Hochschule Konstanz
Technik, Wirtschaft und Gestaltung

Teamprojekt

Duckietown – Simulation und Visualisierung von Wegfindung

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

Konstanz, 10.10.2020

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

Abbildungsverzeichnis

Tabellenverzeichnis

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 verschiedenene 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.

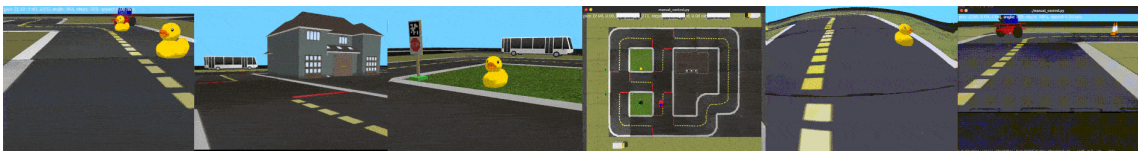

Abbildung 2.1: Gym-Duckietown Simulator

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 Variabeln 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.

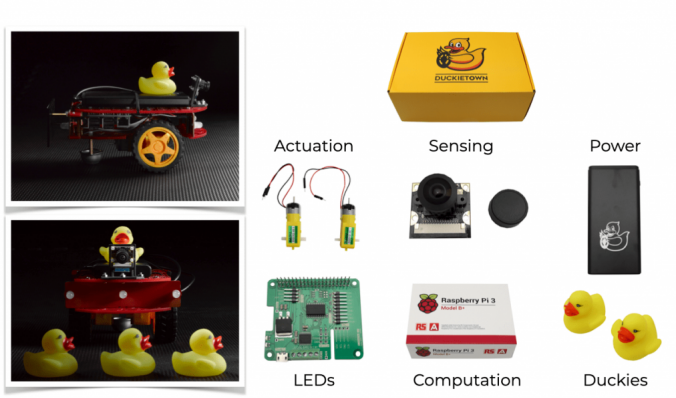

Abbildung 2.2: Komponenten und visuelle Repräsentation des Duckiebot

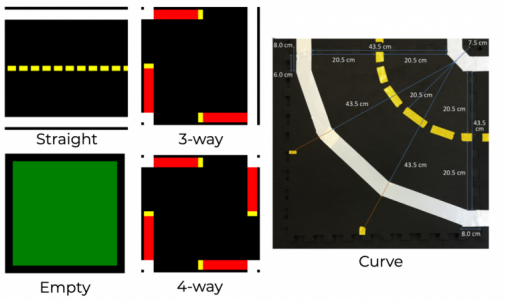

Abbildung 2.3: Oberflächen der Duckietown-Umgebung

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 Neuralen 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 wiederrum 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.

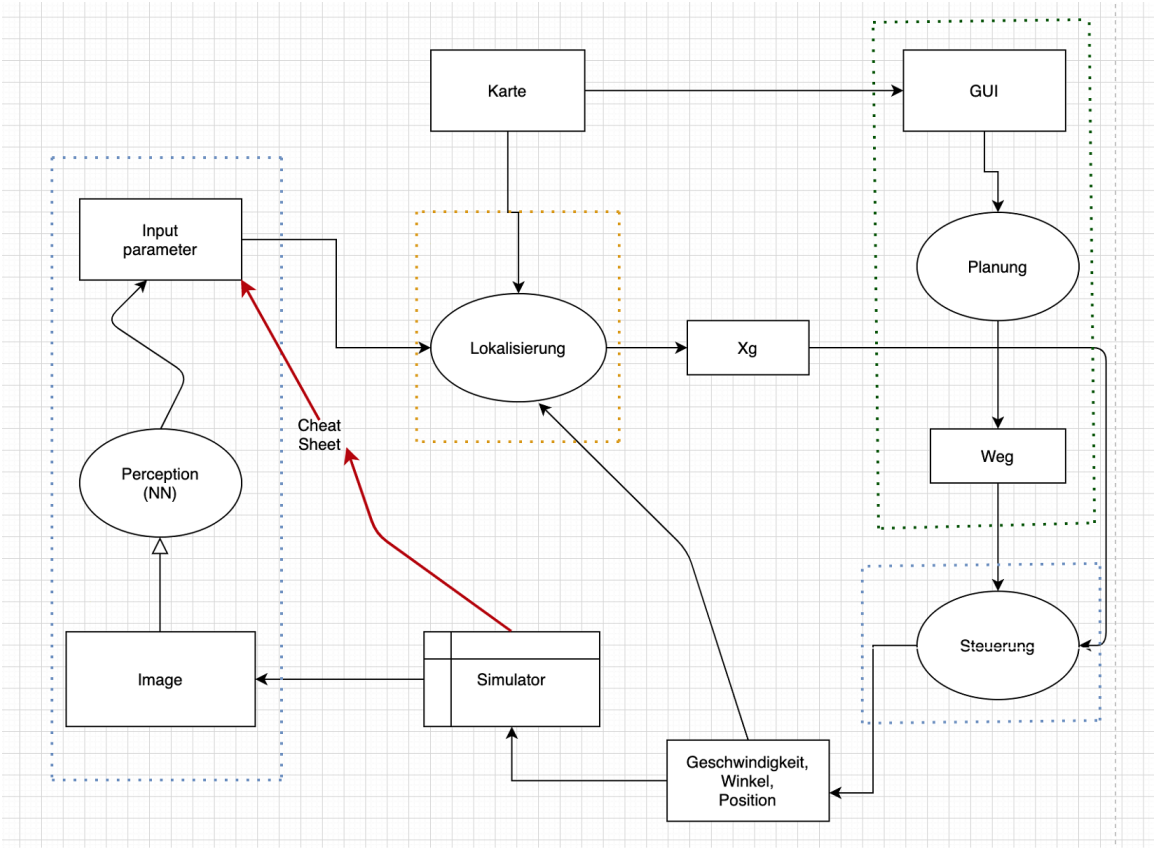

Abbildung 3.2: Projekt Übersicht

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 Pythonpackete können mit dem folgendem Befehl installiert werden:

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

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

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

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

```
cd GUI
python gui.py
```

oder man verwendet eine IDE(Wir haben uns für PyCharm entschieden). Es wird empfohlen eine Linux Distribution wie z.B. Ubuntu zu verwenden. Unter Windows bzw. iOS gibt es Kompatibilitätsproblem mit den Grafikbibliotheken tkinter und pyglet. 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.

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

5.1.1 Initialisierung der Partikel

```
def __init__(self, p_x, p_y, weight, name, map_, env, angle=-1):
self.p_x = p_x
self.p_y = p_y
self.tile: Tile = None
self.weight = weight
self.name = name
self.angle = angle
self.tilesize = 0.61
self.map_ = map_
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 generi

```
def spawn_particle_list(self, robot_pos, robot_angle):
self.p_list = list()
p_x, p_y = (robot_pos[0], robot_pos[2])
i = 0
while i < self.p_number:
randX = random.uniform(p_x - 0.25, p_x + 0.25)
randY = random.uniform(p_y - 0.25, p_y + 0.25)
rand_angle = random.uniform(robot_angle - np.deg2rad(15),
robot_angle + np.deg2rad(15))
a_particle = Particle(randX, randY, 1, i, self.map,self.env, angle=
    rand_angle)
a_particle.set_tile()
if a_particle.tile.type not in ['floor', 'asphalt', 'grass']:
self.p_list.append(a_particle)
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.

