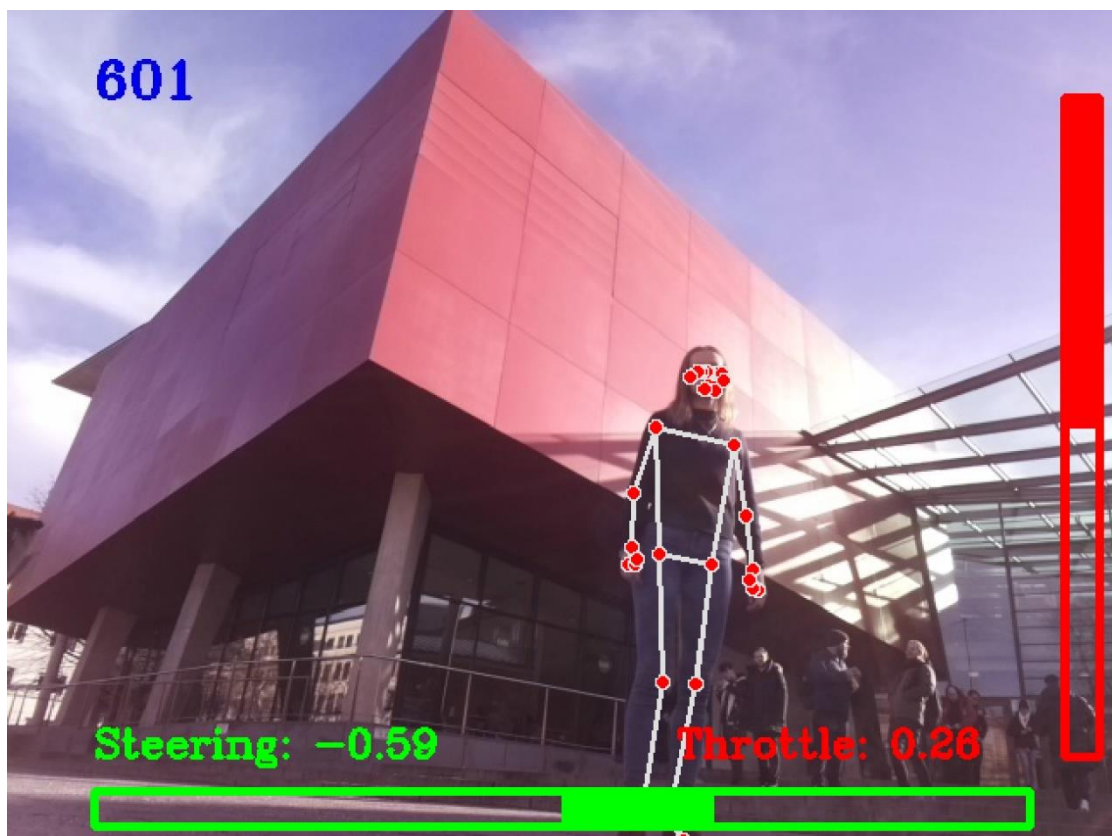


Einführung in ROS2

Projekt Autonome Systeme
16.01.2023



Inhaltsverzeichnis

Inhalt

Inhaltsverzeichnis	2
Abkürzungsverzeichnis	4
1 Aufgabenstellung	5
2 Grundlagen	6
2.1 Turtlebot3.....	6
2.2 ROS2	8
2.2.1 Grundsätzliches	8
2.2.2 Nodes	8
2.2.3 Topics	8
2.2.4 Messages.....	8
2.2.5 Packages	9
2.2.6 Zusammenfassung wichtiger ROS2 bzw. Linux-Kommandos	10
2.3 Python Programmierung.....	11
2.3.1 IntelliSense	11
2.3.2 Anmerkung Debugging	13
2.4 OpenCV und MediaPipe.....	14
3 Verbindung zum Turtlebot aufbauen.....	15
4 Teleoperation Node	17
4.1 Terminals.....	17
4.2 Turtlebot3 BringUp	17
4.3 Teleoperation Node	17
4.4 Weitere Informationen.....	18
5 Aufbau einer Node.....	19
6 Programmierung Teil 1 – Picture Publisher	21
6.1 Grundsätzliches	21
6.2 Programmieren der Node	21
6.2.1 Init -Funktion.....	21
6.2.2 Callback-Funktion	23
6.3 Programmieren der Main-Methode	24
6.4 Testen der Node.....	25

7	Programmierung Teil 2 - Picture Processor	26
7.1	Grundsätzliches	26
7.2	Programmieren der Node	27
7.2.1	Init-Funktion.....	27
7.2.2	Callback-Funktion	29
7.3	Main-Funktion.....	30
	Abbildungsverzeichnis.....	31

Abkürzungsverzeichnis

cmd vel	command velocity (eine ROS Topic)
LASIM	L aboratory for A utonomous S ystems I n M unich
LiDAR	L ight D etection A nd R anging
OpenCR	O pen-source Control module for R OS
QoS	Q uality of S ervice
ROS	R obot O perating S ystem
SSH	S ecure S hell
VSCoDe	V isual S tudio C ode

Anmerkung: In diesem Dokument befinden sich teilweise grobe denglische Sätze wie „Nodes kommunizieren miteinander über das publishen und subscriben von Messages in Topics“. Dies liegt nicht etwa daran, dass die Autoren nach dem Abi in Australien gewesen wären, sondern soll dem Auftrag dienen, eine zielorientierte, lehrreiche Anleitung statt einem wissenschaftlichen Dokument zu schreiben. So müssen die neu erlernten, englischen Funktionen nicht ständig im Kopf aus den ungewohnten Übersetzungen erraten werden.

1 Aufgabenstellung

Ziel ist es einen Roboter so zu programmieren, dass er einem Menschen folgen kann. Dies soll über eine Kamera, mit der der Roboter Bilder seiner Umgebung aufzeichnen kann, geschehen. In diesen Bildern soll maschinell ein Mensch und seine Position relativ zum Roboter erkannt werden. Auf Basis dieser Auswertungen sollen die Elektromotoren des Motors so angesteuert werden, dass der Roboter sich zu dem Menschen ausrichtet und diesem folgt.

