.4: findpath.py

- grid.py

```

1     class Grid:
2         grid_items = []
3         grid_enabled = False
4
5         def __init__(self, canvas):
6             self.canvas = canvas
7             print(self.canvas)
8
9         def enable_grid(self):
10             canvas = self.canvas
11             # draw grid on grid
12             x1 = 0
13             x2 = 1000
14             # draw horizontal lines
15             for k in range(0, 1000, 6):
16                 y1 = k
17                 y2 = k

```

```

18         canvas.create_line(x1, y1, x2, y2, fill="#888888", tags=
'grid' + str(k))
19         self.grid_items.append('grid' + str(k))
20         # draw vertical lines
21         y1 = 0
22         y2 = 1000
23         for k in range(0, 1000, 6):
24             x1 = k
25             x2 = k
26             canvas.create_line(x1, y1, x2, y2, fill="#888888", tags=
'grid' + str(k))
27             self.grid_items.append('grid' + str(k))
28
29     def disabled_grid(self):
30         canvas = self.canvas
31
32         for g in self.grid_items:
33             canvas.delete(g)
34
35     def check_grid(self):
36         grid_enabled = self.grid_enabled
37
38         if grid_enabled:
39             self.disabled_grid()
40             self.grid_enabled = False
41         else:
42             self.enable_grid()
43             self.grid_enabled = True
44
45

```

Listing 8.5: grid.py

- mapping.py

```

1     import yaml
2     from PIL import ImageTk, Image
3
4
5     class Mapping:
6         one = []
7
8         def __init__(self, tk):
9             self.tk = tk
10
11         def gen_map(self, name):
12             global canvas
13             global ducky
14
15             with open('maps/{}.yaml'.format(name), 'r') as map_file:

```

```

16         imap = yaml.load(map_file, Loader=yaml.FullLoader)
17
18         dimension = self.calc_mapsize(imap)
19
20         try:
21             canvas.destroy()
22         except:
23             pass
24
25         canvas = self.tk.Canvas(width=dimension[0], height=dimension
26 [1], bg='black')
27         canvas.grid(row=1, column=0, padx=15, pady=15)
28
29         for row in range(len(imap['tiles'])):
30             for column in range(len(imap['tiles'][0])):
31                 image_file = self.get_image(imap['tiles'][row][
32 column])
33                 image = Image.open(image_file)
34                 resized_image = image.resize((120, 120), Image.
35 ANTIALIAS)
36                 self.one.append(ImageTk.PhotoImage(resized_image))
37                 canvas.create_image(column * 120 + 60, row * 120 +
38 60, image=self.one[-1])
39
40         x = 0
41         y = 0
42
43         ducky = canvas.create_rectangle(x, y, x + 21, y + 36, fill="
44 red", tags='ducky')
45         return [canvas, ducky]
46
47     def get_image(self, name):
48         name = name.replace('/', '_').lower()
49         try:
50             open('./ducky_pictures/' + name + '.png', 'r')
51             return './ducky_pictures/' + name + '.png'
52         except:
53             return './ducky_pictures/' + 'cub.png'
54
55     def calc_mapsize(self, imap):
56         height = len(imap['tiles']) * 120
57         width = len(imap['tiles'][0]) * 120
58
59         return [width, height]

