# 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.

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ rostopic echo /scan
header:
  seq: 1695
  stamp:
    secs: 462
    nsecs: 697000000
  frame_id: "base_scan"
angle_min: 0.0
angle_max: 6.28318977356
angle_increment: 0.0175019223243
time_increment: 0.0
scan_time: 0.0
range_min: 0.119999997318
range_max: 3.5
ranges: [inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
 inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, 2.190685272216797, 2.1
01038694381714, 2.024897575378418, 1.9770562648773193, 1.982537865638733, 1.927798748016
3574, 1.9282212257385254, 1.9175482988357544, 1.8988555669784546, 1.9093867540359497, 1.
8629980087280273, 1.8893554210662842, 1.874774694442749, 1.8965240716934204, 1.884004354
4769287, 1.9169368743896484, 1.9173084497451782, 1.9219475984573364, 1.9310753345489502,
 1.954102873802185, 1.9965500831604004, 2.0424163341522217, 2.0997812747955322, 2.190154
3140411377, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, i
nf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
 inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, in
f, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf,
inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf
, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, inf, 3.049172639
```

*Abbildung 1 - Ausgabe des Topics '/scan'*

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ rosmsg show sensor_msgs/LaserScan
std_msgs/Header header
  uint32 seq
  time stamp
  string frame_id
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

*Abbildung 2 - 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.

```python
#!/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 1 - Python-Skript zum Bestimmen der Anzahl der Messwerte*

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ rosrun laser-scanner-statistics-1-1 checkNumbersOfMe
asurements.py
[INFO] [1699179989.638143]: Anzahl der Messwerte in 'ranges': 360
```

*Abbildung 3 - 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.

2. **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.

```
vm@ros-vm-2020:~$ rosbag record scan
[ INFO] [1699199205.738768304]: Subscribing to scan
[ INFO] [1699199206.070836980, 652.762000000]: Recording to '2023-11-05-16-46-46.bag'.
```

*Abbildung 4 - 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.

3. **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.

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ rosbag play src/exercise-001/laser-scanner-statistic
s-1-1/bagfiles/laserdata_simulation.bag
[ INFO] [1699200463.106020826]: Opening src/exercise-001/laser-scanner-statistics-1-1/ba
gfiles/laserdata_simulation.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
 [RUNNING]  Bag Time:    660.654095   Duration: 7.833095 / 287.338000
```

*Abbildung 5 - 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.

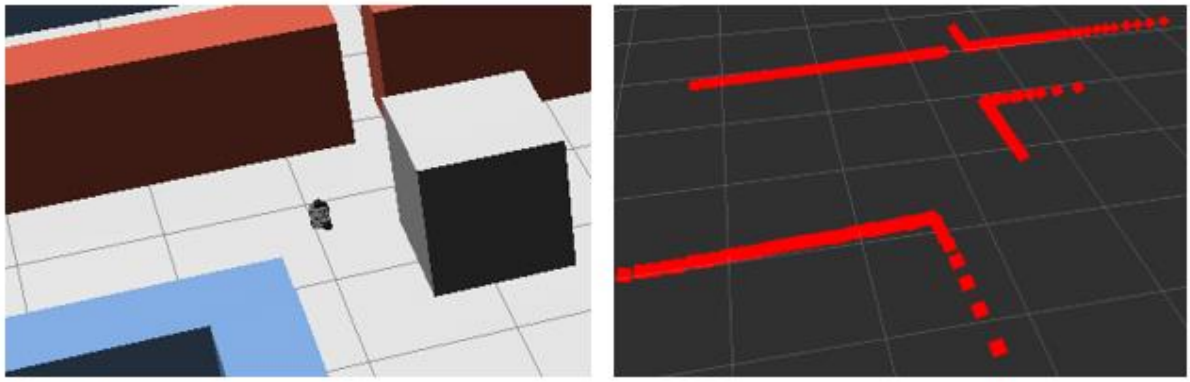

*Abbildung 7 - Simulationsumgebung (links) und visualisierte Daten in RViz (rechts)*

