# Übung 3— Navigation, Lokalisierung und Kartierung

## Teil 1 - Navigation

#!/usr/bin/env python

import rospy

from sensor\_msgs.msg import LaserScan

from geometry\_msgs.msg import Twist

import math

class NavigationNode:

def \_\_init\_\_(self):

rospy.init\_node('navigation\_node')

self.velocity\_pub = rospy.Publisher('/cmd\_vel', Twist, queue\_size=10)

rospy.Subscriber('/scan', LaserScan, self.laser\_callback)

self.d\_min = 0.2

self.d\_max = 1.0

self.v\_max = 0.5

def calc\_weight(self, angle, distance):

w\_ori = math.cos(angle / 1.2)

w\_dist = 1.0 / (1.0 + math.exp(-(distance - 1.5) / 0.20))

w\_dist = 1.0 if distance > 3.5 else w\_dist

return w\_ori \* w\_dist

def laser\_callback(self, laser\_data):

range1 = range(330, 360)

range2 = range(0, 31)

desired\_range = list(range1) + list(range2)

filtered\_ranges = [1.0 if math.isinf(val) or val == 0 else val

for i, val in enumerate(laser\_data.ranges)]

min\_distance = min(val for i, val in enumerate(filtered\_ranges) if i in desired\_range)

v = 0.0

if min\_distance > self.d\_min:

if min\_distance < self.d\_max:

v = self.v\_max \* min\_distance / self.d\_max

else:

v = self.v\_max

range1 = range(270, 360)

range2 = range(0, 91)

desired\_range = list(range1) + list(range2)

weights = []

angle = 0

for (i, d) in enumerate(filtered\_ranges):

if i in desired\_range:

weights.append((angle, self.calc\_weight(math.radians(angle), d)))

angle += 1

if angle > 90:

angle = -90

alpha = math.atan2( sum(math.sin(math.radians(angle)) \* w for i,(angle,w) in enumerate(weights)),

sum(math.cos(math.radians(angle)) \* w for i,(angle,w) in enumerate(weights)))

twist\_msg = Twist()

twist\_msg.linear.x = v

twist\_msg.angular.z = alpha

self.velocity\_pub.publish(twist\_msg)

if \_\_name\_\_ == '\_\_main\_\_':

try:

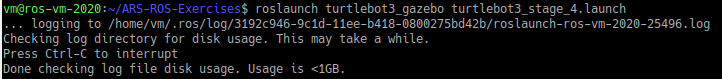
nav\_node = NavigationNode()

rospy.spin()

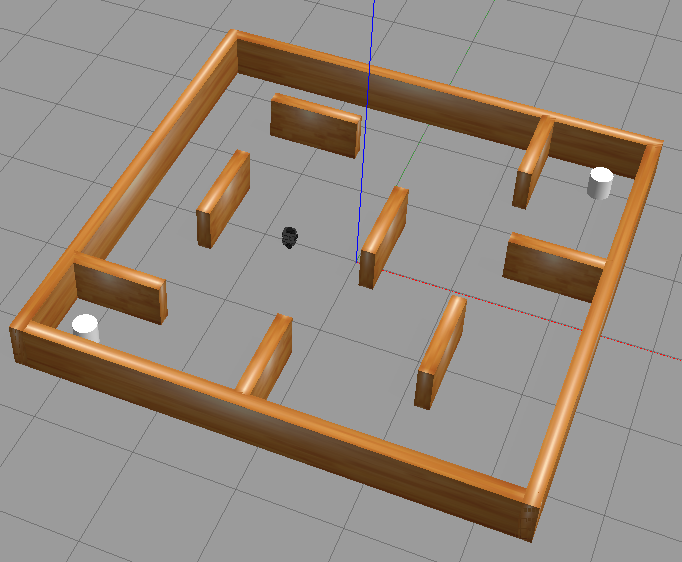
except rospy.ROSInterruptException:

pass

Um den oben gezeigten Code zu testen, wird zunächst eine Gazebo Simulation gestartet. Hierfür wird folgender Command ausgeführt:



Dies startet Gazebo mit der folgenden Welt:



Anschließend wird mit folgendem Command der vorher gezeigte Node ausgeführt:



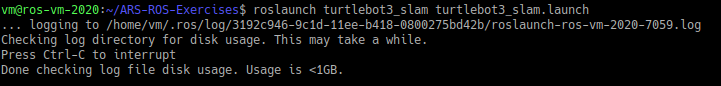
## Teil 2 – Kartierung

Hierfür wurde für Testzwecke folgende Map verwendet:

Ein Bild, das Design, Kunst enthält.

Automatisch generierte Beschreibung mit geringer Zuverlässigkeit

Anschließend wurde ein Rosbag aufgenommen während der Roboter mithilfe der Node aus Teil 1 durch die Map navigiert. Anschließend wurde folgender Command ausgeführt, um RViz zu starten und für das SLAM-Verfahren alles vorzubereiten:



In RViz wurde noch eine Odometry eingefügt, um die Trajektorie mit anzuzeigen. Anschließend wurde das aufgenommene bag-File mit folgenden Command abgespielt:

Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

Ein Bild, das Screenshot, Reihe, Diagramm enthält.

Automatisch generierte BeschreibungEin Bild, das Text, Diagramm, Plan, Reihe enthält.