```

Listing 8.6: mapping.py

- duckietown-env.py



```

1  # coding=utf-8
2  import threading
3
4  import numpy as np
5  from gym import spaces
6
7  from lokalisierung.Ducky_map import DuckieMap
8  from lokalisierung.MCL import MCL
9  from gym_duckietown import logger
10 from gym_duckietown.simulator import Simulator
11 from Observer import Publisher
12
13
14 class DuckietownEnv(Simulator, threading.Thread, Publisher):
15     """
16     Wrapper to control the simulator using velocity and steering
17     angle
18     instead of differential drive motor velocities
19     """
20     def __init__(
21         self,
22         gain=1.0,
23         trim=0.0,
24         radius=0.0318,
25         k=27.0,
26         limit=1.0,
27         line=None,
28         GUI=None,
29         **kwargs
30     ):
31         Publisher.__init__(self)
32         Simulator.__init__(self, **kwargs)
33         logger.info('using DuckietownEnv')
34         threading.Thread.__init__(self)
35         self.action_space = spaces.Box(
36             low=np.array([-1, -1]),
37             high=np.array([1, 1]),
38             dtype=np.float32
39         )
40
41         # Should be adjusted so that the effective speed of the
42         robot is 0.2 m/s
43         self.gain = gain
44
45         self.line = line
46
47         # Directional trim adjustment
48         self.trim = trim

```

```

48         self.register(GUI)
49
50
51         # Wheel radius
52         self.radius = radius
53
54         # Motor constant
55         self.k = k
56
57         # Wheel velocity limit
58         self.limit = limit
59
60     def global_angle_arr(self, point, i):
61         t = self.cur_angle
62         x = self.cur_pos[0]
63         y = self.cur_pos[2]
64         print("\n chster punkt", point[i][1] / 10, point[i][0] / 10)
65         rot = np.arctan2((y - point[i][0] / 10), (point[i][1] / 10 -
66 x))
67
68         if t is None:
69             t = 0
70         a = rot - t
71         a = (a / abs(a)) * (abs(a) % (2 * np.pi))
72         print(i, "t = ", t, "rot = ", rot, "a = ", a)
73         if (a >= np.pi):
74             a = -2 * np.pi + a
75             return a
76         if (a < -np.pi):
77             a = 2 * np.pi + a
78             return a
79         return a
80
81     def global_angle(self, point):
82         t = self.cur_angle
83         x = self.cur_pos[0]
84         y = self.cur_pos[2]
85         rot = np.arctan2((y - point[1]), (point[0] - x))
86         a = rot - t
87         a = (a / abs(a)) * (abs(a) % (2 * np.pi))
88         print(i, "t = ", t, "rot = ", rot, "a = ", a)
89         if (a >= np.pi):
90             a = -2 * np.pi + a
91             return a
92         if (a < -np.pi):
93             a = 2 * np.pi - a
94             return a
95         return a
96
97     def loca_angle(rad, dist):

```

```

96         w = 10
97         tile_size = 1
98         return rad - (tile_size / 20 - dist) * w
99
100     def run(self):
101         print("asd", self.line[0][1], self.line[0][0])
102         io = 0
103         self.cur_pos = [self.line[0][1] / 10, 0, self.line[0][0] /
104 10]
105
106         self.cur_angle = np.pi / 2
107
108         my_map = DuckieMap("maps/udem1.yaml")
109         particle_number = 25
110         mcl = MCL(particle_number, my_map, self)
111         mcl.spawn_particle_list(self.cur_pos, self.cur_angle)
112         step_counter = 0
113         newAStar = False
114         while True:
115             if io == len(self.line):
116                 newAStar = True
117                 continue
118             if newAStar:
119                 print('new aStar starts')
120                 self.cur_pos = [self.line[0][1] / 10, 0, self.line
121 0][0] / 10]
122                 self.cur_angle = np.pi / 2
123                 mcl.spawn_particle_list(self.cur_pos, self.cur_angle
124 )
125                 newAStar = False
126                 lane_pose = self.get_lane_pos2(self.cur_pos, self.
127 cur_angle)
128                 print("self.cur_pose =")
129                 print(self.cur_pos)
130                 print("\n")
131                 distance_to_road_center = lane_pose.dist
132                 angle_from_straight_in_rads = lane_pose.angle_rad
133                 print("dist = ", distance_to_road_center, " rad = ",
134 angle_from_straight_in_rads)
135
136                 speed = 0.2
137                 tol = 0.15
138                 if abs(self.cur_pos[0] - self.line[io][1] / 10) <= tol
139 and abs(self.cur_pos[2] - self.line[io][0] / 10) <= tol:
140                     io = io + 1
141                     if io == len(self.line):
142                         io = 0
143                         self.line = []
144                         print('length of line', len(self.line))
145                         continue

