POLITECNICO DI TORINO

Master's Degree
in Mechatronics Engineering

Master's Degree Thesis

Trajectory Planning for Self-Driving Cars

Supervisor
prof. Stefano Malan

Candidate
Viglietti Claudia

Academic Year 2021-2022

The core challenge is to put a safe and reliable automated driving system on the road

Dr. Stephan Hönle

Abstract

This thesis aims to achieve a better understanding about trajectory and path planning in self-driving. The author participated to the Bosch Future Mobility Challenge 2022 (BFMC 2022), a challenge proposed by Bosch Romania in which students are asked to develop autonomous driving and connectivity algorithms on $1 : 10$ scaled vehicles. In the competition, the car must perform specific tasks:

1. lane follow;
2. lane keeping;
3. intersection detection;
4. traffic sign and traffic light recognition;
5. parking manoeuvre;
6. overtake manoeuvre;
7. object detection;
8. trajectory and path planning based on graphs and GPS connection.

At the beginning, all members of the team worked together to make the vehicle able to fulfill the first 6 tasks. Next, each one had its own assignment and the author job was studying the localisation system and trajectory planning; path planning was studied additionally after the competition to complete a methodological study. The objective of this work is to find a predefined path where the car is able to perform all the tasks required by the challenge and to go deeper into comprehend how to find the shortest path knowing the starting and final points.

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

Technology advances always faster and daily, humans use every kind of electronic devices for various purposes reducing effort, lost time and danger. The production of autonomous vehicles, robots or aircraft becomes a leading industry. The innovations are helpful to improve both performance and safety.

The motivation of the thesis is to design a driver-less vehicle following the rules of the Bosch Future Mobility challenge. In particular, automated vehicle, robot or aircraft can move from a designated starting point to a destination one in a given environment. In order to reach the final point, a path must be planned and there are more methods to achieve the goal. The following approach are studied in deep:

- Trajectory planning is useful to design a predefined path that autonomous system must follow in relation to a defined time law. For example, autonomous car can perform a limited set of possible maneuvers and the planner has to help it to keep the lane, parking or overtake.
- Path planning is used to complete the task in the shortest time with minimum movement, save energy and improve efficiency as well. For instance, the planner can try to choose the fastest path for an autonomous cart that has to transports goods and deliver them.

Both methods are studied on a known environment given by Bosch.

1.2 Autonomous driving

Autonomous driving can provide a fundamental contribution to the solution of greater road safety and climate change. A study of Waymo Simulated Driving Behavior in Reconstructed Fatal Crashes within an Autonomous Vehicle Operating Domain illustrates that the Waymo autonomous car prevents a collision with an estimated of 82% better than an human driver, it reduces 10% of crash-level serious injury risk and it has the same behaviour of a human driver with an estimated of 8% [4].

The World Health Organization evaluated that almost 1.3 million people die each year due to road traffic crashes. A 2015 National Highway Traffic Safety Administration report found that human error causes 94 percent of traffic accidents [5]. Number that is likely to decrease considerably because computers have no emotion, they can not drink or being distracted consequentially they reduce and eliminate the operator errors. Self-driving development depend above people's trust and reliable solutions and in order to work, a vehicle has to be able to perceive and understand its surroundings ("Sense"), process the information, plan a driving strategy ("Think"), and safely implement the planned driving strategy ("Act").

Leonardo da Vinci (1500) designed a cart that is considered the world's first robot. The cart could move along a predetermined path without being pushed or pulled; power is provided by springs under high tension and steering is set up in advance.

Robert Whitehead (1869) invented a torpedo that propelled itself underwater thanks to a pressurization system called "The Secret" for several hundred yards underwater and maintained depth. Torpedo guidance led to a wide range of autonomous devices.

Going forward, technology and knowledge advanced and autonomous projects evolved (Fig. 1).

Nowadays, a lot of vehicles are considered semi-autonomous because they have a wide range of safety features like braking systems, assisted parking and the ability to drive, brake, steer and park themselves. Their technologies rely on GPS capabilities and sensing systems to detect lanes, obstacles, road signs and traffic lights. Companies are racing to build autonomous vehicles for a radically changing consumer world.

