# Übung 1 — Erste Schritte mit der Hardware

## Laserscanner-Statistik

1. **Auf welchen Topics werden die Daten des Laserscanners zur Verfügung gestellt? Wie viele Messwerte liefert der Laserscanner? Was ist die Auflösung und das Field of View?**

Die Daten eines Laserscanners werden über das Topic */scan* veröffentlicht. Abbildung 1 zeigt beispielshaft einen Ausschnitt der Ausgabe des Topics */scan*. Der Message-Typ *sensor\_msgs/LaserScan* wird hierfür verwendet und wird in Abbildung 2 gezeigt.

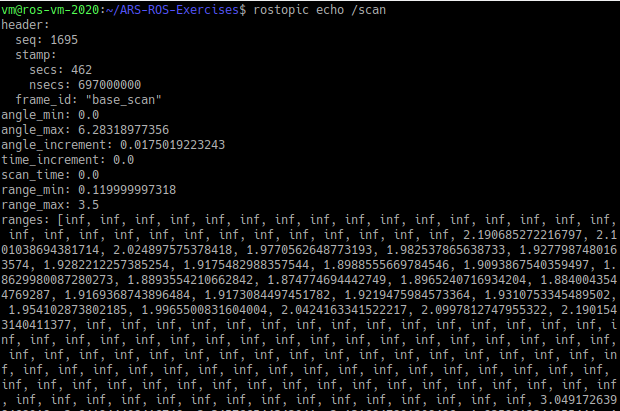


Abbildung - Ausgabe des Topics '/scan'

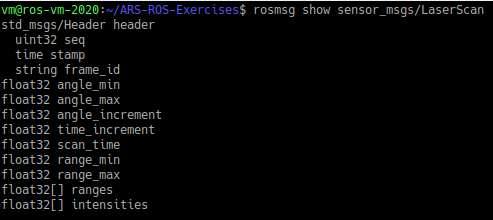


Abbildung - Message-Typ 'sensor\_msgs/LaserScan'

Um die Anzahl der Messwerte zu bestimmen, wurde ein einfaches Python-Skript geschrieben (siehe Code-Snippet 1). Es werden 360 Messwerte vom Laserscanner geliefert, Abbildung 3 zeigt den Aufruf dieses Skriptes.

#!/usr/bin/env python

import rospy

from sensor\_msgs.msg import LaserScan

def laser\_scan\_callback(data):

rospy.loginfo("Anzahl der Messwerte in 'ranges': %d" %

len(data.ranges))

rospy.init\_node('laser\_scan\_listener')

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

rospy.spin()

Code-Snippet - Python-Skript zum Bestimmen der Anzahl der Messwerte



Abbildung - Aufruf von checkNumbersOfMeasurements.py

Um die Winkelauflösung zu berechnen, verwendet man aus Abbildung 1 den Wert *angle\_increment*, zu beachten ist jedoch, dass dieser Wert in Radianten angeben ist. Die Winkelauflösung beträgt daher 1 Grad.

Betrachtet man in Abbildung 1 die Werte angle\_min bzw. angle\_max die in Radianten angegeben werden, erkennt man, dass das Field of View dieses Sensors 360 Grad beträgt.

1. **Nehmen Sie ein Bagfile mit einer Länge von ca. 5 Minuten auf, dass mindestens die Laserscandaten enthält, in dem der Roboter sich nicht bewegt.**

Abbildung 4 zeigt den Command um ein Bagfile für das Topic */scan* aufzunehmen.



Abbildung - Command zum Aufnehmen eines Bagfiles für das Topic '/scan'

Für diese Übung wurden zwei Bagfiles aufgenommen, einmal in einer Simulationsumgebung und einmal mit der echten Hardware.

1. **Zeigen Sie die Daten in rviz an und machen Sie einen Screenshot, der die Umgebung zeigt.**

Um das aufgenommene Bagfile in RViz anzuzeigen, wird RViz zunächst ganz normal gestartet. Anschließend wird der Command aus Abbildung 5 ausgeführt, um das Bagfile wieder abzuspielen.