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

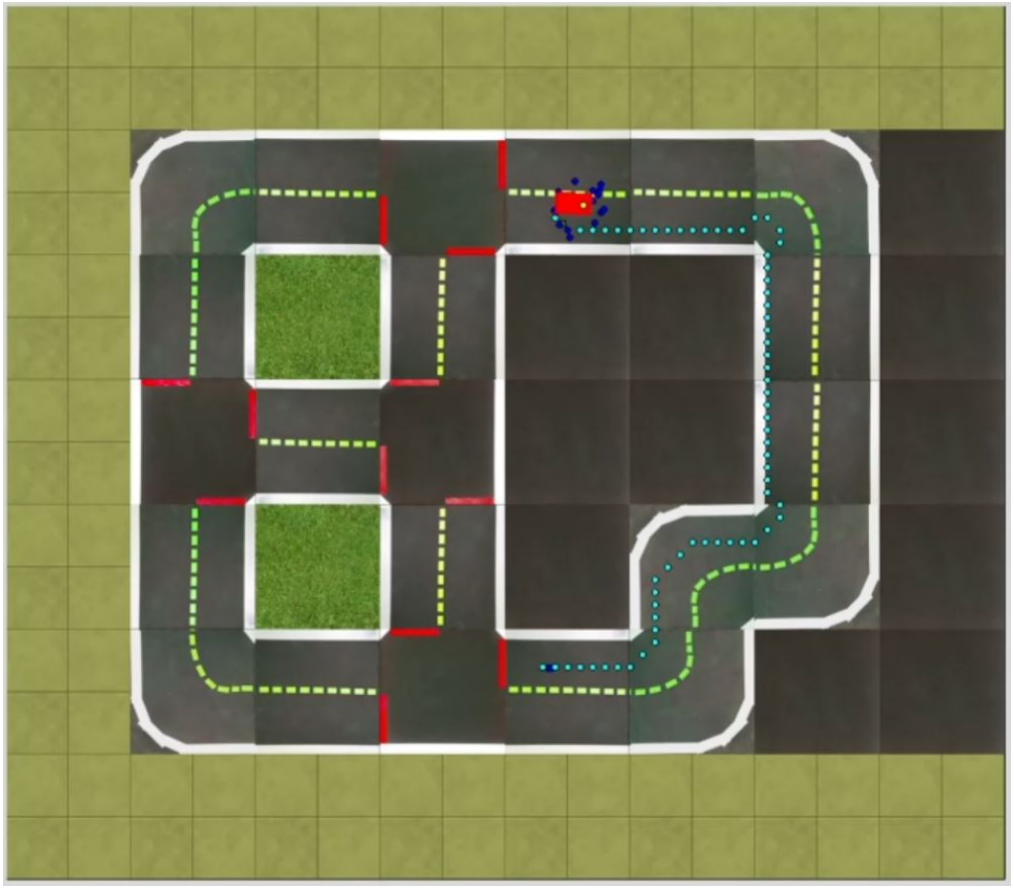

Abbildung 5.1: Partikel Initialisierung
Partikel werden in der Nähe des Duckies initialisiert.
Die geschätzte Pose ist als gelber Punkt markiert.

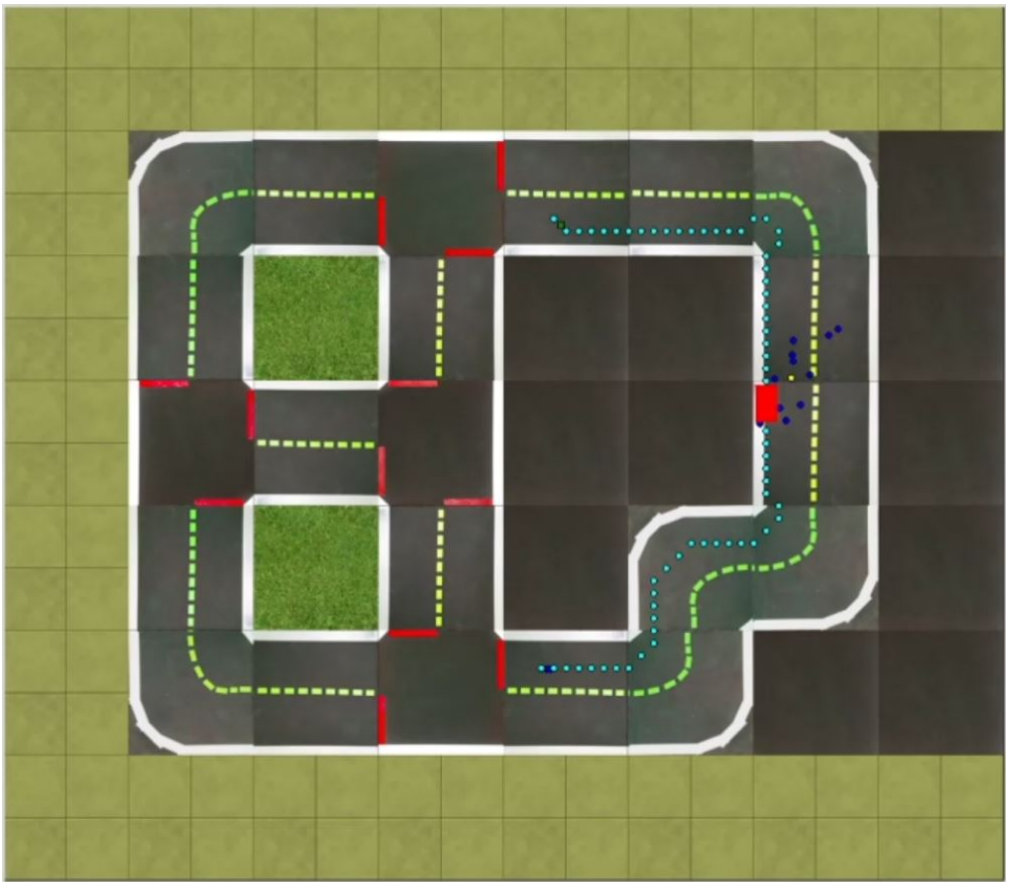

Abbildung 5.2: Partikel zerstreuen sich
Die Partikel zerstreuen sich, aufgrund mehrerer möglicher Positionen.
Die geschätzte Pose ist als gelber Punkt markiert.

```
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”.

```
def integrate_measurement(self, distance_duckie, angle_duckie):
for p in self.p_list:
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.

```

def weight_calculator(self, distance, angle):
cur_pos = [self.p_x, 0, self.p_y]
lane_pos = self.env.get_lane_pos2(cur_pos, self.angle)
distance_p =lane_pos.dist
angle_p = lane_pos.angle_rad
self.weight = self.weight * norm.pdf(distance - distance_p)* norm.
    pdf(angle - angle_p)
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.