```

```

139         steering = self.global_angle_arr(self.line, io)
140         obs, reward, done, info = self.step([speed, steering])
141         mcl.integrate_movement([speed, steering])
142         step_counter += 1
143         mcl.integrate_measurement(distance_to_road_center,
angle_from_straight_in_rads)
144         if step_counter % 2 == 0:
145             # start = time.time()
146             arr_chosenones, possible_location, possible_angle =
mcl.resampling()
147             self.dispatch([arr_chosenones, self.cur_pos, self.
cur_angle])
148             # end = time.time()
149             # duration = end - start
150             print("posloc and robot position", possible_location
, self.cur_pos)
151             print('possible angle and robot angle',
possible_angle, self.cur_angle)
152             mcl.weight_reset()
153
154             # obs, reward, done, info = self.step([speed, loca_angle
(angle_from_straight_in_rads, distance_to_road_center)])
155             # obs, reward, done, info = self.step([speed, 0])
156             # total_reward += reward
157             print("info = ", info)
158             # print('Steps = %s, Timestep Reward=%.3f, Total Reward
=%.3f' % (self.step_count, reward, total_reward))
159
160             self.render()
161
162     def step(self, action):
163         vel, angle = action
164
165         # Distance between the wheels
166         baseline = self.unwrapped.wheel_dist
167
168         # assuming same motor constants k for both motors
169         k_r = self.k
170         k_l = self.k
171
172         # adjusting k by gain and trim
173         k_r_inv = (self.gain + self.trim) / k_r
174         k_l_inv = (self.gain - self.trim) / k_l
175
176         omega_r = (vel + 0.5 * angle * baseline) / self.radius
177         omega_l = (vel - 0.5 * angle * baseline) / self.radius
178
179         # conversion from motor rotation rate to duty cycle
180         u_r = omega_r * k_r_inv

```

```

181         u_l = omega_l * k_l_inv
182
183         # limiting output to limit, which is 1.0 for the duckiebot
184         u_r_limited = max(min(u_r, self.limit), -self.limit)
185         u_l_limited = max(min(u_l, self.limit), -self.limit)
186
187         vels = np.array([u_l_limited, u_r_limited])
188
189         obs, reward, done, info = Simulator.step(self, vels)
190         mine = {}
191         mine['k'] = self.k
192         mine['gain'] = self.gain
193         mine['train'] = self.trim
194         mine['radius'] = self.radius
195         mine['omega_r'] = omega_r
196         mine['omega_l'] = omega_l
197         info['DuckietownEnv'] = mine
198         return obs, reward, done, info
199
200
201 class DuckietownLF(DuckietownEnv):
202     """
203     Environment for the Duckietown lane following task with
204     and without obstacles (LF and LFV tasks)
205     """
206
207     def __init__(self, **kwargs):
208         DuckietownEnv.__init__(self, **kwargs)
209
210     def step(self, action):
211         obs, reward, done, info = DuckietownEnv.step(self, action)
212         return obs, reward, done, info
213
214
215 class DuckietownNav(DuckietownEnv):
216     """
217     Environment for the Duckietown navigation task (NAV)
218     """
219
220     def __init__(self, **kwargs):
221         self.goal_tile = None
222         DuckietownEnv.__init__(self, **kwargs)
223
224     def reset(self):
225         DuckietownNav.reset(self)
226
227         # Find the tile the agent starts on
228         start_tile_pos = self.get_grid_coords(self.cur_pos)
229         start_tile = self._get_tile(*start_tile_pos)