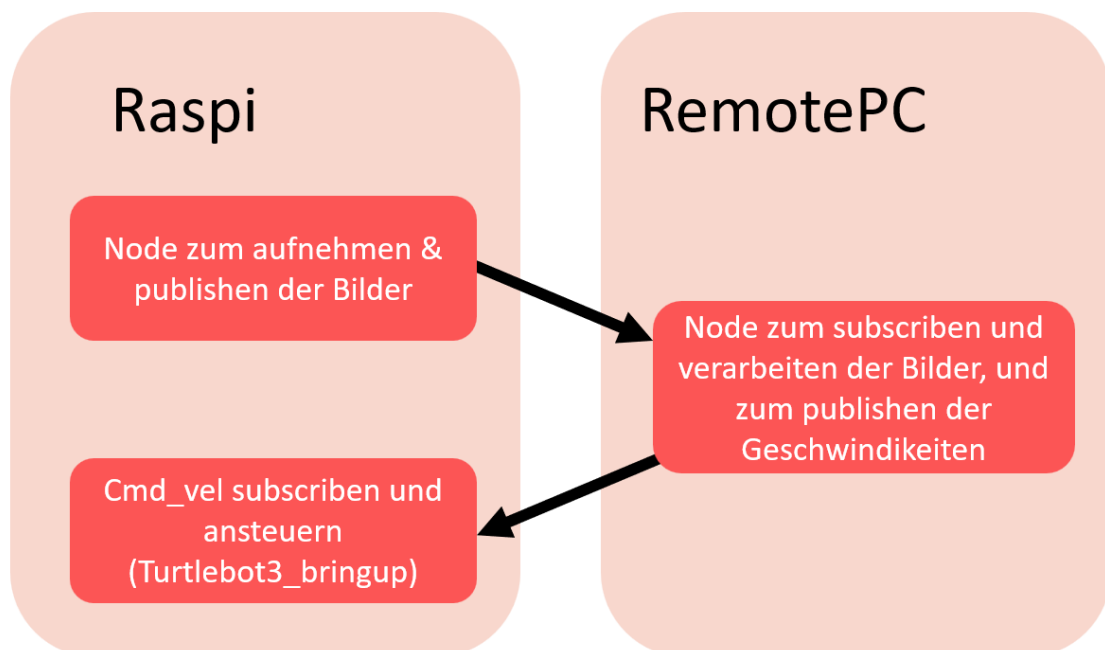


Abbildung 1-1: Übersicht der Kommunikationsstruktur

2 Grundlagen

2.1 Tuttlebot3

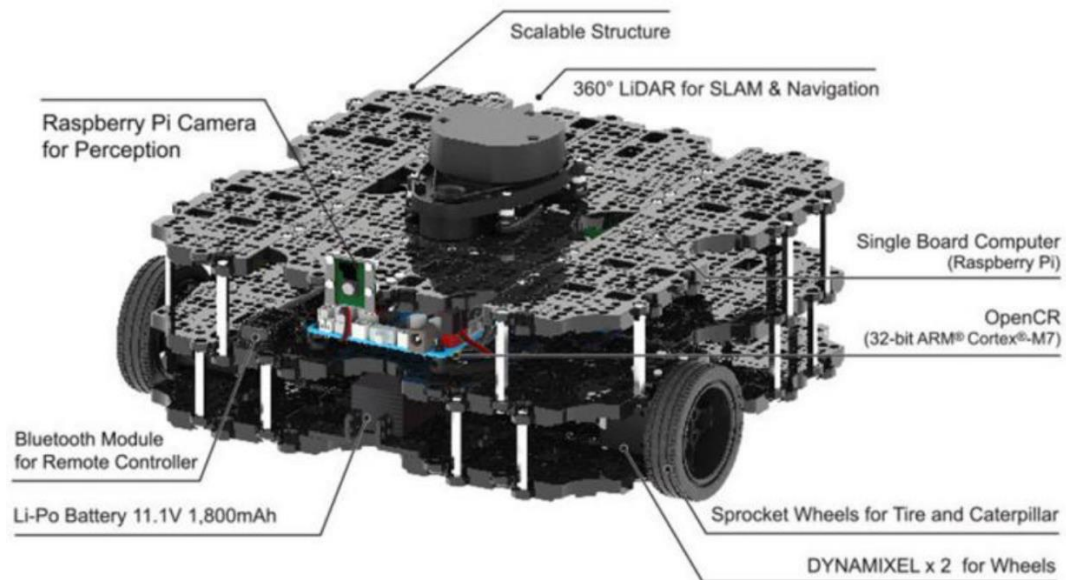


Abbildung 2-1: Turtlebot 3 Waffle Pi¹

Als Roboter wird ein Turtlebot3 in der flachen „Waffle Pi“ Konfiguration verwendet. Zentrales Element des Turtlebots ist der Einplatinencomputer Raspberry Pi 4b. Dieser läuft mit Ubuntu und verfügt er über verschiedene Sensoren, wie einen 2D-LiDAR und eine Kamera. Zudem besitzt er Aktoren zum Ansteuern der Elektromotoren der Antriebe. Die Sensoren und Aktoren werden über ein OpenCR1.0 Bord angesteuert, welches mit dem Raspberry Pi verbunden ist. Das OpenCR1.0 Board wird daher nicht umprogrammiert, da es keine Verhaltenslogik besitzt, sondern nur Steuerbefehle umsetzt.

Um den Roboter zu programmieren werden daher Programme auf dem Raspberry Pi geschrieben und ausgeführt. Um auf diesem Arbeiten zu können, lässt sich entweder ein Bildschirm und eine Tastatur am Raspberry Pi anschließen, dies ist aber unpraktisch, gerade wenn sich der Roboter bewegt. Das Ubuntu System des Raspberry Pis ist standardmäßig so konfiguriert, dass man mit einer Secure Shell (SSH) Verbindung von einem anderen Rechner, der sich im selben Subnetz (IPv4: 192.168.<Subnetz, z.B.: 178>.42) befindet, darauf arbeiten kann.

¹ https://www.researchgate.net/publication/352675079_Comparison_of_Two_SLAM_Algorithms_Provided_by_ROS_Robot_Operating_System

Die einzige nötige Hardwaremodifikation am Turtlebot3 für Ihre Aufgabenstellung ist das Biegen der Kamerahalterung auf einen Winkel von ca. 60°, damit die Kamera auch nahestehende Menschen erkannt werden. Dafür wurde eine der beiden mitgelieferten Kamerahalterungen im Schraubstock unter Hitze nach hinten gebogen.



Abbildung 2-2: Biege-Modifikation der Kamerahalterung am Turtlebot3

2.2 ROS2

2.2.1 Grundsätzliches

Das Robot Operating System (ROS) ist eine Open Source Software zum Betreiben von Robotersystemen. Es lässt sich entgegen seinem Namen nicht den Betriebssystemen, sondern eher der Gruppe der Middlewares, einer Zwischenebene zwischen dem Betriebssystem und der eigentlichen Anwendung, zuordnen. Diese Zwischenebene vereinfacht die Kompatibilität, da Hardwareentwickler sich nicht direkt mit den Anwendungen und Anwendungsentwickler nicht mit der Hardware beschäftigen müssen. Es ist der De-Facto-Standard in der Forschung und für nicht kommerzielle Robotik Projekte und wird auch in einem Großteil der Roboter im LASIM verwendet. Mit ROS2, einer Weiterentwicklung von ROS werden viele Normen und Grundsätze erfüllt, sodass auch Industrieanwendungen möglich sind. Da sich ROS2 noch ein wenig in den Kinderschuhen befindet, wird ROS1 in Spezialfällen noch benutzt, die Zukunft gehört allerdings ROS2, daher wird in dieser Anleitung ROS2 gelehrt. Die beiden Systeme fühlen sich zwar ähnlich an, sind aber grundsätzlich anders aufgebaut und daher auch nicht rückwärts kompatibel. In diesem Dokument ist mit ROS immer ROS2 genannt, sofern nicht explizit ROS1 erwähnt wird.

2.2.2 Nodes

Nodes sind ein wichtiger Baustein von ROS. Damit sind Programme gemeint, die von ROS ausgeführt werden können und die ROS-Infrastruktur nutzen. In ihnen wird die eigentliche Logik, bzw. das Verhalten des Roboters implementiert. Es können Sensoren ausgelesen, ausgewertet, Aktoren angesteuert und viele weitere Funktionen umgesetzt werden.

2.2.3 Topics

Als Kommunikationsprinzip basiert ROS auf dem Publisher-Subscriber Prinzip. Verschiedene Nodes kommunizieren miteinander über das Publishen von Messages in Topics. Auf diese Topics können dann andere Nodes zugreifen, indem Sie das Topic subscriben. Mehrere Nodes können so über Topics miteinander kommunizieren, ohne dass sie etwas übereinander wissen müssen.

2.2.4 Messages

Messages könnte man auch als Datentypen der Topics bezeichnen. Nodes publishen Daten in Form von bestimmten Messages, damit die Subscriber wissen, wie sie die Daten im Topic interpretieren müssen.

2.2.5 Packages

In Packages können mehrere Programme (z.B. Nodes) zusammengefasst und auf einmal aufgerufen werden. Ein Beispiel für ein Package ist das Turtlebot3 BringUp, das Sie im Folgenden noch kennenlernen werden.