```

def resampling(self):
arr_particles = []
for i in range(0, len(self.p_list)):
idx = self.roulette_rad()
arr_particles.append(self.p_list[idx])
sum_py = 0
sum_px = 0
sum_angle = 0
for x in arr_particles:
sum_px = sum_px + x.p_x
```

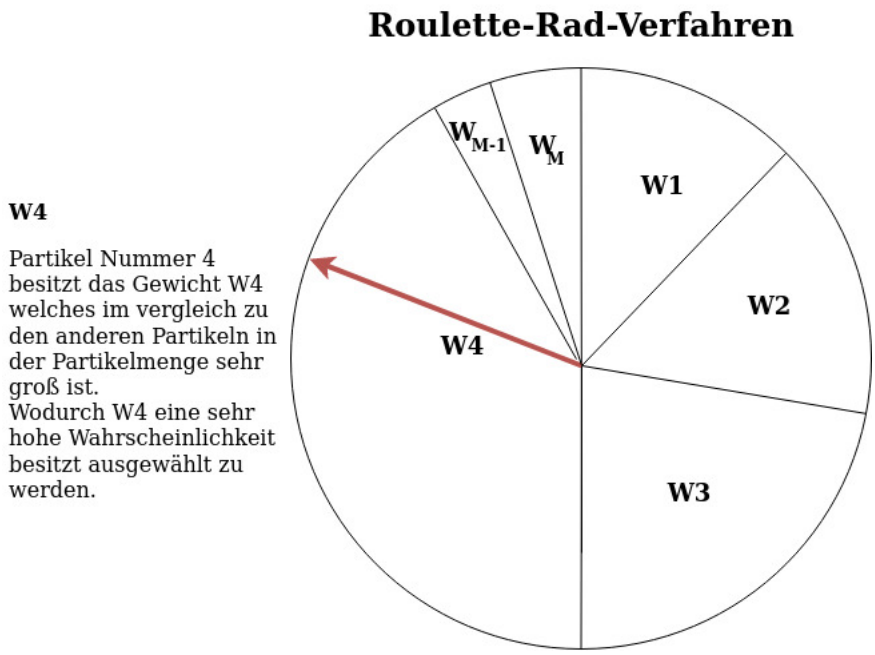

Abbildung 5.3: Roullete rad
source by:`draw.io`

```
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: Resmapling

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)
```

```
the_chosen_one = random.uniform(0, weight_of_particle)
idx_particle = bisect_left(weight_arr, the_chosen_one)
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.

```

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

Listing 5.9: Reward

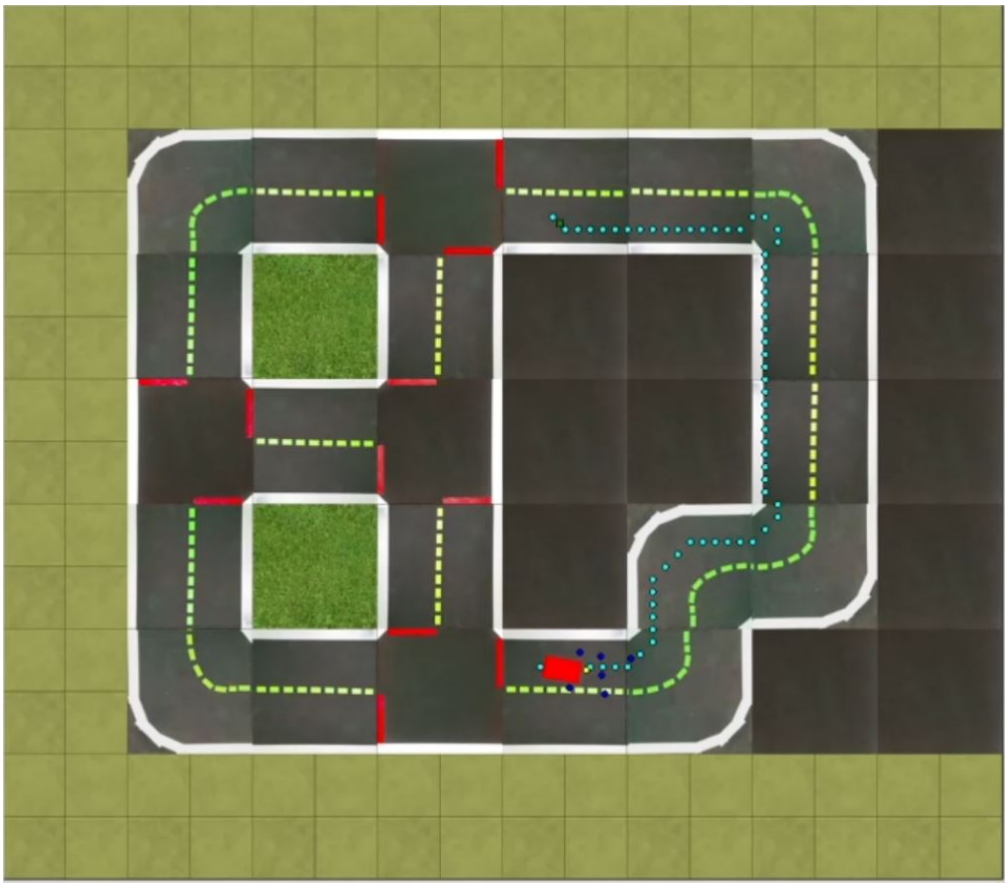

Abbildung 5.4: Ende der Route, Monte Carlo nach mehreren Schritten
Der Monte Carlo vergleicht bereits seid mehreren Schritten Sensorwerte.
Die geschätzte Pose ist als gelber Punkt markiert.

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 befinden 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.

```
    def global_angle_arr(self, point, i):
        t = self.cur_angle
        x = self.cur_pos[0]
        y = self.cur_pos[2]
        print("n chster punkt", point[i][1] / 10, point[i][0] / 10)
        rot = np.arctan2((y - point[i][0] / 10), (point[i][1] / 10 -
    x))
        if t is None:
            t = 0
        a = rot - t
        a = (a / abs(a)) * (abs(a) % (2 * np.pi))
        print(i, "t = ", t, "rot = ", rot, "a = ", a)
        if (a >= np.pi):
            a = -2 * np.pi + a
            return a
        if (a < -np.pi):
            a = 2 * np.pi + a
            return a
        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 Auswahlmöglichkeiteten 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 erwieß sich die Skalierung der Karte aufgrund mangelhafter Dokumentation des Grid-Modules als äußerst kompliziert und führte in unseren Render-Versuchen zu derben Performancebrüchen.

7.1.1 TkInter

TkInter ist ein Standart-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 erwieß), die Möglichkeit diverse Maus-Events abzufangen (zur Interaktion) sowie Methoden um Canvas-Elemente in beliebiger Form (Polygon, Oval, Rectangle) zu zeichnen.

