Aufgabenstellung ROS Tutorial

Einführung

Zur Einführung werdet ihr euch mit den Grundbausteinen des Praktikums vertraut machen:

Carla Simulator

- Der ***Carla Simulator*** beinhaltet die komplette Umgebung und Verkehrsteilnehmer der Simulation.
- Der Simulator läuft dabei als ein Server; Clients können sich mit ihm verbinden (z.B. euer Fahrzeug)
- Carla bietet eine PythonAPI an, über welche die Simulation beeinflusst werden kann. (Ihr werdet nicht direkt mit der API inteagieren, die Kommunikation übernimmt die ***Carla ROS Bridge***)

Aufgabe

1. Startet den ***Carla Simulator*** mit den folgenden Befehlen:
    - `cd ~/PAF/carla/`
    - `./CarlaUE4.sh`

    Nun könnt ihr innerhalb des Simulators mit ***WASD*** und der Maus eine der zur Verfügung stehenden Städte erkunden.

    > ***Hinweis:*** Startet den ***Carla Simulator*** als erstes bevor euren anderen Komponenten; ansonsten kann die ***Carla ROS Bridge*** keine Verbindung aufbauen.

Carla ROS Bridge

- Die ***Carla ROS Bridge*** bildet die Abstraktionsschicht zum Simulator und bietet ein Interface zum Herauslesen und Einschleusen von Daten an.
- Die Kommunikation erfolgt dabei über das Empfangen und Versenden von Messages über die entsprechenden Topics.

Aufgabe

1. Öffnet eine neue Konsole und aktiviert die zur Verfügung gestellte Python-Umgebung. Diese beinhaltet alle nötigen Pakete zur Ausführung der ***Carla ROS Bridge***:
    - `conda activate paf`

    > ***Wichtig:*** Jeglicher Code der mittels der ***Carla ROS Bridge*** ausgeführt wird muss in dieser Umgebung aufgerufen werden!

2. Erzeugt ein manuell steuerbares Fahrzeug anhand der entsprechenden Node der ***Carla ROS Bridge***:
    - `roslaunch carla_ros_bridge carla_ros_bridge_with_example_ego_vehicle.launch`

    Gleichzeitig werden mehrere Verkehrsteilnehmer erzeugt. Mit den entsprechenden Tasten könnt ihr das Fahrzeug in der Simulaion steuern.

rqt_graph

- ***rqt_graph*** bietet eine visuelle Darstellung der registrierten Topics und deren Verbindung mit erzeugten Nodes.

Aufgabe

1. Führt das Programm in einer neuen Konsole mit laufendem Fahrzeug mit folgendem Befehl aus:
    - `rqt_graph`

rviz

- Mit ***rviz*** können die Ausgaben der Sensoren visualisiert werden.

Aufgabe

1. Der Befehl für das Starten von ***rviz*** ist:
    - `rviz`
2. Fügt über den ***Add*** Button links unten weitere Topics der Sensoren der Visualisierung hinzu.

> ***Hinweis:*** Die entsprechenden Topics der Sensoren können auch über das Dropdown-Menü im Fenster ***Displays*** geändert werden.

Bearbeitung

Vorbereitung des Workspace

Damit euer Code teil des ***catkin*** Workspace ist und von ***ROS*** gefunden wird gibt es zwei Möglichkeiten. Der Ordner `src` des ***catkin*** Workspace enthält entweder

- die Dateien mit Code direkt oder
- einen symbolischen Link auf einen externen Ordner.

Zur Übersichtlichkeit verwenden wir die zweite Option.

Aufgabe

1. Erstellt einen Ordner `~/PAF/workspace/` (oder an einem anderen beliebigen Ort).
2. Fügt einen symbolischen Link im Ordner `~/PAF/catkin_ws/src` auf den erstellten Ordner ein.

> ***Hinweis:*** Symbolische Links können mit dem Befehl `ln -s {destination}` erstellt werden.