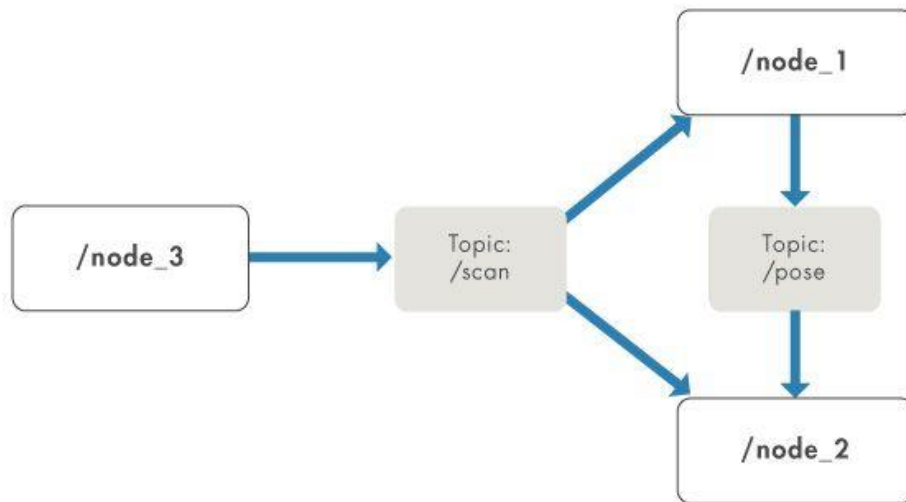


Abbildung 2-3: Überblick über ein ROS Netz²

² <https://de.mathworks.com/products/ros.html>

2.2.6 Zusammenfassung wichtiger ROS2 bzw. Linux-Kommandos

1. SSH-Verbindung zum Raspi aufbauen (auch in Windows CMD verwendbar)

```
$ ssh ubuntu@192.168.21.184
```

2. Turtlebot3 BringUp starten

```
$ ros2 launch turtlebot3_bringup robot.launch.py
```

3. Teleoperation Node starten (Kapitel 4)

```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

4. Liste alle Topics anzeigen

```
$ ros2 topic list
```

5. Werte in einem Topic ausgeben

```
$ ros2 topic echo \name_der_topic
```

6. ROS-Packages builden und ausführen

- a. In den einen Workspace gehen, z.B. picpub_ws

```
$ cd ~/AutoSys-TurtlebotRos2/picpub_ws
```

- b. Dependencies updaten und installieren

```
$ rosdep update
```

```
$ rosdep install -i --from-path src --rosdistro foxy -y
```

- c. Das Package mit colcon kompilieren

```
$ colcon build
```

- d. Alle notwendigen, kompilierten Dateien des Packages ROS bekanntmachen

```
$ . install/setup.bash
```

- e. Eine Node des Packages ausführen

```
$ ros2 run PACKAGENAME NODENAME
```

2.3 Python Programmierung

ROS Nodes können mit C++ oder Python programmiert werden. In dieser Einführung verwenden Sie Python in der Version 3.8.10, da es für den Anfang etwas übersichtlicher und einfacher zu nutzen ist, auch wenn Ihre Programme dadurch langsamer laufen.

2.3.1 IntelliSense

Die Python Extension für VSCode bringt IntelliSense mit, die im Reiter *Extensions* der Sidebar von VSCode installiert wurden. Folgende Funktionen können Sie nutzen.

„Hover“ Sie mit der Maus über den Namen einer Funktion oder Klasse, um mehr Informationen über diese zu erhalten:

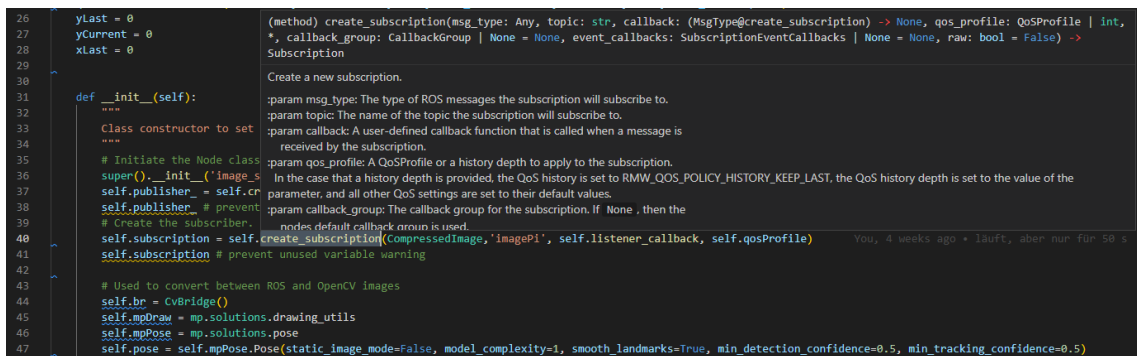


Abbildung 2-4: IntelliSense: Hover Information der Methode `create_subscription`

Die Autovervollständigung erscheint oft an passender Stelle von allein, Sie können sie jedoch auch jederzeit über Strg + Leertaste aufrufen:

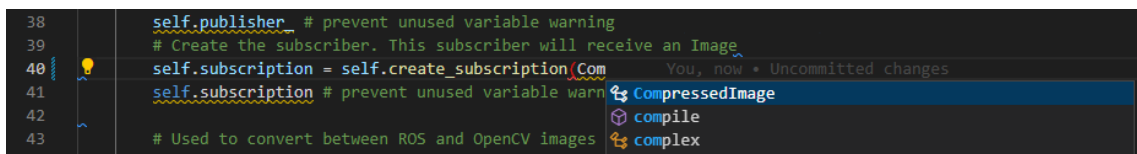


Abbildung 2-5: IntelliSense Autovervollständigung

Interessiert Sie etwas aus einer Bibliothek, klicken Sie mit der rechten Maustaste darauf und wählen Sie *Go to Definition* oder *Peek* → *Peek Definition* an. Hier am Beispiel der `create_subscription`-Methode:

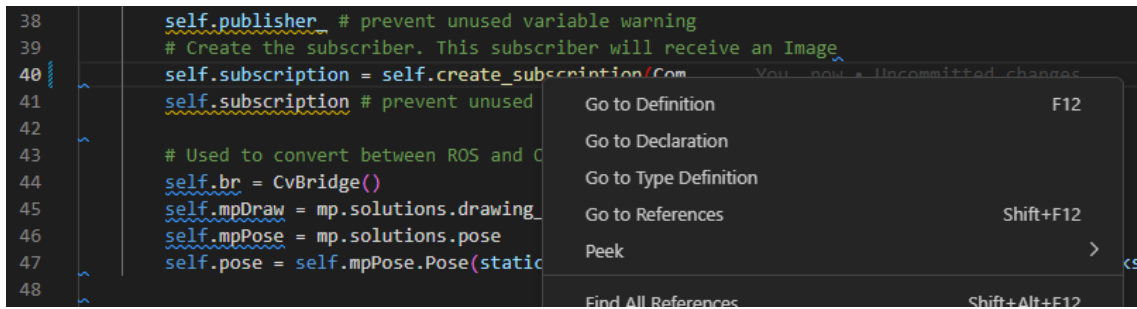


Abbildung 2-6: IntelliSense Go to Definition / Peek → Definition

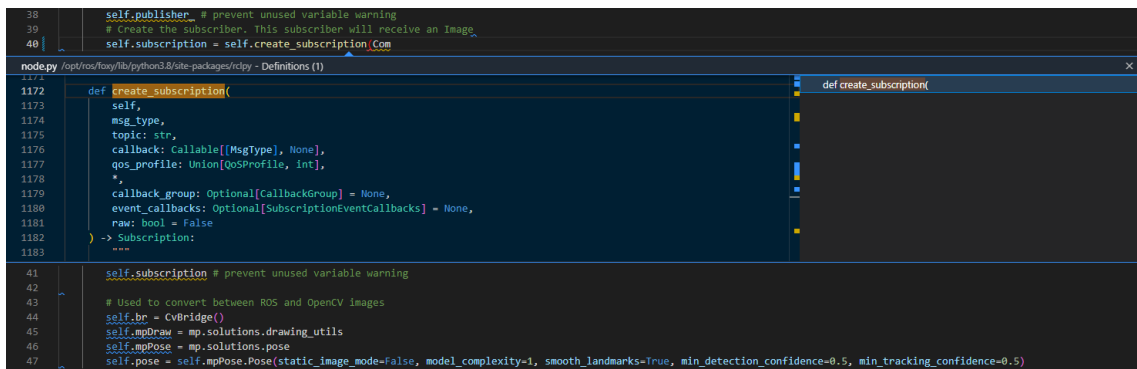


Abbildung 2-7: IntelliSense Definition Peek

2.3.2 Anmerkung Debugging

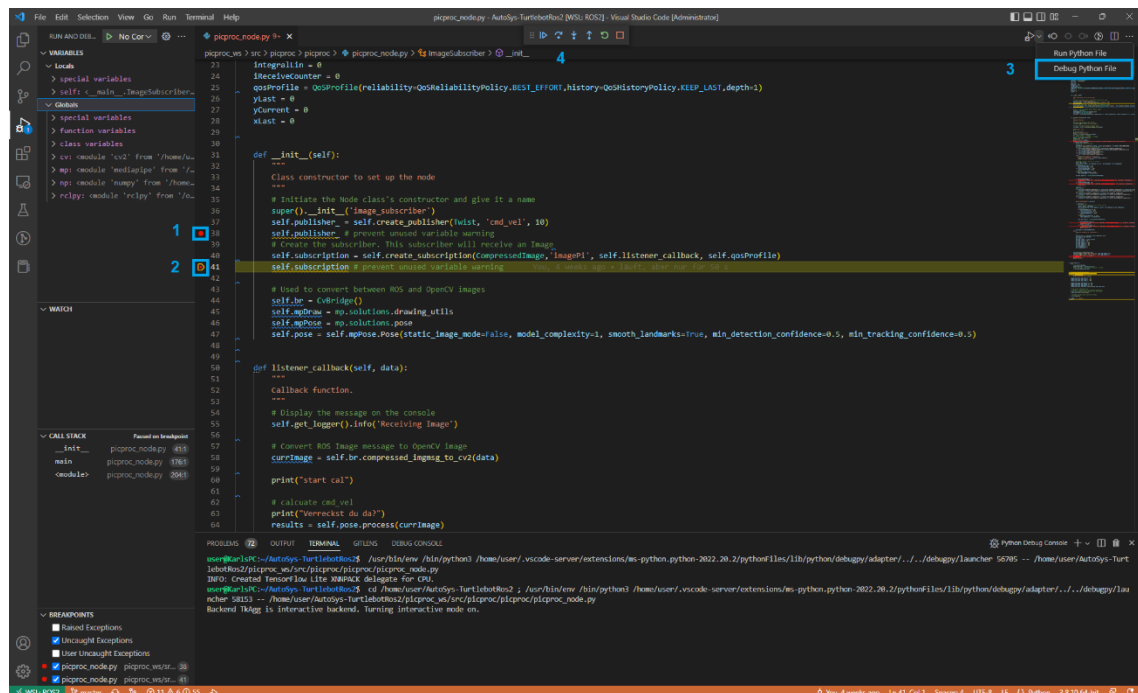


Abbildung 2-8: Debugging von Python Programmen in VSCode

Zum Testen können Sie Ihr Programm statt normal über ROS (siehe 6.4) auch „nur“ als Python Programm über den Pfeil (#3) oben rechts starten. Wenn Sie den Pfeil nicht haben, fehlt Ihnen die Python Extension für VSCode. Dort haben Sie neben der Option zum normalen Ausführen (Pfeil) auch die Option zum Debuggen (Pfeil mit Käfer, #3).

Vorher sollten Sie links neben der Leiste, in der die Zeilennummer steht, durch Klicken Breakpoints hinzufügen (#1, #2). Kommt das Programm an dieser Stelle an, wird es pausiert und die aktuelle Zeile wird gelb markiert (#2). Sie können nun mit der Leiste (#4) oben das Programm wieder weiterlaufen lassen, oder die Unterabläufe Schritt für Schritt abarbeiten.

Falls Sie unter Debugging die Ausgabe von Text in die Konsole verstehen, ist dies ganz normal mit dem Befehl `print(„Text“)` möglich. Was allerdings auch im Terminal ausgegeben wird, parallel aber auch von ROS mitgeloggt wird, ist die Methode `get_logger().info(„Text“)` jeder Node Klasse.