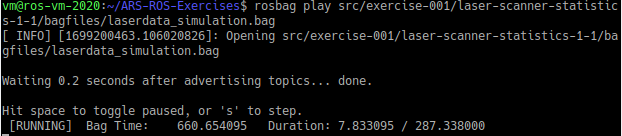


Abbildung - Command zum Abspielen des Bagfiles aus der Simulation

Anschließend wird in RViz mithilfe von Add das Topic */scan* hinzugefügt. Anschließend werden die Daten des Bagfiles angezeigt. Zu beachten ist jedoch, dass in RViz der richtige ‚Fixed Frame‘ ausgewählt ist. Die nachfolgenden Abbildungen zeigen einerseits die Daten der echten Hardware und andererseits die Daten aus der Simulation.

Abbildung - Reale Umgebung (links) und visualisierte Daten in RViz (rechts)

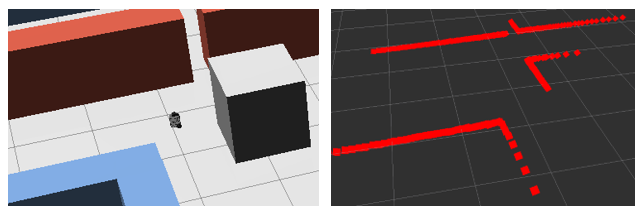
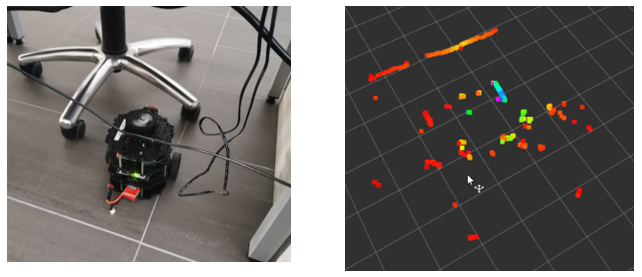


Abbildung - Simulationsumgebung (links) und visualisierte Daten in RViz (rechts)

1. **Schreiben Sie einen ROS-Node zur Berechnung von Mittelwert, Varianz und Standardabweichung für jede der n Messrichtungen des Scanners. Plotten Sie die Werte (z.B. mit gnuplot). Ergeben Sich signifikante Abweichungen in einzelne Richtungen? Wie interpretieren Sie das?**

Code-Snippet 2 zeigt das erstellte Python-Skript.

#!/usr/bin/env python

import rospy

from sensor\_msgs.msg import LaserScan

import statistics

import matplotlib.pyplot as plt

import numpy as np

import subprocess

import threading

values = {}

def laser\_scan\_callback(data):

ranges = np.array(data.ranges)

ranges[np.isinf(ranges)] = 0

cleaned\_ranges = ranges.tolist()

i = 0

for x in cleaned\_ranges:

if i in values:

values[i].append(x)

else:

values[i] = [x]

i += 1

def play\_rosbag(bag\_file\_path):

subprocess.call(['rosbag', 'play', bag\_file\_path])

mean = []

var = []

stdev = []

for i in range(360):

mean.append(statistics.mean(values[i]))

var.append(statistics.variance(values[i]))

stdev.append(statistics.stdev(values[i]))

plt.figure(figsize=(10, 6))

plt.plot(mean, label='mean')

plt.plot(var, label='variance')

plt.plot(stdev, label='standard deviation')

plt.legend()

plt.title('statistics for laserscan')

plt.ylabel('values')

plt.xlabel('scan number')

plt.grid(True)

plt.show()

rospy.init\_node('laser\_scan\_listener\_for\_statistics')

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

bag\_file\_path = 'src/laser-scanner-statistics-1-1/bagfiles/laserdata\_real.bag'

play\_thread = threading.Thread(target=lambda: play\_rosbag(bag\_file\_path))

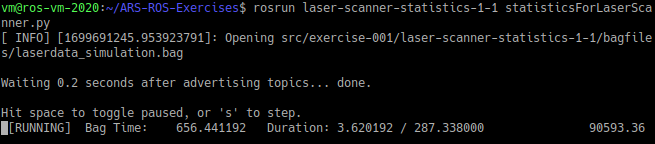
play\_thread.start()

rospy.spin()

Code-Snippet - Skript für statistische Auswertung des Laserscanners

Abbildung 8 zeigt den Command zum Starten der Messungen, sobald das Bagfile abgespielt wurde, werden automatisch die Werte geplottet.