```

```

230
231     # Select a random goal tile to navigate to
232     assert len(self.drivable_tiles) > 1
233     while True:
234         tile_idx = self.np_random.randint(0, len(self.
drivable_tiles))
235         self.goal_tile = self.drivable_tiles[tile_idx]
236         if self.goal_tile is not start_tile:
237             break
238
239     def step(self, action):
240         obs, reward, done, info = DuckietownNav.step(self, action)
241
242         info['goal_tile'] = self.goal_tile
243
244         # TODO: add term to reward based on distance to goal?
245
246         cur_tile_coords = self.get_grid_coords(self.cur_pos)
247         cur_tile = self._get_tile(self.cur_tile_coords)
248
249         if cur_tile is self.goal_tile:
250             done = True
251             reward = 1000
252
253         return obs, reward, done, info
254
255

```

Listing 8.7: duckietown.env.py

- Ducky-map.py

```

1     import yaml
2     import random
3
4     ROW_MAP = 7.0
5     COLUMN_MAP = 8.0
6
7
8     class DuckieMap:
9
10        def __init__(self, yaml_file):
11            with open(yaml_file, "r") as stream:
12                try:
13                    out = yaml.safe_load(stream)
14                    tiles = out["tiles"]
15                    w = len(tiles[0])
16                    h = len(tiles)
17                    self.tiles = [[0 for x in range(h)] for y in range(w
)]

```

```

18         for i in range(h):
19             for j in range(w):
20                 self.tiles[j][i] = Tile(j, i, tiles[i][j])
21     except yaml.YAMLError as exc:
22         print(exc)
23
24     def search_tile(self, x, y):
25         return self.tiles[x][y]
26
27
28 class Particle:
29
30     def __init__(self, p_x, p_y, weight, name, angle=-1):
31         self.p_x = p_x
32         self.p_y = p_y
33         self.tile = None
34         self.weight = weight
35         self.name = name
36         self.angle = angle
37
38     def __repr__(self):
39         return 'Particle ' + str(self.name)
40
41     def set_tile(self, map):
42         self.tile = map.search_tile(int(self.p_x), int(self.p_y))
43
44     def distance_to_wall(self):
45         if self.tile.type == "straight/E" or self.tile.type == "
straight/W": # returns the distance to the closest wall (the
wall can be above or under the particle)
46             distance = self.p_y % 1
47             if distance >= 0.5:
48                 return 1 - distance - self.tile.WhiteTapeWidth
49             return distance - self.tile.WhiteTapeWidth
50         if self.tile.type == "straight/N" or self.tile.type == "
straight/S": # returns the distance to the closest wall (the
wall can be on the left or on the right)
51             distance = self.p_x % 1
52             if distance >= 0.5:
53                 return 1 - distance
54             return distance
55         if self.tile.type == "3way_left/S":
56             return self.p_x % 1
57         if self.tile.type == "3way_left/N":
58             return 1 - self.p_x % 1
59         if self.tile.type == "3way_left/W":
60             return self.p_y % 1
61         if self.tile.type == "3way_left/E":
62             return 1 - self.p_y % 1

```

```

63         if self.tile.type == "curve_left/S":
64             print("hi im a curve")
65
66
67 class Tile:
68
69     def __init__(self, x, y, tile_type):
70         self.x = x
71         self.y = y
72         self.type = tile_type
73         self.WhiteTapeWidth = 0.048
74
75     def __repr__(self):
76         return 'Index:(' + str(self.x) + ', ' + str(self.y) + ') ' +
77             'Type: ' + self.type
78
79     def index(self):
80         return self.x, self.y
81
82 def get_random_particles_list(count):
83     p_list = list()
84     i = 0
85     while i < count:
86         randX = random.uniform(0.0, 8.0)
87         randY = random.uniform(0.0, 7.0)
88
89         a_particle = Particle(randX, randY, 0, i)
90
91         p_list.append(a_particle)
92         i += 1
93     return p_list
94
95
96 def filter_particles(particle_list, map: DuckieMap):
97     for i in particle_list:
98         wert = map.search_tile(int(i.p_x), int(i.p_y)).type
99         if wert == "grass" or wert == "asphalt" or wert == "floor":
100             print(i)
101             particle_list.remove(i)
102     return particle_list
103
104
105 if __name__ == '__main__':
106     my_map = DuckieMap("../gym_duckietown/maps/udem1.yaml")
107     particle_list = get_random_particles_list(10)
108     for i in particle_list:
109         i.set_tile(my_map)
110     particle_list1 = filter_particles(particle_list, my_map)