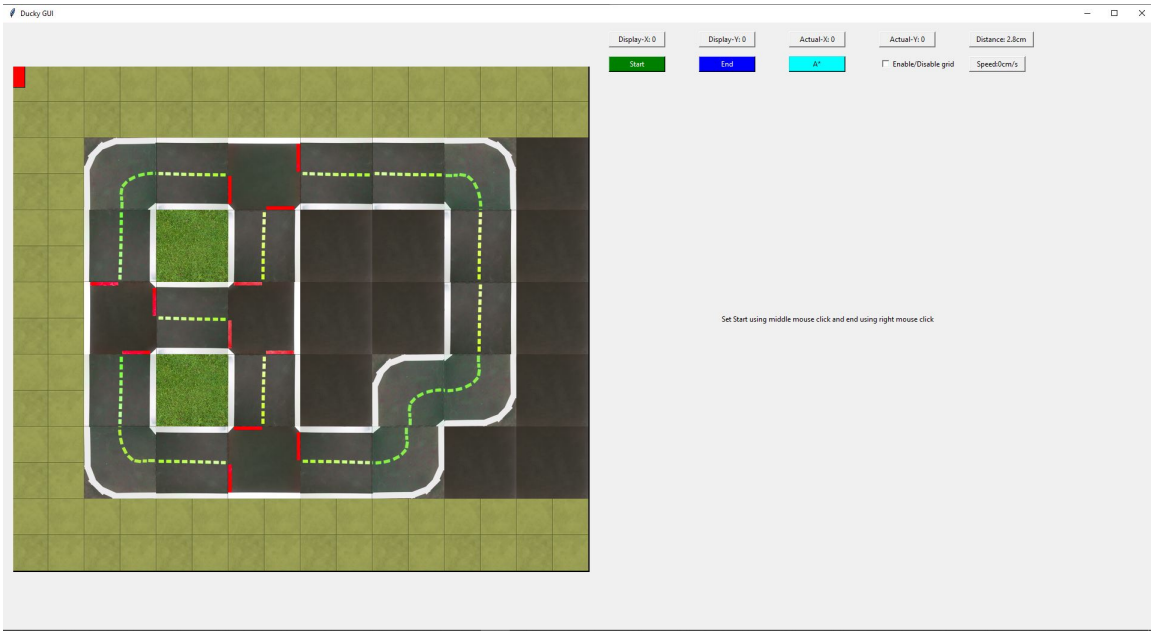

Abbildung 7.1: GUI

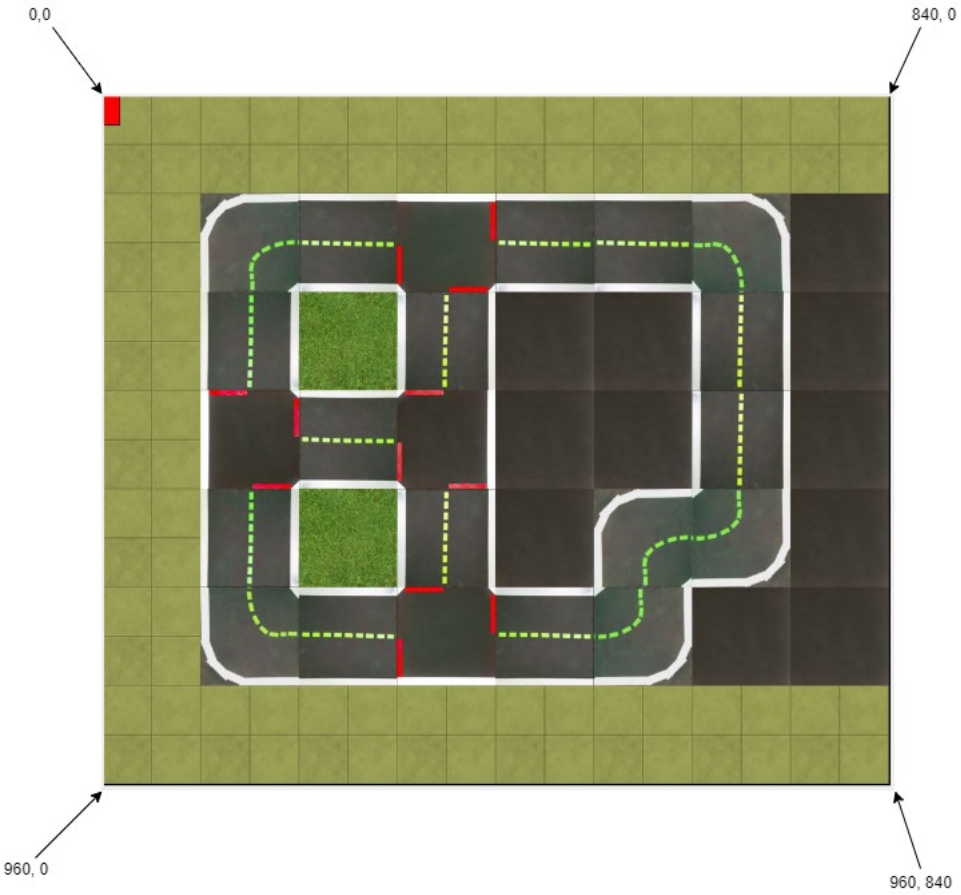

Abbildung 7.2: Coordinate system
source by:`draw.io`

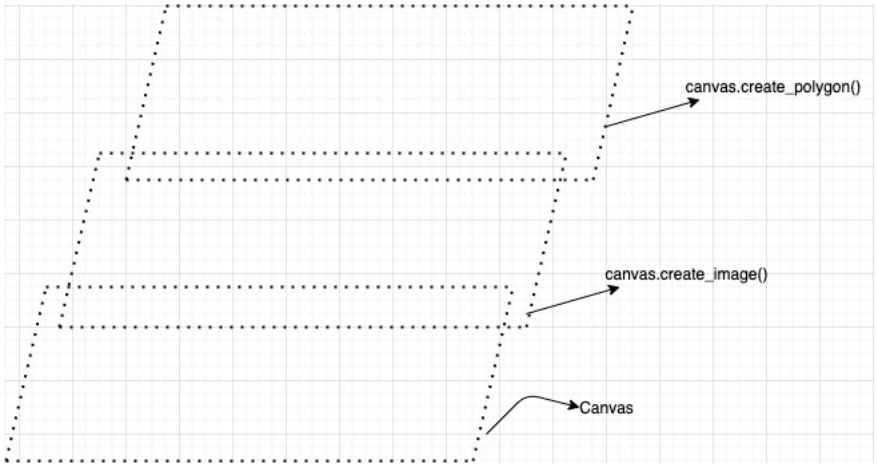

Abbildung 7.3: Canvas Anzeigeliste
source by:`draw.io`

7.1.1.1 Canvas

Der Canvas besitzt eine Anzeigeliste, die sich auf der Rheinfolge 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 eingefuegt. Zudem werden Canvas-Elementen mit einem Tag bezeichnet, damit man ein 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 I 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 Programmier-Bibliothek “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 ·13$cm(gemessen)$
1cm = 6/2,8 (2,14) Einheiten
21cm = 45 (2,14·21)$Einheiten$
13cm = 27 (2,14 ·13)$Einheiten$
45 ·27$Einheiten$

Tabelle 7.2: Messtabelle.

Der Duckiebot selbst wurde als 36*21 Einheiten(da, 45 * 27 Einheiten der Duckie als zu groß dargestellte) in Form von roten Pixeln, als einzelnes Canvas in das Koordinatensystem gezeichnet und über die KeyPress-API der Tkinter-Bibliothekt 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.

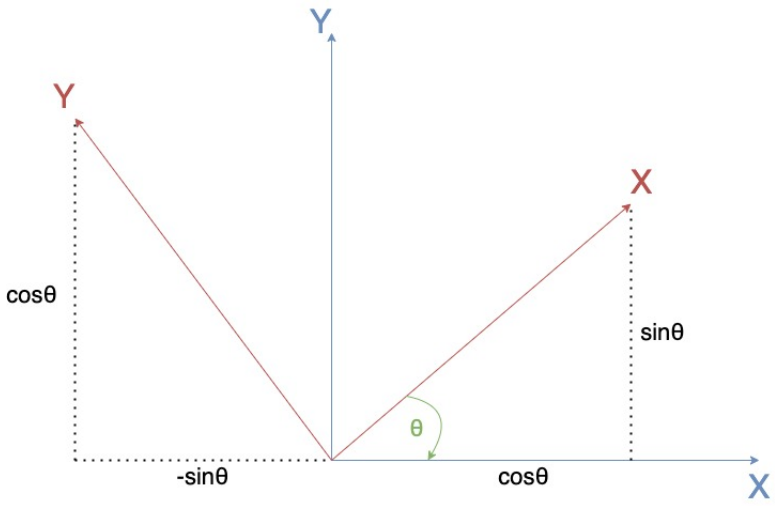

Abbildung 7.4: Rotate Duckie
source by:`draw.io`

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 bezeichet mit G = Distanz von Startknoten A. Umgekehrt wird via Heuristik die Distanz vom Endknoten B zur jeweiligen Kachel (Knoten) berechnet, die sogenannten H-Kosten mit H = 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