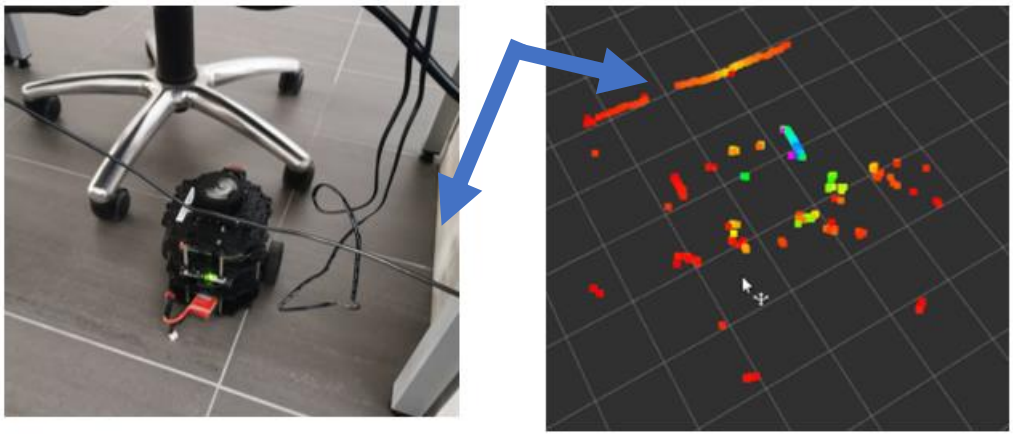

*Abbildung 6 - Reale Umgebung (links) und visualisierte Daten in RViz (rechts)*

4. **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.

```python
#!/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 2 - 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.

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ rosrun laser-scanner-statistics-1-1 statisticsForLaserSca
nner.py
[ INFO] [1699691245.953923791]: Opening src/exercise-001/laser-scanner-statistics-1-1/bagfile
s/laserdata_simulation.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
[RUNNING]  Bag Time:    656.441192   Duration: 3.620192 / 287.338000              90593.36
```

*Abbildung 8 - 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.

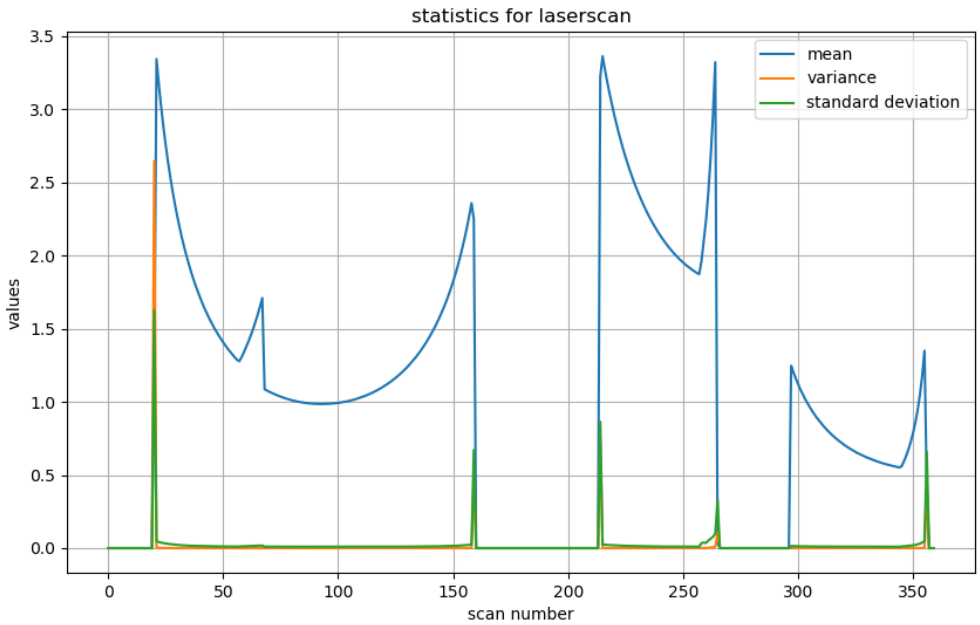


*Abbildung 9 - Laserscanner-Statistik aus der Simulation*

Ergebnis reale Hardware:

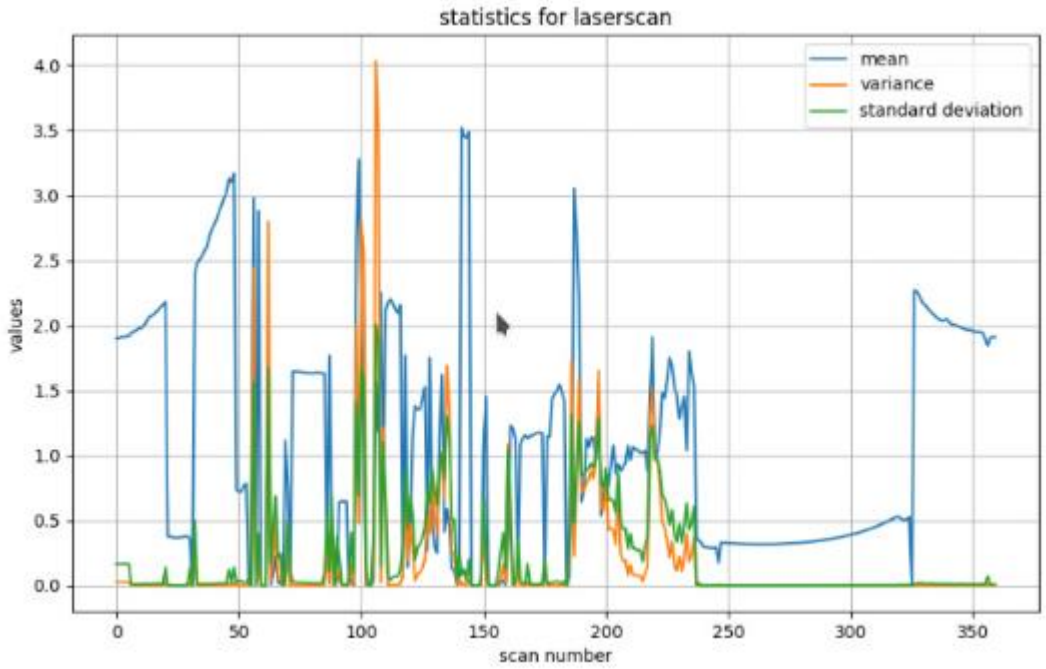


*Abbildung 10 - Laserscanner-Statistik aus der realen Hardware*

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.

5. **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 11 zeigt den Command zum Starten des Skripts.

```python
#!/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 3 - Skript für Histogramm Erstellung von aufsummierten Entfernungen*

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ rosrun laser-scanner-statistics-1-1 sumAllValuesFrom
LaserScanner.py
[ INFO] [1699208997.809309847]: Opening src/exercise-001/laser-scanner-statistics-1-1/ba
gfiles/laserdata_simulation.bag

Waiting 0.2 seconds after advertising topics... done.

Hit space to toggle paused, or 's' to step.
 [DELAYED]  Bag Time:    652.821000   Duration: 0.000000 / 287.338000   Delay: 169920834
[RUNNING]  Bag Time:    812.999064   Duration: 160.178064 / 287.338000
```

*Abbildung 11 - Command zum Starten des sumAllValuesFromLaserScanner.py Skripts*

Ergebnis Simulation:

Abbildung 12Abbildung 9 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.

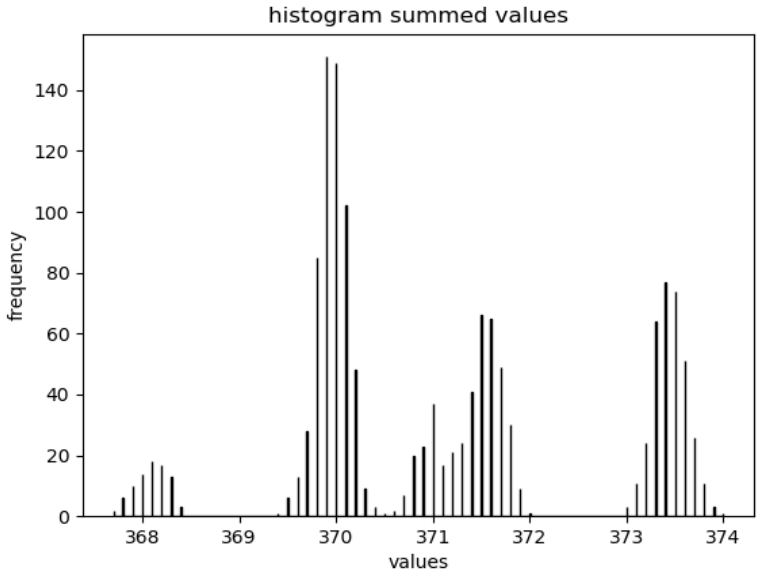


*Abbildung 12 - Histogramm der aufsummierten Entfernungen (simuliert)*

Ergebnis reale Hardware:

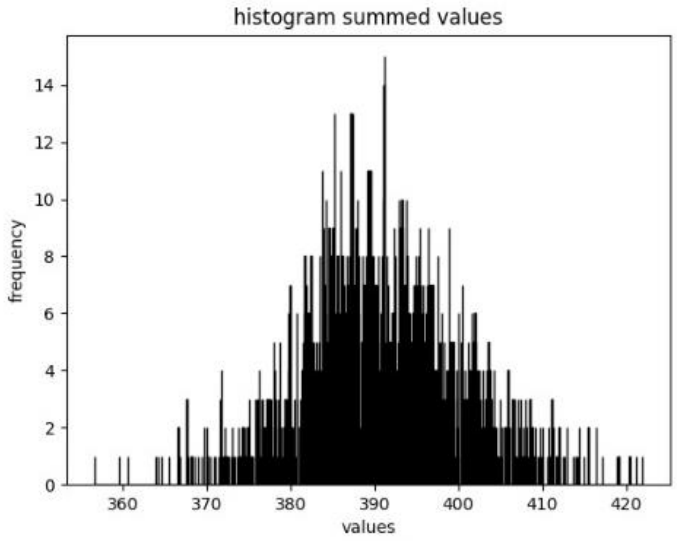


*Abbildung 13 - Histogramm der aufsummierten Entfernungen (echte Hardware)*

Wie erwartet folgt das Histogramm aus Abbildung 13 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 14 zeigt den Command und einen Teil der Ausgabe der Odometrie vom Turtlebot.

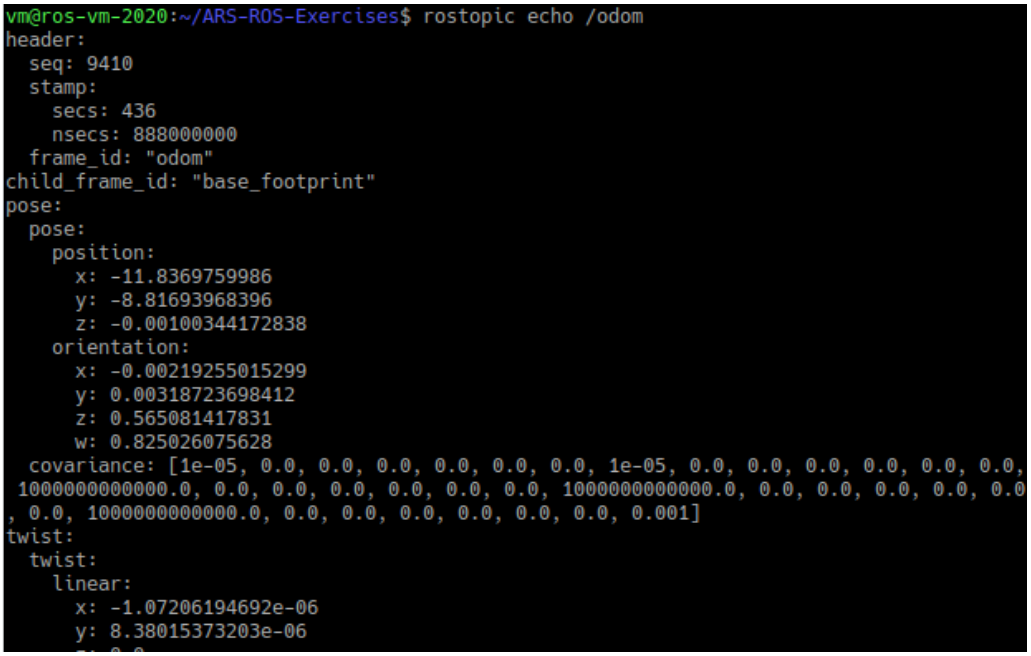


*Abbildung 14 - Odometrie die vom Turtlebot zur Verfügung gestellt wird*

2. **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.

```python
#!/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 4 - Skript für Odometrie-Berechnung*

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

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ roslaunch laser-scanner-statistics-1-1 simulation.launch
... logging to /home/vm/.ros/log/3376b34e-7c1a-11ee-adbd-0800275bd42b/roslaunch-ros-vm-2020-9
738.log
Checking log directory for disk usage. This may take a while.
Press Ctrl-C to interrupt
Done checking log file disk usage. Usage is <1GB.
```