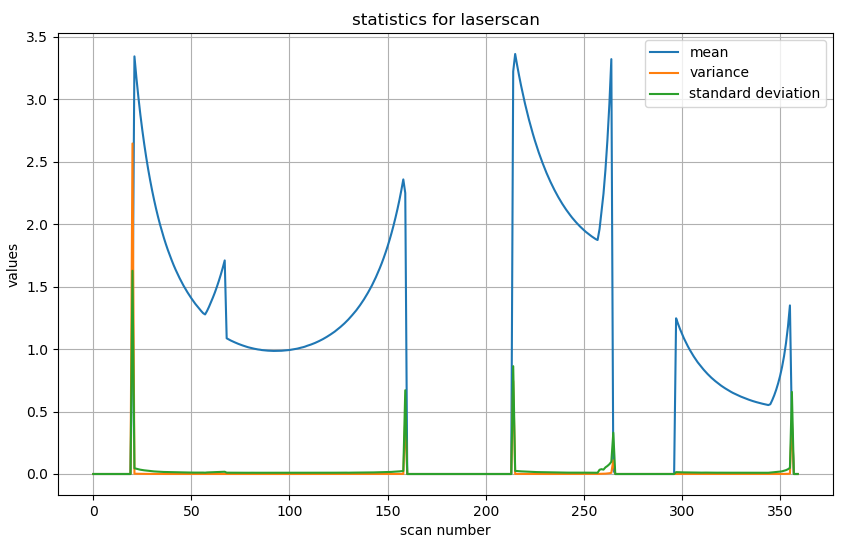
Abbildung - Command zum Starten des statisticsForLaserScanner.py Skripts



Ergebnis Simulation:

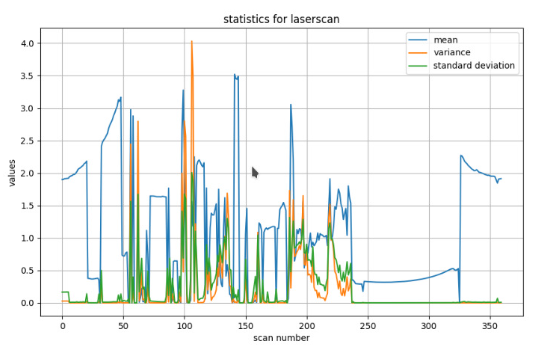
Abbildung 9 zeigt die geplotteten Werte. Da es sich hierbei um die Werte aus der Simulation handelt, ist wie erwartet kein nennenswertes Rauschen zu erkennen. Varianz und Standardabweichung tritt nur bei Beginn oder Ende eines Objektes auf. Daher wird darauf geschlossen, dass das Objekt selbst immer gleich erfasst wird, lediglich bei der genauen Position gibt es Abweichungen.

Abbildung - Laserscanner-Statistik aus der Simulation



Ergebnis reale Hardware:

Abbildung - Laserscanner-Statistik aus der realen Hardware



Wie zu erwarten sind diese Statistiken wesentlich stärker verrauscht als bei den Daten aus der Simulation. Dies lässt sich unteranderem auf Umwelteinflüsse und auf Sensorfehler zurückführen. Zusätzlich dazu ist die Roboter-Position (siehe Abbildung 6) so gewählt worden, dass Objekte (Kabeln, Sesselbeine) nicht unbedingt gut vom LiDAR erfasst werden können. Dies erkennt man auch bei Betrachtung der Varianz bzw. Standardabweichung, da hier manche Ausschläge sehr hoch sind und bei anderen Messrichtungen sehr gering. Bei den Stellen, wo es starke Ausschläge gibt, handelt es sich wahrscheinlich um die Objekte, die von LiDAR schlecht erfassbar sind.

1. **Schreiben Sie ein Programm, das alle Entfernungswerte eines einzelnen Laserscans aufsummiert. Erstellen Sie ein Histogramm über die Summen der Entfernungswerte des Scanners aus ihrem Bagfile. Achten Sie auf eine sinnvolle Diskretisierung des Histogramms. Interpretieren Sie die Ergebnisse.**

Code-Snippet 3 zeigt das erstellte Python-Skript und Abbildung 10 zeigt den Command zum Starten des Skripts.