```
self.get_logger().info(f"Publishing ang: {self.msg.angular.z}")
```

Variablen lassen sich wie in dem Beispiel mit der f-string Schreibweise ausgeben.

2.4 OpenCV und MediaPipe

OpenCV und MediaPipe sind Bibliotheken, die für das Aufzeichnen und Auswerten von Bildern verwendet werden. OpenCV bietet Funktionen, um mit Bildern zu arbeiten, beispielsweise diese aufzunehmen, zu speichern oder anzuzeigen und Ihre Größe und Farbe zu bearbeiten. OpenCV bietet zwar auch komplexere Möglichkeiten, Bilder auszuwerten, in diesem Projekt wird dafür allerdings eine andere Bibliothek, MediaPipe von Google, verwendet. Diese bietet Erkennungs- und Segmentierungslösungen für live Machine Learning in Videostreams an. Wir nutzen die Funktion, mit der die Pose von Menschen erkannt werden kann. Dazu werden markanten Körperteilen, wie der Nase oder dem Ellbogen sogenannte Landmarks zugewiesen werden, die dann für Berechnungen verwendet werden können.

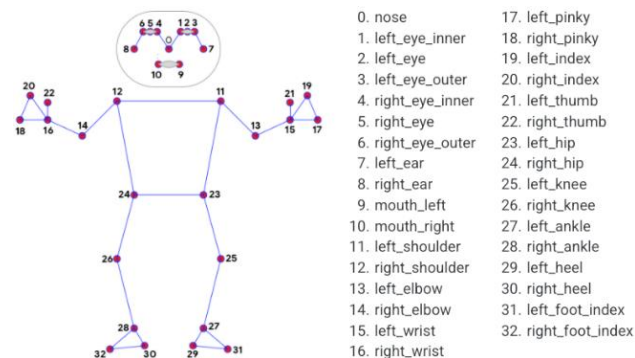


Abbildung 2-9: Übersicht über die Landmarks³

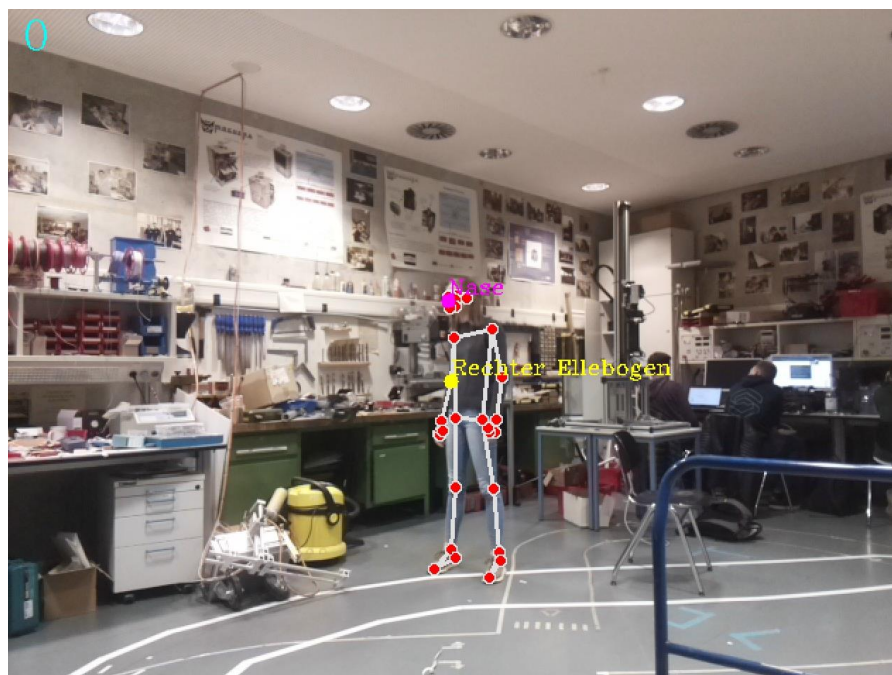


Abbildung 2-10: erkannter Mensch im Kamerabild des Turtlebots im LASIM

³ <https://google.github.io/mediapipe/solutions/pose>

3 Verbindung zum Turtlebot aufbauen

Um auf die Dateien auf dem Turtlebot zugreifen zu können, soll zuerst eine Remote Verbindung aufgebaut werden

1. Öffnen Sie Visual Studio Code

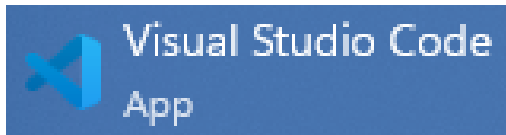


Abbildung 3-1: VSCode

2. Navigieren Sie an der Seite auf die *Remote Explorer* Extension → falls diese nicht sichtbar ist, gehen Sie in den Extension Reiter und installieren die Extension.

Abbildung 3-2: Seitenleiste VSCode



3. Erstellen Sie eine neue Remote Verbindung, der Benutzer ist *ubuntu* und die IPv4 Adresse müssen sie in Ihrer Netzwerkverwaltung herausfinden, oder Sie schließen kurz einen Bildschirm an den Raspberry Pi an.

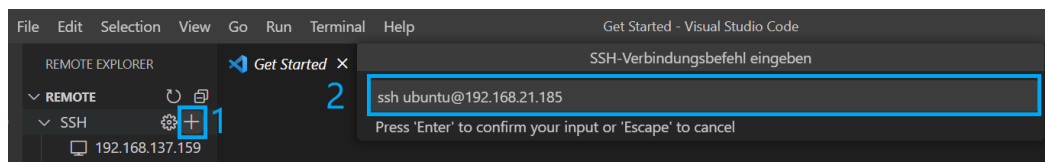


Abbildung 3-3: Verbindung von VSCode mit über SSH

4. Als Betriebssystem wählen Sie Linux und das Passwort des Turtlebots lautet: *turtlebot*
5. Gehen Sie nun nach File → Open Folder und Öffnen Ihr User Directory

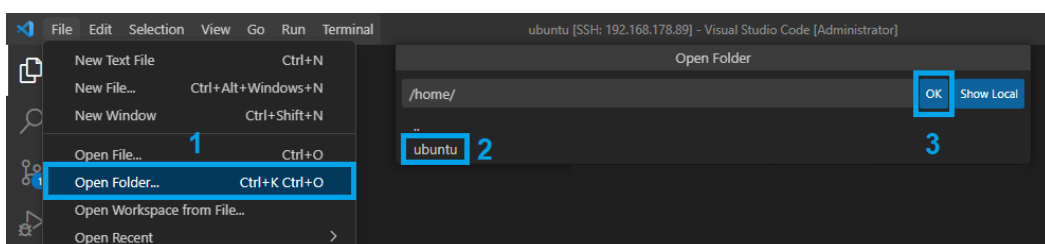


Abbildung 3-4: User Directory öffnen

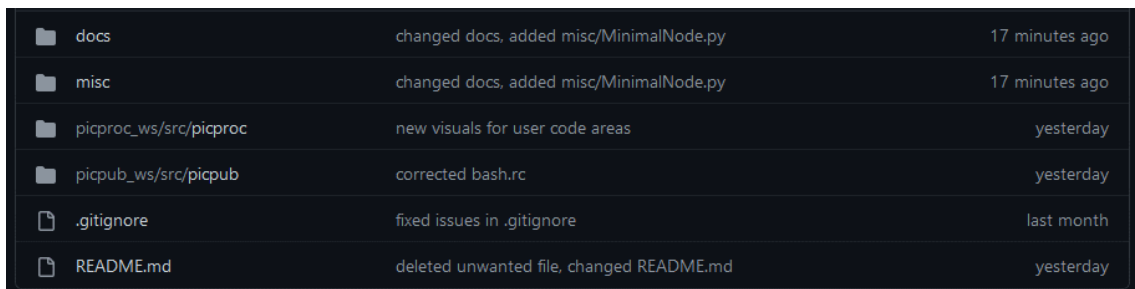
6. Öffnen Sie nun ein neues Terminal in VSCode mit `ctrl+ö` (Windows) oder aus der Titelleiste (Linux).

7. Clonen Sie sich das Repository mit dem Starterprojekt von Github über