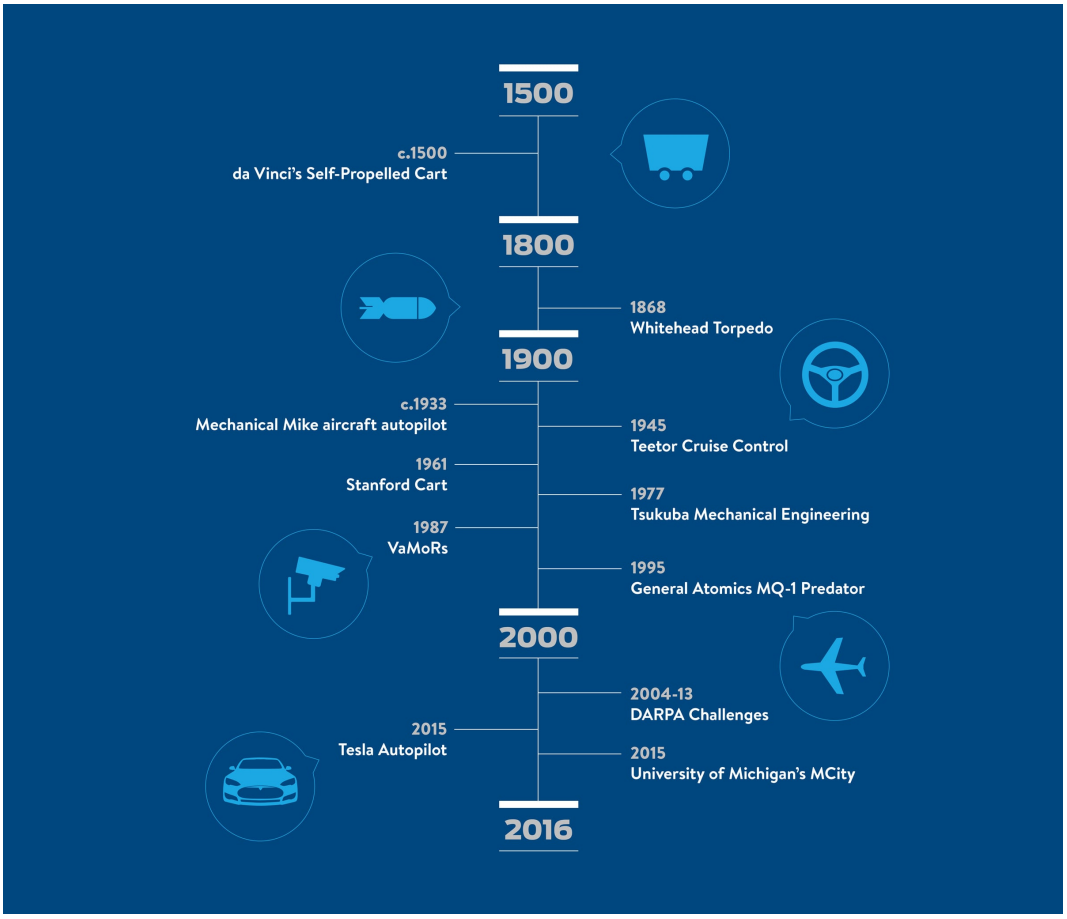

Figure 1: Timeline of autonomous vehicles

1.3 Levels driving automation

The Society of Automotive Engineers (SAE J3016, 2018) provide six levels driving automation from Level 0 to Level 5, from fully manual driving to fully autonomous (Fig. 2):

- Level 0 (No Driving Automation): vehicles are manually controlled. Human drivers control all the dynamic tasks associated with driving.
- Level 1 (Driver Assistance): human drivers are in full control of the vehicle but they are assisted by the single automated system (steering or accelerating).
- Level 2 (Partial Driving Automation): the vehicle can control both steering and accelerating/decelerating through advanced driver assistance systems or ADAS but driver can still manage the car.
- Level 3 (Conditional Driving Automation): vehicles can take some decisions such as emergency braking without human judgment using sensors such as LiDar. They still require human that must be on standby in case the system is unable to execute the task.
- Level 4 (High Driving Automation): vehicles can make decisions and can intervene if there is a system failure. They do not need human interaction in most circumstances but driver still has option to manually override and they can move only within a limited area. This is known as geofencing.
- Level 5 (Full Driving Automation): vehicles do not require human interactions and they can go anywhere being free from geofencing.

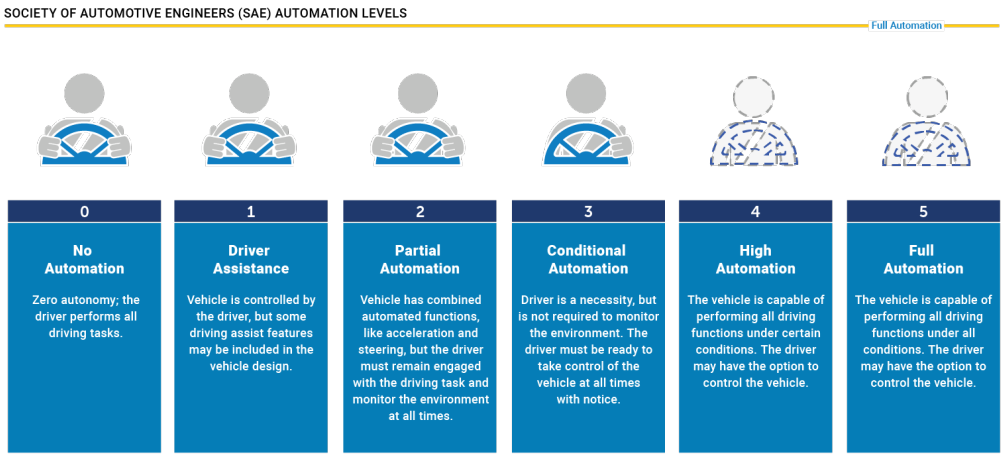

Figure 2: Automation levels of Society of Automotive Engineers (SAE).