#!/usr/bin/env python

import rospy

from sensor\_msgs.msg import LaserScan

import matplotlib.pyplot as plt

import numpy as np

import subprocess

import threading

sum\_values = []

def laser\_scan\_callback(data):

ranges = np.array(data.ranges)

ranges[np.isinf(ranges)] = 0

cleaned\_ranges = ranges.tolist()

sum\_values.append(round(sum(cleaned\_ranges), 1))

def play\_rosbag(bag\_file\_path):

subprocess.call(['rosbag', 'play', bag\_file\_path])

plt.hist(sum\_values, bins=len(sum\_values), edgecolor='black')

plt.xlabel('values')

plt.ylabel('frequency')

plt.title('histogram summed values')

plt.show()

rospy.init\_node('sum\_all\_Values\_from\_laser\_scan')

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

bag\_file\_path = 'src/exercise-001/laser-scanner-statistics-1-1/bagfiles/laserdata\_simulation.bag'

play\_thread = threading.Thread(target=lambda: play\_rosbag(bag\_file\_path))

play\_thread.start()

rospy.spin()

Code-Snippet - Skript für Histogramm Erstellung von aufsummierten Entfernungen

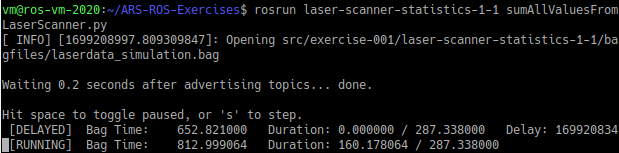


Abbildung - Command zum Starten des sumAllValuesFromLaserScanner.py Skripts

Ergebnis Simulation:

Abbildung 11 zeigt die geplotteten Werte. Da es sich hierbei um die Werte aus der Simulation handelt, sind wie erwartet keine großen Streuungen im Histogramm erkennbar. Die Werte wurden auf eine Kommastelle aufgerundet.

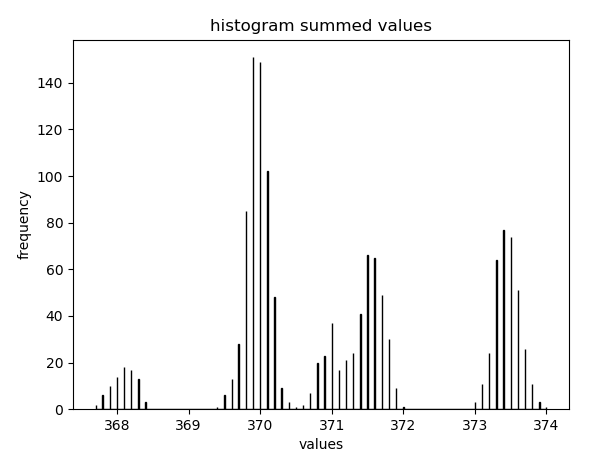


Abbildung - Histogramm der aufsummierten Entfernungen (simuliert)

Ergebnis reale Hardware:

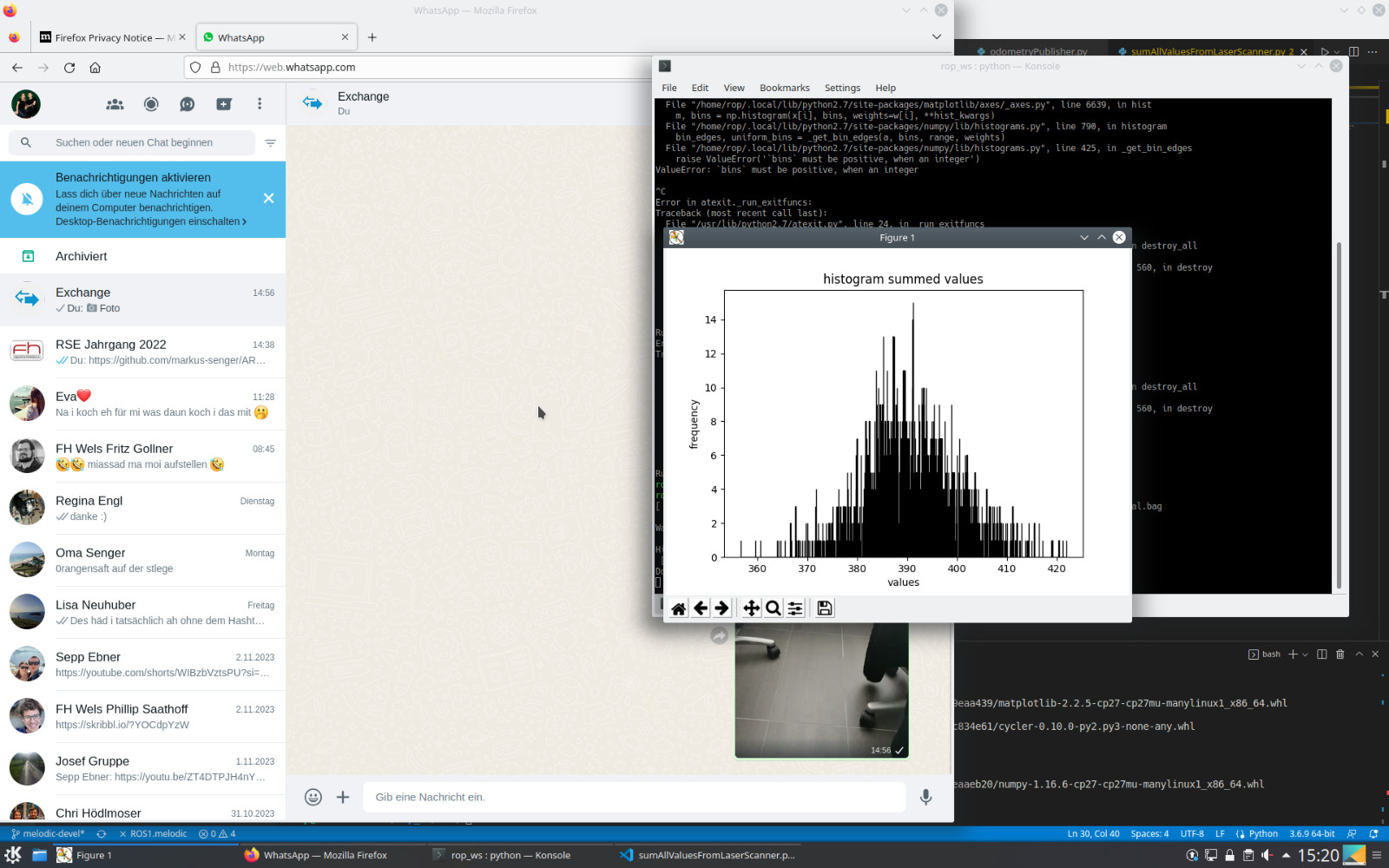


Abbildung - Histogramm der aufsummierten Entfernungen (echte Hardware)

Wie erwartet folgt das Histogramm aus Abbildung 12 einer Normalverteilung, dies lässt sich daher begründen, da sich viele zufällige Fehler immer zu einer Glockenkurve zusammenaddiert. Das passiert bei Sensoren, weil viele verschiedene Einflüsse die Messungen beeinflussen.

## Odometrie

1. **Der *turtlebot node* berechnet die Odometrie des Turtlebots.**

Abbildung 12 zeigt den Command und einen Teil der Ausgabe der Odometrie vom Turtlebot.

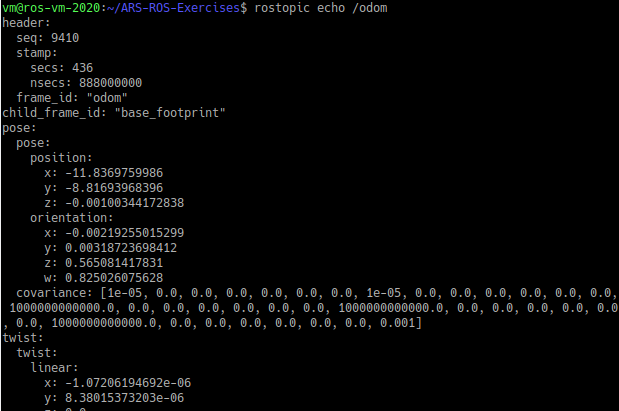


Abbildung - Odometrie die vom Turtlebot zur Verfügung gestellt wird

1. **In der Vorlesung wurden die Berechnung der Odometrie vorgestellt. Implementieren Sie diese Berechnung, um direkt aus den Sensordaten des Turtlebots die Odometrie zu bestimmen.**

Code-Snippet 4 zeigt den erstellten Code zur Odometrie-Berechnung.