```
$ git clone https://github.com/karlscholz/AutoSys-TurtlebotRos2.git
```

8. Öffnen Sie nun den soeben geklonten Ordner. In diesem Ordner befinden sich folgende Elemente, wobei Sie für die Anleitung eigentlich nur die beiden fett gedruckten Ordner interessieren:

docs/	die Dokumentation über technischen Details des Projekts, diese sind für das Durchführen des Versuchs nicht wichtig.
misc/	jeweils eine Kopie der <code>~/.bashrc</code> Dateien des Remote/Labor-PCs und des Raspberry Pis, auch diese ist im Rahmen der Anleitung uninteressant.
picproc_ws/	der Workspace der Picture Processor Node, hier werden Sie in Kapitel 6 arbeiten.
picpub_ws/	der Workspace der Picture Publisher Node, hier werden Sie in Kapitel 7 arbeiten.
.gitignore:	Datei die spezifiziert, welche Dateien von git ignoriert werden sollen, das ist alles, was durch den Kompiliervorgang erstellt wird.
README.md	grobe Beschreibung des Projekts



docs	changed docs, added misc/MinimalNode.py	17 minutes ago
misc	changed docs, added misc/MinimalNode.py	17 minutes ago
picproc_ws/src/picproc	new visuals for user code areas	yesterday
picpub_ws/src/picpub	corrected bash.rc	yesterday
.gitignore	fixed issues in .gitignore	last month
README.md	deleted unwanted file, changed README.md	yesterday

Abbildung 3-5: Übersicht des Repositories⁴

⁴ <https://github.com/karlscholz/AutoSys-TurtlebotRos2/>

4 Teleoperation Node

Bevor Sie Ihre eigenen Nodes programmieren, soll erst Ihre Netzwerkinfrastruktur getestet werden. Dafür verwenden Sie zum Einstieg eine schon vorhandene Node: Die Teleoperation Node. Mit dieser kann der Turtlebot mit einer Tastatur verfahren werden. Achten Sie darauf, dass Sie die Teleop Node nicht per SSH auf dem Raspi ausführen, da Sie so die ROS2 Netzwerk Verbindung nicht überprüfen.

4.1 Terminals

Um Nodes zu starten, Topics zu beobachten und andere Befehlskommandos auszuführen benötigen Sie Terminals.

SSH-Terminals für den RaspberryPi des Turtlebots können ganz einfach in VSCode geöffnet werden, sofern Ihr VSCode gerade mit Remote-SSH auf den Raspi zugreift. Für Terminals am Remote/Labor-PC kann die Tastenkombination Strg+Alt+t verwendet werden.

4.2 Turtlebot3 BringUp

Um die Hardware des Turtlebots steuern und auswerten zu können, muss auf diesem das sogenannte BringUp laufen.

1. Öffnen Sie dazu ein neues SSH-Terminal des Turtlebots in VSCode
2. Geben Sie den folgenden Befehl ein:

```
$ ros2 launch turtlebot3_bringup robot.launch.py
```

➔ Nach einer kurzen Zeit sollte die Hardware hochgelaufen sein, und sich im Zustand „Run“ befinden.

Die Node läuft so lange, bis Sie sie mit *Strg+c* beenden. Lassen Sie die Node so lange laufen, wie Sie den Turtlebot verwenden wollen. Das heißt auch, dass Sie keine neuen Befehle in das verwendete Terminal eingeben werden können. Wenn Sie eine weitere Node starten oder andere Befehle ausführen wollen, müssen Sie dazu ein zweites Terminal mit dem Plus Symbol am oberen rechten Rand der Konsole öffnen.

4.3 Teleoperation Node

Öffnen Sie nun mit Strg+Alt+t ein Terminal auf dem Remote/Labor-PC und geben Sie folgenden Befehl ein:

```
$ ros2 run turtlebot3_teleop teleop_keyboard
```

Wie Sie der Ausgabe entnehmen können, können Sie den Turtlebot mit den Tasten WXSAD steuern. Probieren Sie dies aus.

4.4 Weitere Informationen

Sie haben die Teleoperation Node auf dem Remote/Labor-PC gestartet und steuern damit den Turtlebot – oder auch nicht lol. Um sicherzugehen, ob es an Ihrer Hardware, oder ROS liegt, können Sie die Kommunikation der beiden Computer über das Topic `cmd_vel` (command velocity) überprüfen.

Geben Sie dafür den Befehl