```



```

111     for i in particle_list1:
112         print(f"particle name: {i.name} particle X: {i.p_x} particle
           y: {i.p_y} tile: {i.tile}")
113         print(i.distance_to_wall())
114
115

```

Listing 8.8: Ducky-map.py

- MCL.py

```

1     from lokalisierung.Particle import Particle
2     from lokalisierung.Ducky_map import DuckieMap
3     import random
4     from bisect import bisect_left
5     import numpy as np
6
7
8     class MCL:
9         def __init__(self, p_number, map, env):
10             self.p_number = p_number
11             self.map = map
12             self.p_list = None
13             self.env = env
14
15         def spawn_particle_list(self, robot_pos, robot_angle):
16             self.p_list = list()
17             p_x, p_y = (robot_pos[0], robot_pos[2])
18             i = 0
19             while i < self.p_number:
20                 randX = random.uniform(p_x - 0.25, p_x + 0.25)
21                 randY = random.uniform(p_y - 0.25, p_y + 0.25)
22                 rand_angle = random.uniform(robot_angle - np.deg2rad(15)
, robot_angle + np.deg2rad(15))
23                 a_particle = Particle(randX, randY, 1, i, self.map, self.
env, angle=rand_angle)
24                 a_particle.set_tile()
25                 if a_particle.tile.type not in ['floor', 'asphalt', '
grass']:
26                     self.p_list.append(a_particle)
27                     i = i + 1
28
29         def integrate_movement(self, action):
30             for p in self.p_list:
31                 p.step(action)
32             self.filter_particles()
33
34         def integrate_measurement(self, distance_duckie, angle_duckie):
35             for p in self.p_list:
36                 p.weight_calculator(distance_duckie, angle_duckie)

```

```

37
38     def resampling(self):
39         arr_particles = []
40         for i in range(0, self.p_number):
41             idx = self.roulette_rad()
42             arr_particles.append(self.p_list[idx])
43
44         sum_py = 0
45         sum_px = 0
46         sum_angle = 0
47         for x in arr_particles:
48             sum_px = sum_px + x.p_x
49             sum_py = sum_py + x.p_y
50             sum_angle += x.angle
51         # sum_px = functools.reduce(lambda a,b : a.p_x + b.p_x,
arr_chosenones)
52         # sum_py = functools.reduce(lambda a,b : a.p_y + b.p_y,
arr_chosenones)
53         possible_location = [sum_px / len(arr_particles), 0, sum_py
/ len(arr_particles)]
54         possible_angle = sum_angle / len(arr_particles)
55         return arr_particles, possible_location, possible_angle
56
57     def filter_particles(self):
58         self.p_list = list(filter(lambda p: p.tile.type not in ['
floor', 'asphalt', 'grass'], self.p_list))
59
60     def roulette_rad(self):
61         weight_arr = []
62         weight_of_particle = 0
63         for p in self.p_list:
64             weight_of_particle += p.weight
65             weight_arr.append(weight_of_particle)
66         the_chosen_one = random.uniform(0, weight_of_particle)
67         idx_particle = bisect_left(weight_arr, the_chosen_one)
68         return idx_particle
69
70     def weight_reset(self):
71         for p in self.p_list:
72             p.weight = 1
73
74
75 if __name__ == '__main__':
76     my_map = DuckieMap("../gym_duckietown/maps/udem1.yaml")
77     MCL = MCL(10, my_map)
78     MCL.spawn_particle_list()
79     print(MCL.p_list)
80     MCL.filter_particles()
81     print(MCL.p_list)