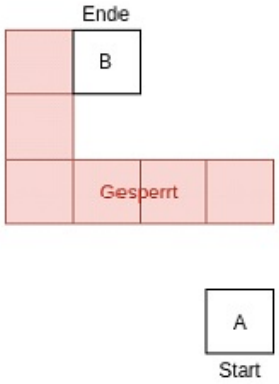

Abbildung 7.5: Gesucht ist der schnellste Weg von A nach B mit Ausnahme über die Rot markierten Felder
source by:`draw.io`

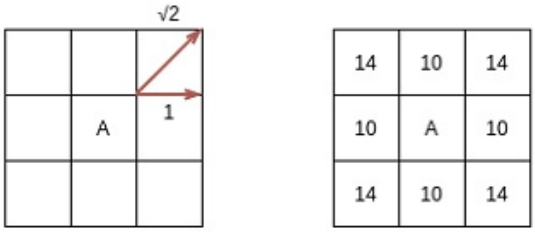

Abbildung 7.6: Gewichtung der Kosten
source by:`draw.io`

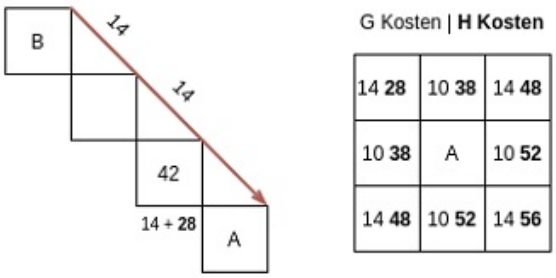

Abbildung 7.7: Beispiel für eine Zusammensetzung der Kosten
source by:`draw.io`

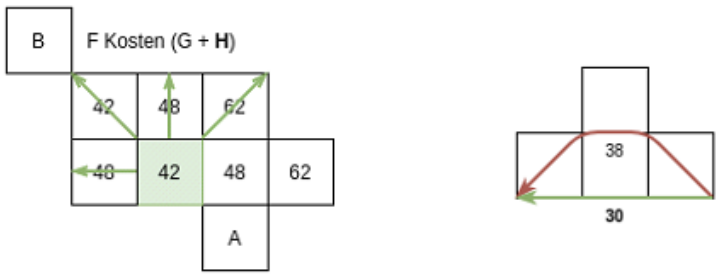

Abbildung 7.8: Auswahl des günstigsten Knoten und Ermitteln neuer Nachbarschaft
source by:`draw.io`

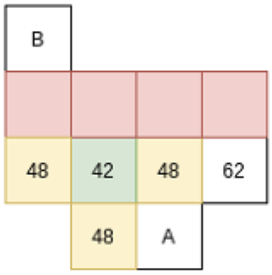

Abbildung 7.9: Bei mehreren Kandidaten wählt der Algorithmus den Kürzesten zu B aus
source by:`draw.io`

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.

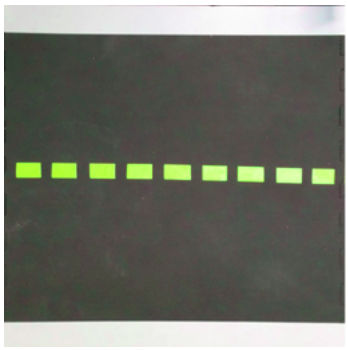

Abbildung 7.11: Ein gerades Stück Fahrbahn

```
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. Paralellisierung 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 Parallelisierung von Simulator und GUI konnte durch die Implementierung konkurrenter Module (Multi-Threading) erreicht werden. Das Projekt wurde auf der Distribution Ubunutu LTS 20.04 umgesetzt und getestet. Auf Windows 10 und Mac IOS gibt es Kompilationsprobleme mit tKinter und pyglet. 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 Mettings 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 LaTeX for Your Thesis.*

[Swift, 2017] Swift, N. (2017). *Easy A* (star) Pathfinding.*

[tutorials point, 2020] tutorials point (2020). *Python - Multithreaded Programming.*

Anhang

- gui.py

```
  # The following class defines GUI.
import threading
import tkinter as tk
import numpy as np
from lokalisierung.duckietown_env import DuckietownEnv
from modules import astar, navigation, grid, mapping, findpath
import time
from Observer import Subscriber
from threading import Thread


class GUI(threading.Thread, Subscriber):

    def __init__(self):
        threading.Thread.__init__(self)
        self.root = tk.Tk()
        self.root.title('Ducky GUI')
        self.root.geometry('1920x1080')
        self.gboll = True
        self.initialise_map = mapping.Mapping(tk)
        self.canvas_and_ducky = self.initialise_map.gen_map('udem1')
        self.canvas = self.canvas_and_ducky[0]
        self.ducky = self.canvas_and_ducky[1]
        self.generated_grid = grid.Grid(self.canvas)

        self.number_of_clicks = 0
        self.text_y_display = tk.StringVar()
        self.text_x_display = tk.StringVar()
        self.text_x_actual = tk.StringVar()
        self.text_y_actual = tk.StringVar()
        self.distance_traveled_display = tk.StringVar()
        self.start_point = tk.StringVar()
        self.end_point = tk.StringVar()
        self.display_speed = tk.StringVar()
        self.label_explain = tk.StringVar()

        self.myduckietown = DuckietownEnv(GUI=self, domain_rand=
    False, draw_bbox=False, map_name="udem1")
        self.START = [0, 0]
        self.END = [0, 0]
        self.start = []
        self.end = []
        self.ovals = []

```

```
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, '
    number_clicks': 0}
57
58      def update(self, message):
59          print("Wird update auch w hrend der simulation aufgerufen?"
    )
60          self.draw_particles(message[0])
61          self.draw_ducky_bot(message[1], message[2])
62          self.draw_expected_pos(message[3])
63
64      def draw_particles(self, parti):
65          for x in range(0, len(parti)):
66              self.canvas.delete('parti' + str(x))
67
68          for x in range(0, len(parti)):
69              print("Particle", x, "is=", parti[x].p_x, parti[x].p_y)
70              self.canvas.create_oval(parti[x].p_x * 120, parti[x].p_y
     * 120, parti[x].p_x * 120 + 6,
71                                      parti[x].p_y * 120 + 6, fill="
    #0000ff",
72                                      tags='parti' + str(x))
73
74      def draw_expected_pos(self, pos):
75          self.canvas.delete('mean_pos_particle')
76          self.canvas.create_oval(pos[0] * 120, pos[2] * 120, pos[0] *
     120 + 6, pos[2] * 120 + 6, fill="yellow",
77                                  tags='mean_pos_particle')
78
79      def set_gui_label_buttons(self):
80
81          """
82
83          :return:
84
85          """
86
```

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

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

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