```
$ ros2 topic echo /cmd_vel
```

in ein SSH-Terminal ein. Der Befehl funktioniert natürlich sowohl auf dem Turtlebot, als auch auf dem Remote/Labor-PC befindet. Sollte es auf dem Remote/Labor-PC auch keinen Output zeigen haben Sie ein massives Problem mit ihrer ROS-Installation.

Die Einträge in das Topic haben den Datentyp *geometry_msgs/msg/Twist*, welcher sich aus linearer und rotatorischer Geschwindigkeit, jeweils in x, y und z-Richtung zusammensetzt. Mit der Teleoperation Node publishen Sie vom Remote/Labor-PC die Geschwindigkeiten, die Sie mit der Tastatur vorgeben, in das Topic. Das BringUp des Turtlebots ist Subscriber des Topics und steuert die Antriebe des Turtlebots entsprechend der vorgegebenen Geschwindigkeiten an.

Verfahren Sie den Turtlebot mit den WXSAD-Tasten und beobachten Sie, wie sich die Einträge im Topic verändern.

5 Aufbau einer Node

Allgemein ist eine Node aus einer Klasse mit Funktionen und einer main Funktion aufgebaut. Die Methoden der Klasse werden entweder durch ein Timerevent, wie hier im Beispiel, oder das Vorhanden sein einer Nachricht in einem subscribeten Topic ausgeführt. Für das Verstehen des Aufbaus wurde eine minimale Node geschrieben, diese finden Sie als *misc/MinimalROS2Node.py*. Führen Sie sie gerne aus, sie sorgt dafür, dass der Turtlebot sich jeweils 2 Sekunden abwechselnd nach links und rechts dreht, bis sie Strg+C in die Konsole eingeben. Gehen Sie die Node einmal Zeile für Zeile durch und machen Sie sich mit dem Aufbau vertraut.

```
import rclpy                                     # Python library for ROS 2
from rclpy.node import Node                     # Handles the creation of nodes
from geometry_msgs.msg import Twist            # For velocity-messages

class myClass(Node):                             # my Class
    def __init__(self):                         # Constructor for my Class
        self.variable = 1.5                   # remembers stuff Objectwide
        self.msg = Twist()                    # variable to publish for driving
        super().__init__('cha_cha_real_smooth') # initiates Node with name
        self.publisher_ = self.create_publisher(msg_type=Twist, topic='cmd_vel', qos_profile=10) # creates publisher for cmd_vel
        self.timer = self.create_timer(timer_period_sec=2, callback=self.timer_callback) # timer, calls callback every 2sec

    def timer_callback(self):                   # method that gets called by timer
        self.variable = [1.5, -1.5][self.variable==1.5] # switches self.variable +-1.5
        self.msg.angular.z = self.variable      # sets angular.z to self.variable
        self.publisher_.publish(self.msg)       # publishes variable msg
        self.get_logger().info(f"Publishing ang: {self.msg.angular.z}") # logs/prints what was published

def main(args=None):                           # function main
    try:                                       # catches errors
        rclpy.init(args=args)               # Initialization of RCL
        myObject = myClass()                 # Create myObject of myClass
        rclpy.spin(myObject)                 # Starts running the Node
    except KeyboardInterrupt as e:           # Catches Ctrl+C
        print("\nEnded with: KeyboardInterrupt: ", repr(e)) # prints Why it ended
    except BaseException as e:               # Catches all other Exceptions
        print("Exception: ", repr(e))        # prints the Exception Info

    myObject.msg.angular.z = 0.0              # makes one last stop msg
    myObject.publisher_.publish(myObject.msg) # publishes the Twist msg to stop
    myObject.get_logger().info(f"Publishing ang: {myObject.msg.angular.z}") # logs/prints what was published

    myObject.destroy_node()                  # Destroys the node explicitly
    rclpy.shutdown()                         # Shutdown the ROS client library

if __name__ == '__main__':                   # checks if file is run top level
    main()                                   # calls main and starts everything
```

Sie fangen in den folgenden Abschnitten nicht mit einer leeren Datei an, sondern mit einem Programm, in dem viele Dinge, wie der Computer Vision Teil und andere Dinge, die im Kontext von ROS2 nicht relevant sind, schon umgesetzt sind. Stellen, in denen Sie arbeiten sollen, sind wie folgt mit langen Linienkommentaren gekennzeichnet:

```
# -----  
  
# Kommentare  
  
# Variablen „Prototypen“  
Self.msg =  
  
# -----
```

Ändern Sie nur Teile innerhalb solcher Linien. Diese Bereiche können auch schon Kommentare zu den Inhalten, oder bereits bereitgestellte Prototypen für zu definierende Variablen enthalten. Schauen Sie sich bei Interesse gerne auch die Programmabschnitte außerhalb der von Ihnen zu bearbeitenden Stellen an.

6 Programmierung Teil 1 – Picture Publisher

6.1 Grundsätzliches

In diesem Kapitel soll eine Node programmiert werden, die mit der Kamera des Turtlebots Bilder aufnimmt und diese dann in einem Topic published. Das Publi-
shen der Bilder ist notwendig, damit diese auch dem RemotePC zur Verfügung
stehen, auf dem die rechenintensivere Auswertung stattfinden sollen. Damit be-
fassen Sie sich allerdings in *Teil 2*.

Teil 1 wird in der Datei *picture_publisher.py* programmiert. Sie finden diese un-
ter `picpub_ws/src/picpub/picpub/picture_publisher_node.py`.
Die Node zum Publishen der Bilder wird in der Klasse *PicturePublisher* pro-
grammiert. Sie besteht aus den beiden Funktionen `__init__(self)` und `ti-
mer_callback(self)`.

Die Funktion `__init__(self)` wird einmal aufgerufen, sobald ein Objekt der Klas-
se definiert wird. Sie wird vor allem verwendet, um Variablen zu Beginn eines
Programms einen Anfangswert zuzuweisen.

`timer_callback(self)` ist, wie der Name schon sagt eine Callback-Funktion. Eine
Callback-Funktion ist eine Funktion, die als Reaktion auf ein bestimmtes Ereig-
nis ausgeführt wird. Solange das Ereignis nicht eingetreten ist, wirkt die Funkti-
on nicht blockierend und es können andere Funktionen ausgeführt werden. In
unserem Fall soll die Callback-Funktion dafür verwendet werden Bilder in re-
gelmäßigen Abständen zu publishen und durch einen Timer aufgerufen werden.

Damit der in der Klasse programmierte Code auch ausgeführt wird muss ein Ob-
jekt dieser Klasse definiert und initialisiert werden. Dies geschieht in der Funkti-
on `main(args=None)`, welche nach dem Start eines Programms automatisch als
erstes aufgerufen wird.

6.2 Programmieren der Node

6.2.1 Init -Funktion

Überklasse initialisieren

Wie die Definition zeigt, ist unsere Klasse *PicturePublisher* eine Unterklasse der Klasse
Node, welche von ROS zur Verfügung gestellt und in Zeile 2 importiert wird.

```
class PicturePublisher(Node):
```

Daher muss auch der Konstruktor (also die `init`-Funktion) der Klasse `Node` aufgerufen werden, um darin enthaltene Initialisierungen auszuführen. Dazu steht Ihnen folgende Funktion zur Verfügung:

```
super().__init__(node_name: str)
```

Mit dem Parameter `'node_name'` kann der `Node` ein Name übergeben werden. Nennen Sie sie `picture_publisher`.

Capture-Objekt erstellen

Mit dem Befehl

```
cv.VideoCapture(index: int)
```

erstellen sie ein Objekt, dass eine Kamera öffnet und das Aufnehmen von Videos ermöglicht. Über den Index kann angegeben werden, welche Kamera verwendet wird. Dieser ist für die Kamera des Turtlebots 0.

QoS-Profil erstellen

Mit sogenannten Quality-of-service Profilen kann die Kommunikation von verschiedenen Nodes spezifiziert werden.

Mit folgendem Befehl können Sie ein Profil erstellen.

```
QoSProfile(**kwargs: Any)
```

In den Klammern der Funktion können verschiedene Eigenschaften des Profils vorgegeben werden. Für unsere Anwendung benötigen wir die Eigenschaften *reliability* und *history*.

Mit *history* können Sie einstellen, ob alle Bilder („keep all“) oder nur die letzten *x* Bilder („keep last, depth *x*“) gespeichert werden sollen. Setzen sie *history* auf „keep last“ und *depth* auf „1“.