#!/usr/bin/env python

import rospy

import math

from geometry\_msgs.msg import Twist, Pose

from sensor\_msgs.msg import JointState

last\_time = None

th = 0

x = 0

y = 0

def euler\_to\_quaternion(roll, pitch, yaw):

qx = math.sin(roll/2) \* math.cos(pitch/2) \* math.cos(yaw/2) - math.cos(roll/2) \*

math.sin(pitch/2) \* math.sin(yaw/2)

qy = math.cos(roll/2) \* math.sin(pitch/2) \* math.cos(yaw/2) + math.sin(roll/2) \*

math.cos(pitch/2) \* math.sin(yaw/2)

qz = math.cos(roll/2) \* math.cos(pitch/2) \* math.sin(yaw/2) - math.sin(roll/2) \*

math.sin(pitch/2) \* math.cos(yaw/2)

qw = math.cos(roll/2) \* math.cos(pitch/2) \* math.cos(yaw/2) + math.sin(roll/2) \*

math.sin(pitch/2) \* math.sin(yaw/2)

return qx, qy, qz, qw

def callback(data):

deltaTime = 0

currentTime = data.header.stamp

vl = data.velocity[0] \* 0.033

vr = data.velocity[1] \* 0.033

b = 0.1577

v = (vl + vr) / 2.0

dth = (vr - vl)/ b

global last\_time

if last\_time is not None:

deltaTime = currentTime - last\_time

last\_time = currentTime

global x

global y

global th

if deltaTime is None:

deltaTime = 0

else:

deltaTime = deltaTime.to\_sec()

x = x + (v \* deltaTime \* math.cos(th))

y = y + (v \* deltaTime \* math.sin(th))

th = th + dth \* deltaTime

pose = Pose()

pose.position.x = x

pose.position.y = y

qx, qy, qz, qw = euler\_to\_quaternion(0, 0, th)

pose.orientation.x = qx

pose.orientation.y = qy

pose.orientation.z = qz

pose.orientation.w = qw

pose\_pub.publish(pose)

rospy.init\_node('turtlebot\_pose\_calculator')

pose\_pub = rospy.Publisher('my\_odom', Pose, queue\_size=10)

joint\_sub = rospy.Subscriber('joint\_states', JointState, callback)

rospy.spin()

Code-Snippet - Skript für Odometrie-Berechnung

In den nächsten Abbildungen werden die Commands gezeigt, um das Odometrie-Skript zu testen.

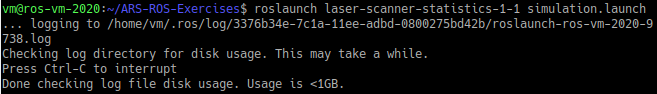


Abbildung - Command, um SImulation zu starten



Abbildung - Command der Odometrie-Skript startet



Abbildung - Command, um Roboter-Controller zu starten

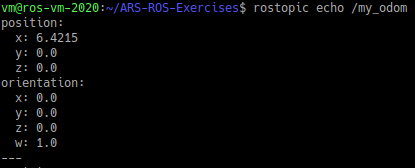


Abbildung - Topic 'my\_odom' ausgeben

1. **Schreiben Sie einen ROS-Node, der die Trajektorien der zwei Odometrievarianten in eine Datei schreibt.**

Code-Snippet 5 zeigt das Skript, um die zwei Odometrievarianten in jeweils eine Datei zu schreiben. Eventuell verstehe ich die Angabe nicht korrekt, aber für mich macht es mehr Sinn die Odometriedaten pro Variante jeweils in eine Textdatei zu schreiben.

Um ein einfaches Ausführen der Skripte zu ermöglichen, zeigt Code-Snippet 6 das erstellte Launch-File. Gazebo und der Steering-Node wird im Launch-File nur benötigt, um das Skript vorab in der Simulation zu testen. Mit der echten Hardware wurde ein Bagfile aufgenommen und dieses abgespielt.

#!/usr/bin/env python

import rospy

from nav\_msgs.msg import Odometry

from geometry\_msgs.msg import Pose

import os

def odom\_callback(data):

global trajectory\_file\_odom

pose = data.pose.pose

write\_to\_file(trajectory\_file\_odom, pose)

def my\_odom\_callback(data):

global trajectory\_file\_my\_odom

write\_to\_file(trajectory\_file\_my\_odom, data)

def write\_to\_file(file, pose):

x = pose.position.x

y = pose.position.y

z = pose.position.z

qx = pose.orientation.x

qy = pose.orientation.y

qz = pose.orientation.z

qw = pose.orientation.w

file.write("{} {} {} {} {} {} {}\n".format(x, y, z, qx, qy, qz, qw))