Automatisch generierte BeschreibungIn den nächsten Abbildungen wird das fertige Ergebnis in RViz gezeigt:

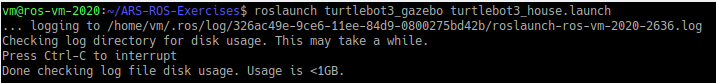
Abschließend wurde mit folgenden Command die Map auf dem map\_server gespeichert.

Ein Bild, das Text, Screenshot, Schrift enthält.

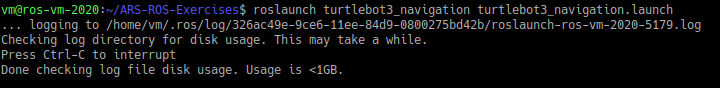
Automatisch generierte Beschreibung

## Teil 3 – Monte Carlo Lokalisierung

Zunächst wird wieder die Gazebo-Simulation mit folgenden Command gestartet.



Anschließend wird indirekt durch ausführend des nachfolgenden Commands AMCL und RViz gestartet.



Hierfür muss jedoch im Skript „amcl.launch“ die initial Pose angegeben werden, hierfür wurde die initial Pose von „turtlebot3\_house.launch“ übernommen. Theoretisch würde AMCL keine korrekte initial Pose benötigen, passt die Pose nicht mit den Sensordaten überein, werden die „Amcl Particles“ stark über die Map gestreut und je weiter der Roboter bewegt wird, desto geringer wird die Streuung. Trotzdem wurde hier die richtige initial Pose angegeben. Und im „turtlebot3\_navigation.launch“ muss die im vorherigen Teil erstellte Karte bzw. der Dateipfad zur Karte angegeben werden.

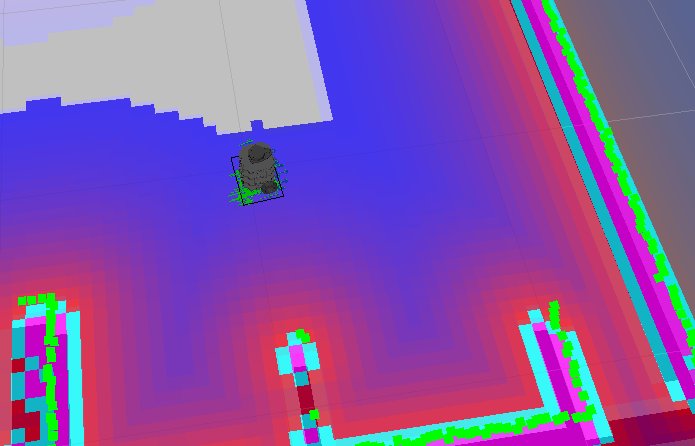
Anschließend wird in RViz durch die „Amcl Particles“ folgendes dargestellt:

Ein Bild, das Screenshot, Farbigkeit, Kunst enthält.

Automatisch generierte Beschreibung

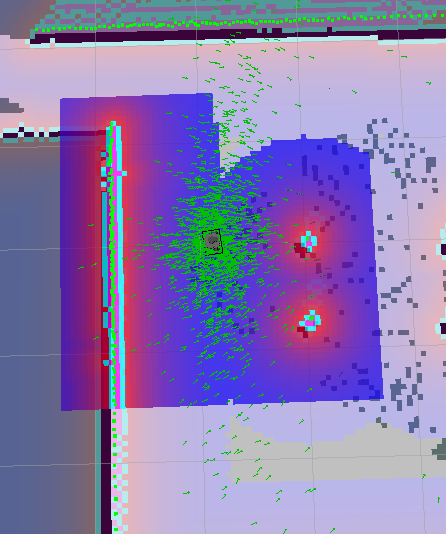
Amcl Particles

Man erkennt das zu Beginn die geschätzte Pose die durch die „Amcl Particles“ noch relatiiv weit gestreut sind, fährt man jedoch mit dem Turtlebot durch die Map wird die Streuung immer geringer, siehe nachfolgende Abbildung und nachfolgenden Command um die Steuerung des Turtlebots zu aktiveren.

  
Ein Bild, das Text, Screenshot, Schrift enthält.

Automatisch generierte Beschreibung

In der nächsten Abbildung wird das Kidnapped-Robot-Problem gezeigt, hierfür wurde der Roboter in Gazebo verschoben. Dementsprechend haben die „Amcl Particles“ in RViz wieder eine große Streuung.



Um dieses Problem zu verbessern kann man im „amcl.launch“-File folgende Parameter optimieren:

* Recovery\_alpha\_slow / recovery\_alpha\_fast: Diese Parameter beeinflussen die Art und Weise, wie AMCL versucht, sich nach unerwarteten Ereignissen zu erholen, wie zum Beispiel, wenn der Roboter "gekidnappt" wird. Die Werte sollten so eingestellt werden, dass sie eine angemessene Balance zwischen Robustheit und Konsistenz bieten. Ich habe die Werte von 0 auf 0.1 jeweils geändert.
* Erhöhung der Partikelanzahl: Um generelle Robustheit zu erhöhen habe ich „min\_particles“ und „max\_particles“ auf 1000 bzw. 5000 erhöht.
* Optimierung update\_min\_d / update\_min\_a: Diese Werte geben an, wie schnell sich die Partikeln updaten, bzw. wie viel Translation und Rotation durchgeführt werden müssen, bis die Partikeln geupdated werden.

*Anmerkung:*

*Für Videos siehe https://github.com/markus-senger/ARS-ROS-Exercises/tree/main/src/exercise-003*