3. Erstellt ein ***ROS***-Paket im verlinkten Ordner mit dem Namen `paf_tutorial` via `catkin_create_pkg`.
4. Baut den ***catkin*** Workspace im Ordner `~/PAF/catkin_ws/` via `catkin_make` neu.

> ***Hinweise:***
>
> - Zu Beginn besteht die Möglichkeit dass der Workspace zweimal gebaut werden muss um fehlerfrei durchzulaufen.

> - Sobald ihr neue Pakete in eurer Node benutzt, vergesst nicht die entsprechenden Abhängigkeiten in den Dateien `CMakeLists.txt` und `package.xml` zu ergänzen.

Vorbereitung des Pakets `paf_tutorial`

Die restliche Implementierung der Einführung findet im oben erstellten Paket `paf_tutorial` statt. Dazu erstellt ihr Sourcedateien die eueren Code beinhalten und Launchfiles um ihn entsprechend auszuführen.

Aufgabe

1. Erstellt im Pfad eures Pakets zwei neue Ordner `src` und `launch` welche die entsprechenden Dateien enthalten werden.

2. Im selben Verzeichnis wird zudem die Datei `setup.py` benötigt, in der die Ordnerstruktur des Pakets deklariert ist. Kopiert dazu folgenden Code in die Datei:

   ```
   from distutils.core import setup
   from catkin_pkg.python_setup import generate_distutils_setup

   d = generate_distutils_setup(packages=["paf_tutorial"], package_dir={"":
   "src"})

   setup(**d)
   ```

3. Jetzt kommen wir zum eigentlichen Teil der Implementierung. Als Referenz dienen euch die Dateien der ***Carla ROS Bridge*** unter `~/PAF/ros-bridge/`. Erstellt dazu eine Datei `paf_tutorial.py` im Ordner `src/paf_tutorial/`.

   > Für Einzelheiten bietet die Sourcedatei der ***Ackermann Control*** (`~/PAF/ros-bridge/carla_ackermann_control/src/carla_ackermann_control/carla_ackermann_control_node.py`) eine gute Referenz.

   Die Datei soll dabei folgende Merkmale besitzen:

   - Die Hauptklasse der Datei muss von `CompatibleNode` erben.
   - Die Hauptklasse benötigt eine Funktion
     - `__init__` (zunächst nur für Logging).
     - `run` mit einer Ausführungschleife via Timer.
     - `__del__` für das Aufräumen bei Programmende (zunächst leer).
     - `main` zur Initialisierung der Node und Erzeugung der Hauptklasse.

     Füllt die Funktionen jeweils mit `print`-Befehlen zum Logging. Die Datei muss zudem die Berechtigung zum Ausführen haben. wendet dafür den Befehl `chmod +x paf_tutorial.py` an.

4. Zum Starten des Pakets benötigt ihr noch eine Datei `paf_tutorial.launch` im entsprechenden Ordner `launch`.

> Als Referenz dient euch dabei das Launchfile der ***Ackermann Control*** (`~/PAF/ros-bridge/carla_ackermann_control/launch/carla_ackermann_control.launch`).

Im ersten Schritt genügt eine parameterlose Ausführung der Node.

5. Testet euer Implementierung, indem ihr den gesamten Stack (via `roslaunch`) ausführt:
    - ***Carla Simulator*** starten
    - ***Carla ROS Bridge*** starten
    - `paf_tutorial` starten

    Ihr solltet nun in der Konsole der ***Carla ROS Bridge*** den Output der Logging-Aufrufe innerhalb der Funktionen sehen.

Fahrzeug in der Simulation

Im nächsten Schritt lassen wir eure Node ein Fahrzeug in der Simulation erzeugen und steuern. Dabei soll das Auto zunächst geradeaus fahren und dann auf Kommando einen U-Turn ausfüren.