rospy.init\_node('my\_trajectory\_writer')

odom\_topic = 'odom'

my\_odom\_topic = 'my\_odom'

try:

rospy.wait\_for\_message(odom\_topic, Odometry, timeout=100)

rospy.wait\_for\_message(my\_odom\_topic, Pose, timeout=100)

except rospy.ROSException:

rospy.logerr("error")

odom1\_sub = rospy.Subscriber(odom\_topic, Odometry, odom\_callback)

odom2\_sub = rospy.Subscriber(my\_odom\_topic, Pose, my\_odom\_callback)

trajectory\_file\_odom = open(rospy.get\_param('odom\_file\_path'), 'w+')

trajectory\_file\_my\_odom = open(rospy.get\_param('my\_odom\_file\_path'),

'w+')

rospy.spin()

Code-Snippet - Skript, um Odometrie in Textdateien zu schreiben

<launch>

<!-- Arguments -->

<arg name="model" default="$(env TURTLEBOT3\_MODEL)" doc="model type

[burger, waffle, waffle\_pi]"/>

<arg name="map\_file" default="$(find laser-scanner-statistics-1-

1)/world/maze.world"/>

<arg name="open\_rviz" default="true"/>

<arg name="open\_gazebo" default="false"/>

<arg name="x\_pos" default="-14"/>

<arg name="y\_pos" default="-10"/>

<arg name="z\_pos" default="0.0"/>

<!-- Turtlebot3 -->

<include file="$(find turtlebot3\_bringup)/launch/turtlebot3\_remote.launch">

<arg name="model" value="$(arg model)" />

</include>

<param name="robot\_description" command="$(find xacro)/xacro --inorder $(find

turtlebot3\_description)/urdf/turtlebot3\_$(arg model).urdf.xacro" />

<!-- rviz -->

<group if="$(arg open\_rviz)">

<node pkg="rviz" type="rviz" name="rviz" required="true"

args="-d $(find

turtlebot3\_navigation)/rviz/turtlebot3\_navigation.rviz -f

base\_link"/>

</group>

<!-- gazebo -->

<include file="$(find gazebo\_ros)/launch/empty\_world.launch">

<arg name="world\_name" value="$(arg map\_file)"/>

<arg name="paused" value="false"/>

<arg name="use\_sim\_time" value="true"/>

<arg name="gui" value="true"/>

<arg name="headless" value="false"/>

<arg name="debug" value="false"/>

</include>

<group if="$(arg open\_gazebo)">

<node pkg="gazebo\_ros" type="spawn\_model" name="spawn\_urdf"

args="-urdf -model turtlebot3\_$(arg model) -x $(arg x\_pos) -y $(arg

y\_pos) -z $(arg z\_pos) -param robot\_description" />

</group>

<node name="steering\_node" pkg="rqt\_robot\_steering" type="rqt\_robot\_steering"

output="screen"/>

<node name="my\_odom\_node" pkg="my-odometry-1-2" type="myOdom.py"

output="screen"/>

<param name="odom\_file\_path" type="string" value="$(find my-odometry-1-

2)/trajectoryTextfiles/trajectory\_odom.txt" />

<param name="my\_odom\_file\_path" type="string" value="$(find my-odometry-1-

2)/trajectoryTextfiles/trajectory\_my\_odom.txt" />

<node name="my\_trajectory\_writer\_node" pkg="my-odometry-1-2"

type="trajectoryWriter.py" output="screen"/>

</launch>

Code-Snippet – Launchfile für Vergleich von zwei Odometrievarianten

1. **Fahren Sie mit dem Roboter eine Trajektorie ab und beschreiben Sie die gefahrene Trajektorie.**

Abbildung 19 zeigt grafisch die gefahrene Trajektorie.