```
history=QoSHistoryPolicy.KEEP_LAST,depth=1
```

Reliability kann die Werte „reliable“ oder „best effort“ annehmen. Bei „reliable“ wird sichergestellt, dass alle Bilder übertragen werden, wodurch ein zeitlicher Versatz entstehen kann. Bei „best effort“ werden so viele Bilder wie möglich übertragen. Wenn die Übertragung zu lange dauert, werden ältere Bilder zugunsten von aktuelleren Bildern übersprungen. Setzen Sie die *reliability* auf „BEST_EFFORT“. Gehen sie dabei analog zur *history* vor. Die Eigenschaft „BEST_EFFORT“ finden Sie in der Klasse `QoSReliabilityPolicy`

Publisher erstellen

Erstellen Sie nun einen Publisher, mit dem Sie später die Bilder in ein Topic publishen:

```
self.create_publisher(self, msg_type, topic: str, qos_profile: Union[QoSProfile, int], *, callback_group: Optional[CallbackGroup] = None, event_callbacks: Optional[PublisherEventCallbacks] = None)
```

Sie müssen der Funktion den Datentyp der Daten, die Sie publishen wollen, (CompressedImage), den Name des Topics (imagePi) und das QoS-Profil, das Sie gerade erstellt haben, übergeben. Die anderen Parameter sind optional und werden mit „None“ initialisiert, falls Sie nichts anderes übergeben.

Aufruf der Callback-Funktion

Zuletzt müssen Sie dafür sorgen, dass die Callback-Funktion alle 0,2 Sekunden aufgerufen wird. Setzen Sie dafür die Variable timer_period auf die gewünschte Zeit und erstellen Sie einen Timer mit:

```
self.create_timer(delay_seconds: int, callback: Callable)
```

Als Argumente übergeben Sie die vorher programmierte Zeit und die Callback-Funktion (self.timer_callback)

6.2.2 Callback-Funktion

Bild speichern

Zuerst brauchen Sie ein Bild, das Sie im Folgenden publishen können. Dazu holen Sie sich das letzte von der Kamera aufgenommene Bild und speichern es in einer Variable. Dieses Bild ist in der Variable last_frame des Objekts self.cam_cleaner gespeichert. Sie können also mit folgendem Befehl darauf zugreifen:

```
self.cam_cleaner.last_frame
```

Bild publishen

Der in der init-Funktion erzeugte Publisher verfügt über die Funktion publish():

```
self.publisher.publish(self, msg: Union[MsgType, bytes])
```

Als Argument übergeben Sie eine komprimierte Form des Bildes, das Sie im letzten Schritt gespeichert haben. Um das Bild zu komprimieren steht Ihnen folgender Befehl zur Verfügung:

```
self.bridge.cv2_to_compressed_imgmsg(self, cvim, dst_format='jpg')
```

Aktion loggen

ROS2 verfügt über eine logging-Funktion mit der ein logging-Eintrag gleichzeitig an mehrere Ziele (z.B. die Konsole und das topic `\rosout`) übergeben werden kann. Erzeugen Sie eine Textausgabe nach jedem publishen eines Bildes mit der Funktion:

```
self.get_logger().info(self, message, **kwargs)
```

6.3 Programmieren der Main-Methode

Programmstart

Bei Beginn des Programms müssen Sie ROS einmal initialisieren. Dazu verwenden Sie den Befehl

```
rclpy.init(args=args)
```

Anschließend müssen Sie ein Objekt der Klasse `PicturePublisher`, die Sie in 6.1 programmiert haben, erstellen. Dieses ist dann die Node, die die Bilder der Kamera in das Topic `imagePi` published. Die Syntax dafür lautet:

```
object = Class()
```

Danach müssen Sie dafür sorgen, dass die Node in ROS kontinuierlich ausgeführt wird. Dafür gibt es den Befehl:

```
rclpy.spin(node: 'Node', 'Executor' = None)
```

Programmende

Am Ende des Programms soll die Node `PicturePublisher` gelöscht werden, damit der Speicher, den diese verwendet hat, wieder freigegeben wird.

```
node.destroy_node()
```

Wenn die Node gelöscht wurde, müssen Sie als letztes noch ROS beenden:

```
rclpy.shutdown()
```


6.4 Testen der Node

Nachdem Sie die Node nun fertig programmiert haben, können Sie diese nun testen:

1. Speichern Sie dazu zuerst Ihre Python-Datei mit Strg+s oder unter File → Save.
2. Öffnen Sie nun ein neues Terminal und navigieren Sie in das Verzeichnis „picpub_ws“.

```
cd ~AutoSys-TurtlebotRos2/picpub_ws
```

3. Führen Sie dort folgende Befehle aus:

```
$ colcon build  
$ . install/setup.bash
```

Es kann sein, dass dabei beim ersten Mal ein Fehler auftritt. Dann müssen Sie noch die Dependencies updaten

```
$ rosdep update  
$ rosdep install -i --from-path src --rosdistro foxy -y
```

4. Nun können Sie die Node starten:

```
$ ros2 run picpub picture_publisher_node
```

Wenn Sie alles richtig programmiert haben, bekommen Sie im Terminal Ausgaben, dass Bilder gepublished werden. Falls das nicht der Fall ist, können Sie gleich auch noch einen weiteren wichtigen Teil der Programmierung üben: Die Fehlersuche.

7 Programmierung Teil 2 - Picture Processor

7.1 Grundsätzliches

Im ersten Kapitel wurde bereits eine Node programmiert, die die aufgenommenen Bilder in das Topic *imagePi* published. Durch Subscriben des Topics können die Bilder eingelesen und damit anschließend ein automatisiertes Folgen implementiert werden. Um eine Person zu verfolgen, muss das Bild ausgewertet werden. Dafür wird die Mediapipe-Bibliothek benötigt. Die damit erlangten Informationen sollen dann genutzt werden, um dem Turtlebot Fahrkommandos zu geben, d.h. sie in das Topic *cmd_vel* zu publishen, die dann von einer vorprogrammierten Node im Turtlebot ausgelesen werden und die Motoren ansteuern. Es handelt sich bei der Bildauswertung mit Mediapipe um eine eher rechenintensive Aufgabe, welche die Recheneinheit auf dem Turtlebot überfordern würde. Deshalb wird die Node auf dem *RemotePC* ausgeführt. Die Kommunikation kann elegant mittels des ROS2 Frameworks implementiert werden.

Teil 2 wird in der Datei *picproc_node.py* programmiert. Sie finden diese unter *AutoSys-TurtlebotRos2\picproc_ws\src\picproc\picproc\picproc_node.py*. Die Node zum Publishen der Bilder wird in der Klasse *PictureProcessor* programmiert. Sie besteht aus den beiden Funktionen *__init__(self)* und *listener_callback(self, data)*. Des Weiteren beinhaltet sie die Hilfsfunktionen *calcX_is(self, results)*, *calcY_distance(self, results)*, *PIDRot(self, x_is, currImage)* und *PIDLin(self, y_distance, currImage)*.

Die Funktion *__init__(self)* wird einmal aufgerufen, sobald ein Objekt der Klasse definiert wird. Sie wird vor allem verwendet, um Variablen zu Beginn eines Programms einen Anfangswert zuzuweisen, sie also zu initialisieren.

listener_callback(self, data) ist, wie der Name schon sagt eine Callback-Funktion. Eine Callback-Funktion ist eine Funktion, die als Reaktion auf ein bestimmtes Ereignis ausgeführt wird. Solange das Ereignis nicht eingetreten ist, wirkt die Funktion nicht blockierend und es können andere Funktionen ausgeführt werden. Diese Callback-Funktion wird jedes Mal aufgerufen, wenn ein neues Bild in das *imagePi*-Topic gepublished wurde.

calcX_is(self, results) berechnet die Position entlang der x-Achse (horizontal) der im Bild erkannten Position aus den Koordinaten der erkannten Hüftlandmarks und gibt diese als Rückgabewert zurück. Dabei handelt es sich um einen relativen Wert. Befindet sich die Person beispielsweise in der Mitte, dann gibt

die Funktion den Wert 0.5 zurück, befindet sie sich ganz links bzw. ganz rechts im Bild, dann gibt sie 0 bzw. 1 zurück. Wurden die nötigen Landmarks nicht erkannt, so gibt die Funktion -1 zurück.

calcY_distance(self, results) berechnet die Länge bzw. die mittlere Länge des Oberschenkels aus den erkannten Hüft- und Knie-Landmarks und gibt diese zurück. Dieser Wert wird ebenfalls relativ zum Bild ausgegeben. Wurden die benötigten Landmarks nicht erkannt, gibt die Funktion -1 zurück.

PIDRot(self, x_is) berechnet aus dem Rückgabewert der Funktion *calcX_is(self, results)* (*x_is*) die Rotationsgeschwindigkeit um die z-Achse (Rotation des Roboters) und gibt diese zurück. Dabei wird diese mit dem Ziel geregelt, dass sich die Person in der Mitte des Bildes befindet und *x_is* den Wert 0.5 annimmt.

PIDLin(self, y_distance) berechnet aus dem Rückgabewert der Funktion *calcY_distance(self, results)* (*y_distance*) die Verfahrensgeschwindigkeit in x-Richtung (nach vorne) und gibt diese zurück. Diese wird so geregelt, dass die relative Länge des Oberschenkels im Bild 0.23 beträgt. Ist der Wert größer, ist die Person zu nah am Turtlebot, ist der Wert kleiner, ist sie zu weit entfernt.

Damit der in der Klasse programmierte Code auch ausgeführt wird, muss ein Objekt dieser Klasse definiert und initialisiert werden. Dies geschieht in der Funktion *main(args=None)*, welche nach dem Start eines Programms automatisch als erstes aufgerufen wird.

7.2 Programmieren der Node

7.2.1 Init-Funktion

Überklasse initialisieren

Analog zu *Teil 1* wird die Klasse *ImageProcessor* erstellt, in der sich die *__Init__(self)* Funktion befindet.