*Abbildung 15 - Command, um SImulation zu starten*

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ rosrun my-odometry-1-2 myOdom.py
```

*Abbildung 16 - Command der Odometrie-Skript startet*

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ rosrun rqt_robot_steering rqt_robot_steering
```

*Abbildung 17 - Command, um Roboter-Controller zu starten*

```
vm@ros-vm-2020:~/ARS-ROS-Exercises$ rostopic echo /my_odom
position:
  x: 6.4215
  y: 0.0
  z: 0.0
orientation:
  x: 0.0
  y: 0.0
  z: 0.0
  w: 1.0
---
```

*Abbildung 18 - Topic 'my_odom' ausgeben*

3. **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.

```python
#!/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 5 - 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 6 – Launchfile für Vergleich von zwei Odometrievarianten*

4. **Fahren Sie mit dem Roboter eine Trajektorie ab und beschreiben Sie die gefahrene Trajektorie.**
   Abbildung 19 zeigt grafisch die gefahrene Trajektorie.

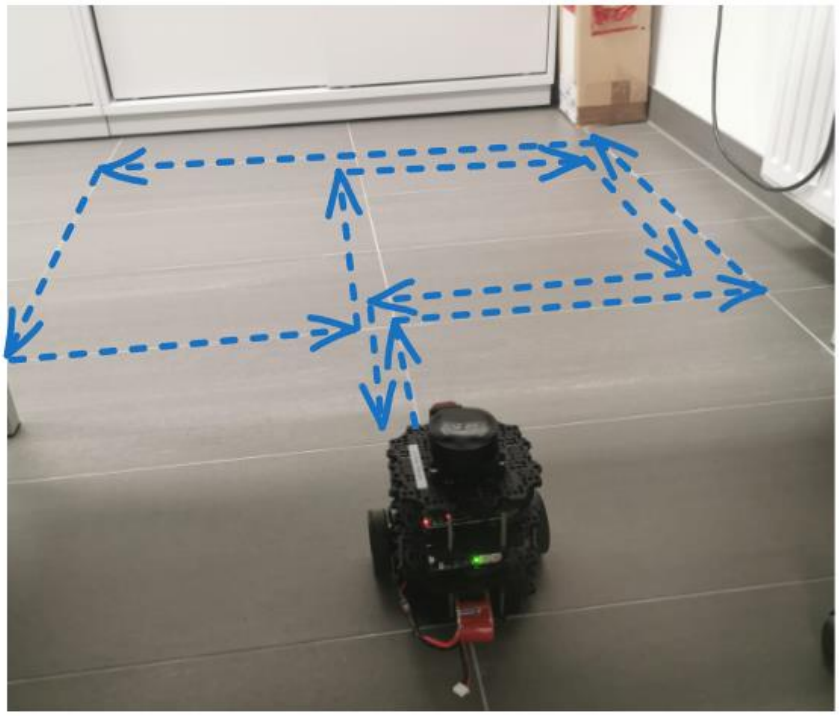

*Abbildung 19 - Gefahrene Trajektorie*

5. **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.

```python
#!/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 7 – 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.

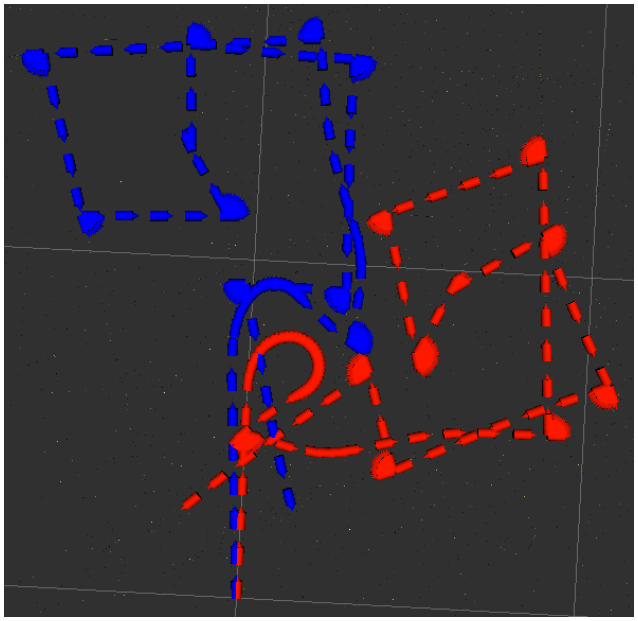

*Abbildung 20 - Vergleich Odometrie-Varianten*