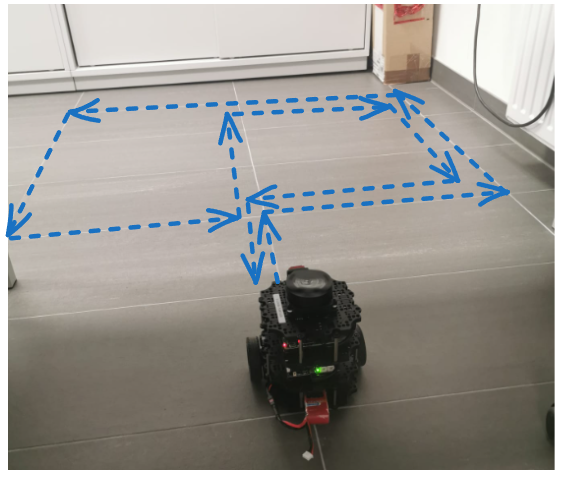


Abbildung - Gefahrene Trajektorie

1. **Plotten die zwei Odometrie-Varianten. Stellen Sie anhand mehrerer signifikanten Punkte die Unterschiede hervor und vergleichen Sie dies auch mit der tatsächlich gefahrenen Trajektorie.**

Zunächst wird RViz gestartet und zwei Odometry-Einträge hinzugefügt. Ein Eintrag hört auf das Topic */odometry\_data* und das andere auf */odometry\_data2*. Anschließend werden durch einen Node die Einträge in den zuvor erstellten Textdateien auf die entsprechenden Topics gepublished. Code-Snippet 7 zeigt diesen Node, der einfachhalber wurden die Startwerte der Odometrie-Aufzeichnungen vom Turtlebot3 hartkodiert auf 0 gesetzt.

#!/usr/bin/env python

import rospy

from nav\_msgs.msg import Odometry

def odometry\_publisher():

rospy.init\_node('odometry\_publisher', anonymous=True)

pub = rospy.Publisher('odometry\_data', Odometry, queue\_size=10)

pub2 = rospy.Publisher('odometry\_data2', Odometry, queue\_size=10)

rate = rospy.Rate(10)

with open('src/exercise-001/my-odometry-1-

2/trajectoryTextfiles/trajectory\_my\_odom.txt', 'r') as file:

for line in file:

parts = line.strip().split()

if len(parts) == 7:

odometry\_msg = Odometry()

odometry\_msg.header.frame\_id = "base\_link"

odometry\_msg.pose.pose.position.x = float(parts[0])

odometry\_msg.pose.pose.position.y = float(parts[1])

odometry\_msg.pose.pose.position.z = float(parts[2])

odometry\_msg.pose.pose.orientation.x = float(parts[3])

odometry\_msg.pose.pose.orientation.y = float(parts[4])

odometry\_msg.pose.pose.orientation.z = float(parts[5])

odometry\_msg.pose.pose.orientation.w = float(parts[6])

pub.publish(odometry\_msg)

rate.sleep()

with open('src/exercise-001/my-odometry-1-

2/trajectoryTextfiles/trajectory\_odom.txt', 'r') as file:

for line in file:

parts = line.strip().split()

if len(parts) == 7:

odometry\_msg = Odometry()

odometry\_msg.header.frame\_id = "base\_link"

odometry\_msg.pose.pose.position.x = -float(parts[0])

- 0.727819979191

odometry\_msg.pose.pose.position.y = -float(parts[1])

- -0.142933622003

odometry\_msg.pose.pose.position.z = -float(parts[2])

odometry\_msg.pose.pose.orientation.x = float(parts[3])

odometry\_msg.pose.pose.orientation.y = float(parts[4])

odometry\_msg.pose.pose.orientation.z = float(parts[5])

- 0.0376496426761

odometry\_msg.pose.pose.orientation.w = float(parts[6])

- 0.99929100275

pub2.publish(odometry\_msg)

rate.sleep()

odometry\_publisher()

Code-Snippet – Node zum Publishen der zwei Odometrievarianten

Prinzipiell hat die Odometrieberechnung gut funktioniert, wie man anhand Abbildung 20 erkenne kann. Hierbei ist die blaue Line die Odometrie vom Turtlebot und die rote Linie die eigene Odometrieberechnung. Die Odometrie vom Turtlebot funktioniert besser, dies liegt aber daran, dass der Turtlebot hierbei auch die Sensordaten für die Berechnung berücksichtigt. Lediglich bei der ersten Kurve dreht die eigene Odometrieberechung zu weit, ansonsten könnten diese Odometriedaten durchaus mit der Odometriedaten vom Turtlebot mithalten. Im Vergleich zur tatsächlich gefahrenen Trajektorie ist die Odometrie vom Turtlebot auch gut, lediglich die erste Kurve wird etwas zu früh durchgeführt.

Abbildung - Vergleich Odometrie-Varianten