```
class ImageProcessor(Node):
```

Wie in *Teil 1* muss der Konstruktor der Klasse *Node* mit der folgenden Funktion aufgerufen werden.

```
super().__init__(node_name: str)
```

Mit dem Parameter 'node_name' kann der *Node* ein Name übergeben werden. Nennen Sie sie *picproc*.

QoS-Profil erstellen

Aus Kompatibilitätsgründen sollte das QoS-Profil zum Subscriben eines Topics dem QoS-Profil entsprechen, welches zum Publishen verwendet wird (Tipp: siehe *Teil 1*).

Publisher erstellen

Erstellen Sie nun einen Publisher, mit dem Sie später die Geschwindigkeiten in ein Topic publishen.

```
self.create_publisher(msg_type, topic: str, qos_profile: Union[QoSProfile, int], *, callback_group: Optional[CallbackGroup] = None, event_callbacks: Optional[PublisherEventCallbacks] = None)
```

Sie müssen der Funktion den Datentyp der Daten, die Sie publishen wollen (Twist), den Namen des Topics (cmd_vel) und ein QoS-Profil (10), mit dem das cmd_vel gepublishet werden muss, übergeben. Die anderen Parameter sind optional und werden mit „None“ initialisiert, falls Sie nichts anderes übergeben.

Subscriber erstellen

Erstellen Sie einen Subscriber, mit dem Sie die Bilder subscriben.

```
self.create_subscriber(msg_type, topic: str, callback: Callable[[~MsgType], None], qos_profile: Union[QoSProfile, int], *, callback_group: Optional[CallbackGroup]=None, event_callbacks: Optional[SubscriptionEventCallbacks]=None, raw=False)
```

Sie müssen der Funktion den Datentyp der Daten, die Sie subscriben wollen, (CompressedImage), den Namen des Topics (imagePi), eine Callback-Funktion, die aufgerufen wird, sobald etwas gepublishet wurde (listener_callback) und das QoS-Profil, das Sie gerade erstellt haben, übergeben. Die anderen Parameter sind optional und werden mit „None“ initialisiert, falls Sie nichts anderes übergeben.

Twist-Message erstellen:

Erstellen Sie nun eine Message-Objekt vom Typ Twist in der Sie die Geschwindigkeitswerte speichern werden. Sie können das Objekt mit folgendem Aufruf erstellen:

```
Twist()
```

Die Twist-Message besteht aus zwei Vektoren (angular, linear) mit je 3 Werten (x, y, z) vom Typen *float* und wurde in Zeile 5 wie folgt importiert.

```
from geometry_msgs.msg import Twist
```

7.2.2 Callback-Funktion

Die `listener_callback(self, data)` wird aufgerufen, sobald ein Bild in das Topic `imagePi` gepublished wird. In `data` befindet sich das eingelesene Bild. Sie sollen nun einen logging-Eintrag erstellen, welcher informiert, dass ein Bild eingelesen wurde. Verwenden Sie dafür die aus *Teil 1* bekannte Funktion:

```
self.Get_logger().info(self, message, **kwargs)
```

In der Callback wird anschließend mit einer if-Abfrage geprüft, ob eine Person bzw. deren Landmarks erkannt wurden. Ist das der Fall, werden die Werte der Geschwindigkeit in x-Richtung und der Winkelgeschwindigkeit in z-Richtung berechnet. Nun müssen diese noch in die vorher erstellte Nachricht `msg` geschrieben werden. Überlegen Sie nun, welche der vorherberechneten Variablen in welches Feld der Nachricht `msg` geschrieben werden muss (Tipp: Lesen Sie sich die Beschreibung der Hilfsfunktionen durch). Stellen Sie zudem sicher, dass es sich bei den Werten um Werte vom Typ `float` handelt. Sie können Werte zu `float` werten konvertieren mit:

```
float(zuKonvertierenderWert)
```

Dabei wird der Wert als `float`-Wert zurückgegeben.

Im else-Zweig der if-Abfrage landet das Programm, sobald keine Landmarks detektiert wurden. Der Roboter soll dann einfach stehen bleiben. Setzen Sie die Felder der Nachricht `msg` so, dass dies gewährleistet ist.

Anschließend müssen Sie die Nachricht nur noch publishen. Verwenden Sie dafür den vorher erstellten Publisher `publisher_`. Das Publisher-Objekt enthält eine Funktion, mit der Sie die Nachricht veröffentlichen können.

```
Publish(msg: Union[~MsgType, bytes])
```

Sie müssen also die zu publishende Nachricht übergeben.

Erstellen Sie nun erneut einen logging-Eintrag, der die Werte der Geschwindigkeit in x-Richtung und der Winkelgeschwindigkeit in z-Richtung festhält. Die nötige Funktion ist Ihnen bereits bekannt.

Anschließend werden einige Informationen in das Bild (`currImage`) geschrieben und dieses angezeigt.

7.3 Main-Funktion

Initialisieren Sie Ros, kreieren Sie ein Objekt der Klasse `ImageProcessor` und lassen Sie die Node kontinuierlich laufen (Tipp: siehe *Teil 1*).

Bevor Sie die Node ordnungsgemäß, wie in *Teil 1*, beenden. Sollen Sie sicherstellen, dass der Turtlebot stillsteht, sobald das passiert. (Tipp: Geschwindigkeiten des Objekts `imageProcessor` in `msg` setzen; diese `msg` mit der in `imageProcessor` enthaltenen `publish_`-Funktion publishen).

Abbildungsverzeichnis

Abbildung 1-1: Übersicht der Kommunikationsstruktur	5
Abbildung 2-1: Turtlebot 3 Waffle Pi.....	6
Abbildung 2-2: Biege-Modifikation der Kamerahalterung am Turtlebot3	7
Abbildung 2-3: Überblick über ein ROS Netz.....	9
Abbildung 2-4: IntelliSense: Hover Information der Methode <i>create_subscription</i>	11
Abbildung 2-5: IntelliSense Autovervollständigung	11
Abbildung 2-6: IntelliSense Go to Definition / Peek → Definition	12
Abbildung 2-7: IntelliSense Definition Peek.....	12
Abbildung 2-8: Debugging von Python Programmen in VSCode	13
Abbildung 2-9: Übersicht über die Landmarks	14
Abbildung 2-10: erkannter Mensch im Kamerabild des Turtlebots im LASIM.....	14
Abbildung 3-1: VSCode	15
Abbildung 3-2: Seitenleiste VSCode.....	15
Abbildung 3-3: Verbindung von VSCode mit über SSH	15
Abbildung 3-4: User Directory öffnen.....	15
Abbildung 3-5: Übersicht des Repositories	16

Alle Bilder, bei denen die Quelle nicht explizit per Fußnote angegeben wurde, wurden vom Verfasser erstellt.