```
197         if self.end_set:
198             self.canvas.delete('end')
199 
200         self.canvas.create_rectangle(event.x, event.y, event.x + 6,
    event.y + 6, fill="blue", tags="end")
201         self.end_set = True
202         x = int(round(event.x / 12))
203         y = int(round(event.y / 12))
204         self.end.append(x)
205         self.end.append(y)
206 
207         self.end_point.set('{}, {}'.format(x, y))
208 
209     def draw_ducky_bot(self, ducky_pos, ducky_angle):
210 
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",
    tags='ducky')
228 
229     def move_ducky(self, line):
230 
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
    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             ducky_speed = (5.6 / self.sleep_time)
251 
252             parti = []
253             for i in range(-25, 25):
254                 parti.append((scaled_x + i, scaled_y + i))
255 
256             # self.draw_particles(parti)
257 
258             # print the coordinates of middle point of the ducky
259             # print("scaled_x = ", scaled_x, " scaled_y = ",
    scaled_y)
260             self.text_x_display.set('Display-X: ' + str(scaled_x))
261             self.text_x_actual.set('Actual-X: ' + str(scaled_x / 12)
    )
262             self.text_y_display.set('Display-Y: ' + str(scaled_y))
263             self.text_y_actual.set('Actual-Y: ' + str(scaled_y / 12)
    )
264             self.display_speed.set('Speed:' + "{:.1f}".format(
    ducky_speed) + 'cm/s')
265             distance_traveled = self.coordinate_elements['
    distance_traveled'] = self.coordinate_elements[
266 
                    'distance_traveled'] + 5.6
267             self.distance_traveled_display.set('Distance: ' + "{:.1f
    }".format(distance_traveled) + 'cm')
268 
269             old_x = scaled_x
270             old_y = scaled_y
271 
272             time.sleep(self.sleep_time)
273 
274     def find_path(self):
275 
276         """
277 
278         :return:
279 
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 
                                       sticky='NW',
314 
                                       padx=15,
315 
                                       pady=0)
316         self.start_point.set('Start')
317         tk.Button(self.root, textvariable=self.end_point, bg='blue',
     fg='white', width=12).grid(row=1, column=1,
318 
                                   sticky='NW',
319 
                                   padx=165,
320 
                                   pady=0)
321         self.end_point.set('End')
322 
```

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

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

```
            subscriber.update(message)


```

Listing 8.2: Observer.py

- astar.py

```
  import numpy
import yaml

way_left_N = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['N', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['N', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['N', 'S', 'S', 'S', 'N', 'N', 'N', 'N', 'N', 'X'],
                ['X', 'S', 'S', 'S', 'N', 'N', 'N', 'N', 'N', 'X'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['S', 'S', 'S', 'S', 'N', 'N', 'N', 'N', 'N', 'X'],
                ['S', 'S', 'S', 'S', 'N', 'N', 'N', 'N', 'N', 'X'],
                ['S', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]

way_left_E = [['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'X', 'X'],
                ['W', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'X', 'X'],
                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
                ['W', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'W', 'W'],
                ['X', 'X', 'X', 'X', 'X', 'E', 'E', 'E', 'E', 'X'],
                ['X', 'X', 'X', 'X', 'E', 'E', 'E', 'E', 'E', 'X'],
                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]

way_left_S = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'X', 'X', 'X', 'X'],
                ['X', 'S', 'S', 'S', 'S', 'S', 'N', 'N', 'N', 'N'],
                ['X', 'S', 'S', 'S', 'S', 'S', 'N', 'N', 'N', 'S'],
                ['X', 'S', 'S', 'S', 'S', 'S', 'N', 'N', 'N', 'S'],
                ['X', 'S', 'S', 'S', 'S', 'S', 'N', 'N', 'N', 'X'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]

way_left_W = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
                ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
                ['X', 'W', 'W', 'W', 'W', 'X', 'X', 'X', 'X', 'X'],
                ['X', 'W', 'W', 'W', 'X', 'X', 'X', 'X', 'X', 'X'],
```

```
                ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
                ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
                ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
                ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X']]

curve_left_n = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
                ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'X'],
                ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'X'],
                ['N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'N', 'X'],
                ['X', 'X', 'X', 'X', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['X', 'X', 'X', 'X', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['S', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['S', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['X', 'X', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
                ['X', 'X', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]

curve_left_e = [['X', 'X', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X'],
                ['X', 'X', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X'],
                ['W', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X'],
                ['W', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'X'],
                ['X', 'X', 'X', 'X', 'X', 'E', 'E', 'E', 'E', 'X'],
                ['X', 'X', 'X', 'X', 'E', 'E', 'E', 'E', 'E', 'X'],
                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'X'],
                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'X'],
                ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'X'],
                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]

curve_left_s = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'X', 'X'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'X', 'X'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'S'],
                ['X', 'S', 'S', 'S', 'X', 'X', 'X', 'X', 'X', 'X'],
                ['X', 'S', 'S', 'S', 'S', 'S', 'X', 'X', 'X', 'X'],
                ['X', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S'],
                ['X', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S'],
                ['X', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S', 'S'],
                ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]

curve_left_w = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
                ['X', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
                ['X', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
                ['X', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
                ['X', 'W', 'W', 'W', 'W', 'X', 'X', 'X', 'X', 'X'],
                ['X', 'W', 'W', 'W', 'X', 'X', 'X', 'X', 'X', 'X'],
                ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
                ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'E', 'E'],
                ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'X', 'X'],
                ['X', 'W', 'W', 'W', 'X', 'X', 'E', 'E', 'X', 'X']]

```

```python
straight_w = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
              ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
              ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
              ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
              ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
              ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
              ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
              ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
              ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
              ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]

straight_n = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]

straight_s = [['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X'],
              ['X', 'S', 'S', 'S', 'X', 'X', 'N', 'N', 'N', 'X']]

straight_e = [['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
              ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
              ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
              ['W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W', 'W'],
              ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
              ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X'],
              ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
              ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
              ['E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E', 'E'],
              ['X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X', 'X']]


class Node:
    def __init__(self, parent=None, position=None):
        self.parent = parent
        self.position = position
```

```
141
142         self.g = 0
143         self.h = 0
144         self.f = 0
145
146     def __eq__(self, other):
147         return self.position == other.position
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
    [len(maze) - 1]) - 1), maze):
182             if d[x.getIndex()] == 200000:
183                 kl.append(x)
184             cost = d[v.getIndex()] + 1  # maze[x.position[0]][x.
    position[1]]
185             if cost < d[x.getIndex()]:
186                 p[x.getIndex()] = v
187                 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) + ((
    x.position[1] - end.position[1]) ** 2))
197         if low > d[x.getIndex()] + h:
198             low = d[x.getIndex()] + h
199             v = x
200     return v
201 
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)
    , (-1, 1), (1, -1), (1, 1)]:
217     adjacent_vertices = []
218     # (y,x)
219     print('maze value at y = ', current_node.position[0], 'x =',
    current_node.position[1], 'value =' , maze[current_node.position
    [0]][current_node.position[1]])
220     if maze[current_node.position[0]][current_node.position[1]] == '
    S':
221         adjacent_vertices = [(0, -1), (1, 0), (0, 1), (1, -1), (1,
    1)]
222     elif maze[current_node.position[0]][current_node.position[1]] ==
     'N':
223         adjacent_vertices = [(0, 1), (-1, 0), (0, -1), (-1, 1), (-1,
     -1)]
224     elif maze[current_node.position[0]][current_node.position[1]] ==
     'E':
225         adjacent_vertices = [(0, 1), (1, 0), (-1, 0), (1, 1), (-1,
    1)]
226     elif maze[current_node.position[0]][current_node.position[1]] ==
```