Aufgabe

1. Erweitert euer Launchfile um folgende Argumente:

Argument	Wert (default)	Beschreibung
`role_name`	ego_vehicle	ID für die Benennung der Topics
`control_loop_rate`	0.1	Rate der Steuerkommandos (pro Sekunde)
`town`	Town04	Karte, die geladen wird
`vehicle_filter`	vehicle.tesla.model3	Modell des Fahrzeugs
`spawn_point`	344.8, -10.6, 0.4, 0.0, 0.0, 179.0	Startpunkt des Fahrzeugs

2. Um den Stack bequemer auszuführen fügt folgende Launchfiles mit entsprechenden Parametern in eurem Launchfile ein:
    - `carla_ros_bridge_with_example_ego_vehicle.launch`
        - Übergebene Argumente:
            - `role_name`
            - `town`
            - `vehicle_filter`
            - `spawn_point`
    - `carla_ackermann_control.launch`
        - Übergebene Argumente:
            - `role_name`
            - `control_loop_rate`
    - `paf_tutorial`
        - Übergebene Argumente:
            - `role_name`

        - `control_loop_rate`

3. Damit das Fahrzeug geradeaus fahren kann müssen die entsprechenden Message an die ***Ackermann Control*** gesendet werden. Erweitert dafür euer Skript um folgende Änderungen:

    - In `__init__`:
        - Parameter des Launchfiles auslesen
        - Publisher des Topics `/carla/{}/ackermann_cmd` erzeugen
    - In `run`:
        - Innerhalb der Ausführungsschleife:
            - Message `AckermannDrive` erzeugen
            - Daten der Message so setzen, dass euer Fahrzeug geradeaus fährt
            - Message publishen
        - Timer mit `control_loop_rate` und Loop-Funktion starten

    > Als Referenz dient euch hier wieder die Sourcedatei der ***Ackermann Control*** und die Beschreibung der Message im ***ROS*** Wiki.

    Das Fahrzeug sollte nun beim Starten des Simulators und eurem Launchfile geradeaus fahren.

4. Im letzten Schritt soll das Programm auf ein von außen gegebenes Topic reagieren und ein Wendemanöver ausführen.

    - Erstellt dazu analog zum eben erstellten Publisher zwei Subscriber für die Topics:

        - IMU
        - U-Turn (benutzerdefiniert)

        Die Message des U-Turns kann dabei `Empty` sein, da wir lediglich am Empfangen der Nachricht interessiert sind.

    - Verarbeitet beide Messages mit entsprechenden Funktionen.

    - Erweitert eure Loop-Funktion:

        - Bei Empfangen der U-Turn Message mithilfe der IMU-Daten solange eine Kurve fahren, bis 180° abgeschlossen sind.
        - Nach abgeschlossenem U-Turn weiter geradeaus fahren.

    Achtet darauf, dass das Fahrzeug nicht weiter als 180° dreht. Spielt gegebenenfalls mit den Werten der `AckermannDrive`-Message.

    > ***Hinweis:*** Zur Konvertierung der IMU-Daten kann euch die Funktion
    >
    > ```
    > from tf.transformations import euler_from_quaternion
    > ```
    >
    > hilfreich sein.

Anbindung an das Carla Leaderboard

Als letzte Aufgabe soll das von euch erstellte Fahrzeug an das ***Carla Leaderboard*** angeschlossen werden. Das ***Carla Leaderboard*** bietet dafür vorgefertigte Dateien zum Ausführen des Programms an.

> Hilfreiche Links:
>
> - https://leaderboard.carla.org/get_started/
> - https://github.com/carla-simulator/leaderboard-agents

Aufgabe

1. Testet zunächst das ***Carla Leaderboard*** mit manueller Steuerung, indem ihr die Datei `test_run.sh` im Verzeichnis `~/PAF/leaderboard/` ausführt. Nachdem die Route geladen wurde könnt ihr das Fahrzeug mit euer Tastatur steuern. Bei Abbruch der Route (`Ctrl + C`) wird der Score berechnet und im Konsolenfenster angezeigt.