```

82  
83

Listing 8.9: MCL.py

- Particle.py

```
1  import numpy as np
2  from gym_duckietown.simulator import _update_pos
3  from lokalisierung import Tile
4  from gym_duckietown.simulator import WHEEL_DIST, DEFAULT_FRAMERATE,
    DEFAULT_ROBOT_SPEED
5  from scipy.stats import norm
6
7
8  class Particle:
9
10     def __init__(self, p_x, p_y, weight, name, map, env, angle=-1):
11         self.p_x = p_x
12         self.p_y = p_y
13         self.tile: Tile = None
14         self.weight = weight
15         self.name = name
16         self.angle = angle
17         self.tilesizesize = 0.61
18         self.map = map
19         self.env = env
20
21     def __repr__(self):
22         return 'Particle ' + str(self.name) + str(self.tile)
23
24     def set_tile(self):
25         self.tile = self.map.search_tile(int(self.p_x), int(self.p_y
26 ))
27
28     # todo: Jan should review this
29     def step(self, action: np.ndarray):
30         vel, angle = action
31         baseline = WHEEL_DIST
32
33         # assuming same motor constants k for both motors
34         k_r = 27.0
35         k_l = 27.0
36
37         # adjusting k by gain and trim
38         k_r_inv = (1.0 + 0) / k_r
39         k_l_inv = (1.0 - 0) / k_l
40
41         omega_r = (vel + 0.5 * angle * baseline) / 0.0318
42         omega_l = (vel - 0.5 * angle * baseline) / 0.0318
```

```

42
43     # conversion from motor rotation rate to duty cycle
44     u_r = omega_r * k_r_inv
45     u_l = omega_l * k_l_inv
46     u_r_limited = max(min(u_r, 1.0), -1.0)
47     u_l_limited = max(min(u_l, 1.0), -1.0)
48
49     vels = np.array([u_l_limited, u_r_limited])
50     vels = np.clip(vels, -1, 1)
51     # Actions could be a Python list
52     vels = np.array(vels)
53
54     cur_pos = [self.p_x, 0.0, self.p_y]
55     wheelVels = vels * DEFAULT_ROBOT_SPEED * 1
56     old_angle = self.angle
57     new_pos, cur_angle = _update_pos(cur_pos,
58                                     old_angle,
59                                     WHEEL_DIST,
60                                     wheelVels,
61                                     deltaTime=1.0 /
DEFAULT_FRAMERATE)
62
63     self.angle = cur_angle
64     self.p_x = new_pos[0]
65     self.p_y = new_pos[2]
66     self.set_tile()
67     return self.p_x, self.p_y, self.angle
68
69     def weight_calculator(self, distance, angle):
70         cur_pos = [self.p_x, 0, self.p_y]
71         lane_pos = self.env.get_lane_pos2(cur_pos, self.angle)
72         distance_p = lane_pos.dist
73         angle_p = lane_pos.angle_rad
74         self.weight = self.weight * norm.pdf(distance - distance_p)
75         * norm.pdf(angle - angle_p)
76         return self.weight

```

Listing 8.10: MCL.py

- Tile.py

```

1     class Tile:
2
3     def __init__(self, x, y, tile_type):
4         self.x = x
5         self.y = y
6         self.type = tile_type
7         self.WhiteTapeWidth = 0.048
8

```

```
9 def __repr__(self):
10     return 'Index(' + str(self.x) + ', ' + str(self.y) + ') ' +
    'Type: ' + self.type
11
12 def index(self):
13     return self.x, self.y
14
15
```

Listing 8.11: MCL.py