```
    'W':
227         adjacent_vertices = [(0, -1), (-1, 0), (1, 0), (-1, -1), (1,
    -1)]
228 
229     for newPosition in adjacent_vertices:
230         # for newPosition in [(0, -1), (0, 1), (-1, 0), (1, 0)]:
231         node_position = (
232             current_node.position[0] + newPosition[0],  # X
233             current_node.position[1] + newPosition[1])  # Y
234         v = Vertex(node_position)
235         # Range
236         range_criteria = [
237             node_position[0] > max_x,
238             node_position[0] < 0,
239             node_position[1] > max_y,
240             node_position[1] < 0,
241         ]
242 
243         if any(range_criteria):
244             continue
245 
246         # if maze[node_position[0]][node_position[1]] > 30:
247         #     continue
248         list_of_vertices.append(v)
249     return list_of_vertices
250 
251 
252 def create_path(list_of_vertices, start, end):
253     path = []
254     previous = end
255     print("number of element in p", len(list_of_vertices))
256     if list_of_vertices[end.getIndex()] is None:
257         return []
258     while start.getIndex() != previous.getIndex():
259         path.append(previous.position)
260         previous = list_of_vertices[previous.getIndex()]
261     path.append(start.position)
262     path.reverse()
263 
264     return path
265 
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 
```

```
x = load_map('udem1')


def create_map(mapdata):
    print('start map')
    data = []
    mod = 1
    for m in mapdata:
        for i in range(0, 10):
            row = []
            for t in m:
                if t == 'floor' or t == 'asphalt' or t == 'grass':
                    for o in range(10):
                        row.append(200)
                else:
                    if t == 'straight/E':
                        for l in straight_e[i]:
                            row.append(l * mod)
                    elif t == 'straight/W':
                        for l in straight_w[i]:
                            row.append(l * mod)
                    elif t == 'straight/N':
                        for l in straight_n[i]:
                            row.append(l * mod)
                    elif t == 'straight/S':
                        for l in straight_s[i]:
                            row.append(l * mod)
                    elif t == 'curve_left/S':
                        for l in curve_left_s[i]:
                            row.append(l * mod)
                    elif t == 'curve_left/N':
                        for l in curve_left_n[i]:
                            row.append(l * mod)
                    elif t == 'curve_left/E':
                        for l in curve_left_e[i]:
                            row.append(l * mod)
                    elif t == 'curve_left/W':
                        for l in curve_left_w[i]:
                            row.append(l * mod)
                    elif t == '3way_left/E':
                        for l in way_left_E[i]:
                            row.append(l * mod)
                    elif t == '3way_left/S':
                        for l in way_left_S[i]:
                            row.append(l * mod)
                    elif t == '3way_left/W':
                        for l in way_left_W[i]:
                            row.append(l * mod)
                    elif t == '3way_left/N':
```

```
                        for l in way_left_N[i]:
                            row.append(l * mod)
                    else:
                        for o in range(10):
                            row.append(1 * mod)

            print(row)
            data.append(row)
    return data


# star_map = create_areas(load_map('udem1'))


def main(start, end):
    star_map = create_map(load_map('udem1'))

    start_x = start[0]
    start_y = start[1]

    start = (start_y, start_x)

    end_x = end[0]
    end_y = end[1]

    end = (end_y, end_x)

    print('a star active')
    print(star_map, start, end)

    v = shortest(star_map, Vertex(start), Vertex(end))
    print('v =', v)
    s = Vertex(start)
    e = Vertex(end)
    if len(v) > 0:
        p = create_path(v, s, e)
        for x in p:
            print(x)
        print("length of my path", len(p))
        return p
    return []


```

Listing 8.3: astar.py

- findpath.py

```
  class Findpath():
    start_set = False
```

```
    end_set = False
    start = []
    end = []
    ovals = []

    def __init__(self, canvas):
        self.canvas = canvas

    def set_start(self, event):
        global start_set
        global start_point
        global start

        if start_set:
            self.canvas.delete('start')

        self.canvas.create_rectangle(event.x, event.y, event.x + 6,
  event.y + 6, fill="green", tags="start")
        start_set = True
        x = int(round(event.x / 12))
        y = int(round(event.y / 12))
        start.append(x)
        start.append(y)

        start_point.set('{}, {}'.format(x, y))


```

Listing 8.4: findpath.py

- grid.py

```
class Grid:
    grid_items = []
    grid_enabled = False

    def __init__(self, canvas):
        self.canvas = canvas
        print(self.canvas)

    def enable_grid(self):
        canvas = self.canvas
        # draw grid on grid
        x1 = 0
        x2 = 1000
        # draw horizontal lines
        for k in range(0, 1000, 6):
            y1 = k
            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:
```

```
            imap = yaml.load(map_file, Loader=yaml.FullLoader)

        dimension = self.calc_mapsize(imap)

        try:
            canvas.destroy()
        except:
            pass

        canvas = self.tk.Canvas(width=dimension[0], height=dimension
    [1], bg='black')
        canvas.grid(row=1, column=0, padx=15, pady=15)

        for row in range(len(imap['tiles'])):
            for column in range(len(imap['tiles'][0])):
                image_file = self.get_image(imap['tiles'][row][
    column])
                image = Image.open(image_file)
                resized_image = image.resize((120, 120), Image.
    ANTIALIAS)
                self.one.append(ImageTk.PhotoImage(resized_image))
                canvas.create_image(column * 120 + 60, row * 120 +
    60, image=self.one[-1])

        x = 0
        y = 0

        ducky = canvas.create_rectangle(x, y, x + 21, y + 36, fill="
    red", tags='ducky')
        return [canvas, ducky]

    def get_image(self, name):
        name = name.replace('/', '_').lower()
        try:
            open('./ducky_pictures/' + name + '.png', 'r')
            return './ducky_pictures/' + name + '.png'
        except:
            return './ducky_pictures/' + 'cub.png'

    def calc_mapsize(self, imap):
        height = len(imap['tiles']) * 120
        width = len(imap['tiles'][0]) * 120

        return [width, height]

```

Listing 8.6: mapping.py

- duckietown-env.py

```python
  # coding=utf-8
import threading

import numpy as np
from gym import spaces

from lokalisierung.Ducky_map import DuckieMap
from lokalisierung.MCL import MCL
from gym_duckietown import logger
from gym_duckietown.simulator import Simulator
from Observer import Publisher


class DuckietownEnv(Simulator, threading.Thread, Publisher):
    """
    Wrapper to control the simulator using velocity and steering angle
    instead of differential drive motor velocities
    """

    def __init__(
            self,
            gain=1.0,
            trim=0.0,
            radius=0.0318,
            k=27.0,
            limit=1.0,
            line=None,
            GUI=None,
            **kwargs
    ):
        Publisher.__init__(self)
        Simulator.__init__(self, **kwargs)
        logger.info('using DuckietownEnv')
        threading.Thread.__init__(self)
        self.action_space = spaces.Box(
            low=np.array([-1, -1]),
            high=np.array([1, 1]),
            dtype=np.float32
        )