2. Erzeugt ein neues Launchfile `paf_tutorial_leaderboard.launch`:

   - Argumente:

Argument	Wert (default)
`role_name`	hero
`control_loop_rate`	0.1

   - Nodes:
     - `Carla Ackermann Control`
     - `rviz`
     - `paf_tutorial`

   Ein entsprechende Config Datei für ***rviz*** ist im Ordner `~/Dokumente` zu finden. Dieses Launchfile wird vom ***Carla Leaderboard*** bei Programmstart aufgerufen.

3. Erstellt eine neue Datei `paf_agent.py` im Ordner `src`. Dieser dient als Parametrisierung des ***Carla Leaderboards*** und eures Fahrzeugs. Achtet dabei auf folgende Punkte:

   - Der Entrypoint beschreibt den Namen eurer Agentenklasse, welche von ROS1Agent erbt, und das entsprechende Launchfile mit Parametern:

     ```
     from leaderboard.autoagents.ros1_agent import ROS1Agent
     from leaderboard.autoagents.autonomous_agent import Track

     def get_entry_point():
         return "PAFAgent"


     class PAFAgent(ROS1Agent):

         def get_ros_entrypoint(self):
             return {
                 "package": "paf_tutorial",
                 "launch_file": "paf_tutorial_leaderboard.launch",
                 "parameters": {
     ```

```
                "role_name": "hero",
                "control_loop_rate": "0.1"
            }
        }
```

- Die Methode `setup` setzt den gewählten Track des ***Carla Leaderboards***. Da ihr mit Kartendaten arbeiten dürft wählt ihr hier den Track.MAP:

```
    def setup(self, path_to_conf_file):
        self.track = Track.MAP
```

- Die Methode `sensors` definiert die Sensoren an eurem Fahrzeug. Für das Tutorial benötigen wir eine Kamera und eine IMU (optional Speedometer):

```
    def sensors(self):
        sensors = [
            {"type": "sensor.camera.rgb", "id": "rgb_view", "x": 0.7,
"y": 0.0, "z": 1.60, "roll": 0.0, "pitch": 0.0, "yaw": 0.0, "width":
800, "height": 600, "fov": 100},
            {"type": "sensor.other.imu", "id": "imu", "x": 0.0, "y":
0.0, "z": 0.0, "roll": 0.0, "pitch": 0.0, "yaw": 0.0},
            {"type": "sensor.speedometer", "id": "speed"}
        ]
        return sensors
```

- Die Methode `destroy` räumt bei Programmende auf:

```
    def destroy(self):
        super(PAFAgent, self).destroy()
```

4. In der Datei `paf_tutorial.py` müssen ein paar Änderungen vorgenommen werden:

- Ändert die Geschwindigkeit eures Fahrzeugs auf einen einen sehr kleinen Wert, aber größer als `0.00001` (z.B. `0.000011`) bei normaler Fahrt ohne U-Turn, um eine anfängliche Kollision zu vermeiden.
- Fügt einen neuen Publisher am Ende der Funktion `__init__` hinzu. Er signalisiert dem ***Carla Leaderboard*** dass der Software Stack initialisiert wurde.

```
from ros_compatibility.qos import QoSProfile, DurabilityPolicy

self.status_pub = self.new_publisher(Bool,
"carla/{}/status".format(self.role_name), qos_profile =
```

```
QoSProfile(depth = 1, durability = DurabilityPolicy.TRANSIENT_LOCAL))
self.status_pub.publish(True)
```

5. Um den Entrypoint des ***Carla Leaderboards*** zu wechseln benötigen wir ein neues Bashskript. Erstellt im Ordner `~/PAF/leaderboard/` die Datei `paf_run.sh`. Der Inhalt der Datei ist analog zum Skript mit manueller Steuerung. Ändert lediglich den Entrypoint auf eure Datei `paf_agent.py` und den Track zu Track.MAP. Ändert auch hier die Berechtigung zur Ausführung der Datei.

6. Startet den ***Carla Simulator*** und das ***Carla Leaderboard*** und sendet via `rostopic` eine Message für den U-Turn.