        # Should be adjusted so that the effective speed of the robot is 0.2 m/s
        self.gain = gain

        self.line = line

        # Directional trim adjustment
        self.trim = trim
```

```
48
49          self.register(GUI)
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 -
     x))
66          if t is None:
67              t = 0
68          a = rot - t
69          a = (a / abs(a)) * (abs(a) % (2 * np.pi))
70          print(i, "t = ", t, "rot = ", rot, "a = ", a)
71          if (a >= np.pi):
72              a = -2 * np.pi + a
73              return a
74          if (a < -np.pi):
75              a = 2 * np.pi + a
76              return a
77          return a
78
79      def global_angle(self, point):
80          t = self.cur_angle
81          x = self.cur_pos[0]
82          y = self.cur_pos[2]
83          rot = np.arctan2((y - point[1]), (point[0] - x))
84          a = rot - t
85          a = (a / abs(a)) * (abs(a) % (2 * np.pi))
86          print(i, "t = ", t, "rot = ", rot, "a = ", a)
87          if (a >= np.pi):
88              a = -2 * np.pi + a
89              return a
90          if (a < -np.pi):
91              a = 2 * np.pi - a
92              return a
93          return a
94
95      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] /
    10]
104         self.cur_angle = np.pi / 2
105
106         my_map = DuckieMap("maps/udem1.yaml")
107         particle_number = 25
108         mcl = MCL(particle_number, my_map, self)
109         mcl.spawn_particle_list(self.cur_pos, self.cur_angle)
110         step_counter = 0
111         newAStar = False
112         while True:
113             if io == len(self.line):
114                 newAStar = True
115                 continue
116             if newAStar:
117                 print('new aStar starts')
118                 self.cur_pos = [self.line[0][1] / 10, 0, self.line
    [0][0] / 10]
119                 self.cur_angle = np.pi / 2
120                 mcl.spawn_particle_list(self.cur_pos, self.cur_angle
    )
121                 newAStar = False
122             lane_pose = self.get_lane_pos2(self.cur_pos, self.
    cur_angle)
123             print("self.cur_pose =")
124             print(self.cur_pos)
125             print("\n")
126             distance_to_road_center = lane_pose.dist
127             angle_from_straight_in_rads = lane_pose.angle_rad
128             print("dist = ", distance_to_road_center, " rad = ",
    angle_from_straight_in_rads)
129
130             speed = 0.2
131             tol = 0.15
132             if abs(self.cur_pos[0] - self.line[io][1] / 10) <= tol
    and abs(self.cur_pos[2] - self.line[io][0] / 10) <= tol:
133                 io = io + 1
134                 if io == len(self.line):
135                     io = 0
136                     self.line = []
137                     print('length of line', len(self.line))
138                     continue
```

```
            steering = self.global_angle_arr(self.line, io)
            obs, reward, done, info = self.step([speed, steering])
            mcl.integrate_movement([speed, steering])
            step_counter += 1
            mcl.integrate_measurement(distance_to_road_center,
angle_from_straight_in_rads)
            if step_counter % 2 == 0:
                # start = time.time()
                arr_chosenones, possible_location, possible_angle =
mcl.resampling()
                self.dispatch([arr_chosenones, self.cur_pos, self.
cur_angle])
                # end = time.time()
                # duration = end - start
                print("posloc and robot position", possible_location
, self.cur_pos)
                print('possible angle and robot angle',
possible_angle, self.cur_angle)
                mcl.weight_reset()

            # obs, reward, done, info = self.step([speed, loca_angle
(angle_from_straight_in_rads, distance_to_road_center)])
            # obs, reward, done, info = self.step([speed, 0])
            # total_reward += reward
            print("info = ", info)
            # print('Steps = %s, Timestep Reward=%.3f, Total Reward
=%.3f' % (self.step_count, reward, total_reward))

            self.render()

    def step(self, action):
        vel, angle = action

        # Distance between the wheels
        baseline = self.unwrapped.wheel_dist

        # assuming same motor constants k for both motors
        k_r = self.k
        k_l = self.k

        # adjusting k by gain and trim
        k_r_inv = (self.gain + self.trim) / k_r
        k_l_inv = (self.gain - self.trim) / k_l

        omega_r = (vel + 0.5 * angle * baseline) / self.radius
        omega_l = (vel - 0.5 * angle * baseline) / self.radius

        # conversion from motor rotation rate to duty cycle
        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
    )]
```

```
                for i in range(h):
                    for j in range(w):
                        self.tiles[j][i] = Tile(j, i, tiles[i][j])
            except yaml.YAMLError as exc:
                print(exc)

    def search_tile(self, x, y):
        return self.tiles[x][y]


class Particle:

    def __init__(self, p_x, p_y, weight, name, angle=-1):
        self.p_x = p_x
        self.p_y = p_y
        self.tile = None
        self.weight = weight
        self.name = name
        self.angle = angle

    def __repr__(self):
        return 'Particle ' + str(self.name)

    def set_tile(self, map):
        self.tile = map.search_tile(int(self.p_x), int(self.p_y))

    def distance_to_wall(self):
        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)
            distance = self.p_y % 1
            if distance >= 0.5:
                return 1 - distance - self.tile.WhiteTapeWidth
            return distance - self.tile.WhiteTapeWidth
        if self.tile.type == "straight/N" or self.tile.type == "straight/S":  # returns the disctance to the closest wall (the wall can be on the left or on the right)
            distance = self.p_x % 1
            if distance >= 0.5:
                return 1 - distance
            return distance
        if self.tile.type == "3way_left/S":
            return self.p_x % 1
        if self.tile.type == "3way_left/N":
            return 1 - self.p_x % 1
        if self.tile.type == "3way_left/W":
            return self.p_y % 1
        if self.tile.type == "3way_left/E":
            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) + ') ' +
        'Type: ' + self.type
77
78      def index(self):
79          return self.x, self.y
80
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)
```

```
    for i in particle_list1:
        print(f"particle name: {i.name} particle X: {i.p_x} particle y: {i.p_y} tile: {i.tile}")
        print(i.distance_to_wall())


```

Listing 8.8: Ducky-map.py

- MCL.py

```
 from lokalisierung.Particle import Particle
from lokalisierung.Ducky_map import DuckieMap
import random
from bisect import bisect_left
import numpy as np


class MCL:
    def __init__(self, p_number, map, env):
        self.p_number = p_number
        self.map = map
        self.p_list = None
        self.env = env

    def spawn_particle_list(self, robot_pos, robot_angle):
        self.p_list = list()
        p_x, p_y = (robot_pos[0], robot_pos[2])
        i = 0
        while i < self.p_number:
            randX = random.uniform(p_x - 0.25, p_x + 0.25)
            randY = random.uniform(p_y - 0.25, p_y + 0.25)
            rand_angle = random.uniform(robot_angle - np.deg2rad(15), robot_angle + np.deg2rad(15))
            a_particle = Particle(randX, randY, 1, i, self.map,self.env, angle=rand_angle)
            a_particle.set_tile()
            if a_particle.tile.type not in ['floor', 'asphalt', 'grass']:
                self.p_list.append(a_particle)
                i = i + 1

    def integrate_movement(self, action):
        for p in self.p_list:
            p.step(action)
        self.filter_particles()

    def integrate_measurement(self, distance_duckie, angle_duckie):
        for p in self.p_list:
            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.tilesize = 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     # todo: Jan should review this
28     def step(self, action: np.ndarray):
29         vel, angle = action
30         baseline = WHEEL_DIST
31
32         # assuming same motor constants k for both motors
33         k_r = 27.0
34         k_l = 27.0
35
36         # adjusting k by gain and trim
37         k_r_inv = (1.0 + 0) / k_r
38         k_l_inv = (1.0 - 0) / k_l
39
40         omega_r = (vel + 0.5 * angle * baseline) / 0.0318
41         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)
    * norm.pdf(angle - angle_p)
75          return self.weight
76
```

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