1.4 Thesis outline

In this thesis, trajectory planning and path planning are studied in deep. Chapters are organised as follows:

Chapter 2: Introduction to Bosch Future Mobility Challenge (Fig. 3) and details about the project, including software and hardware components.

Chapter 3: Explanation of the communication between files, including crucial Python utilities and communication with LAN and UDP.

Chapter 4: Information about technology behind autonomous driving with a deepening on sensors and localization system. Description of sensors, explanation of the localisation system and corresponding algorithms used for competition.

Chapter 5: Technicality and algorithms about trajectory and path planning. Explanation of trajectory planning algorithm used in competition. Deepening of Dijkstra algorithm and A* algorithm and code explanation.

Chapter 6: Introduction to the parking task and description of parking algorithm used for the challenge.

Chapter 7: Conclusions and future development about the work are reported.

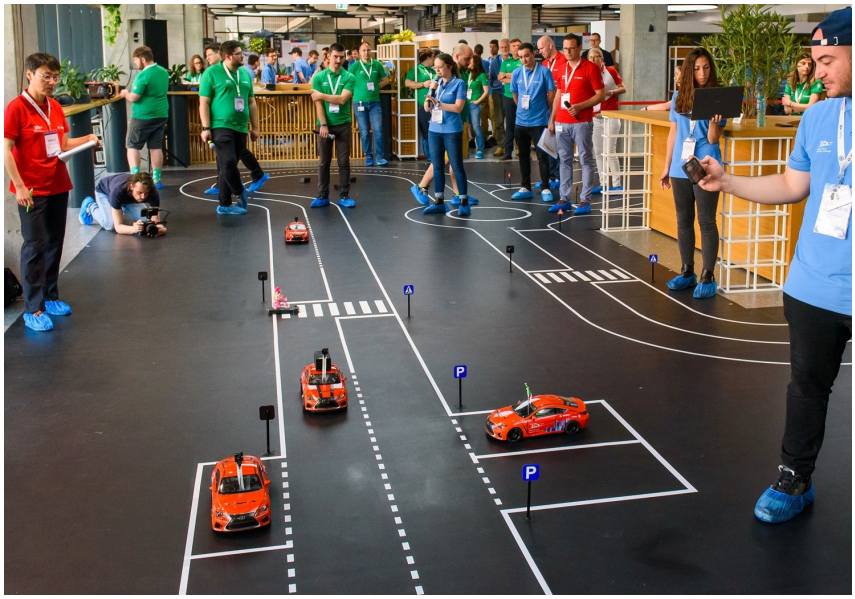

Figure 3: Car of the team to the finals.

2 Bosch Future Mobility Challenge

2.1 Introduction

This work has been realized based on the effort and the assignments performed during the participation to the so-called *Bosch Future Mobility Challenge* (BFMC): it is an international autonomous driving and connectivity competition for bachelor and master students organized by *Bosch Engineering Centre Cluj* since 2017. The competition invites student teams from all over the world every year to develop autonomous driving and connectivity algorithms on $1:10$ scaled vehicles, provided by the company, to navigate in a designated environment simulating a miniature smart city. The students work on their projects in collaboration with Bosch experts and academic professors for several months to develop the best-performing algorithms.

The author of this work has joined the challenge under the team *PoliTron* (Fig. 4) composed by 4 other colleagues from the master's degree program in Mechatronic Engineering of the Polytechnic of Turin, with the guidance of the supervisor himself, Professor Stefano Malan.

Figure 4: PoliTron team logo.

The job to carry out during the challenge, which lasts from November to May, consists in developing the algorithms involved in the realization of the autonomous car guide and implementing them into the received car, therefore it commits both the software and the hardware parts. All in all, it is a real and complete accomplishment of self-driving car.

2.2 The Competition

The competition requires that, in addition to the activities carried out by the teams to achieve the final objective, participating teams send a monthly periodic status via the competition website containing the followings to show their progress to the Bosch representatives:

- A **technical report** describing the development in the last sprint.
- A **project plan** alongside with the **project architecture**.
- A **video file** emphasizing with visual aid the contributions from the past month activity (already present in the report and project plan).

In the middle of the competition, on middle March, a first eliminatory phase takes place, the Mid-Season Quality Gate, in which each team is requested to send a 3-minutes (at most) long video in which the car must perform the following tasks in a single autonomous run:

1. Lane keeping.
2. Intersection crossing.

3. Complete manoeuvre after the following signs:

   3.1. *Stop* sign – the car must stop for at least 3 s.

   3.2. *Crosswalk* sign - the car must visibly reduce the speed and if a pedestrian is crossing, the car must stop.

   3.3. *Priority Road* sign - act normal, you are on a priority road and any vehicle coming from a non-priority road should stop.

   3.4. *Parking* sign - it is found before and after a parking lot and, if empty, can be used to perform the parking manoeuvre.

These tasks can be demonstrated by means of one of three possible alternatives:

- A video of the car performing the actions on a real-life like map.
- A video of the car in front of a Desktop, taking a video as a simulated input and acting accordingly.
- A video of the car in front of a Desktop where the simulator is running, taking as visual input the one from the camera inside the simulator.

The author's team has chosen the first option, realizing physically the track shown in Figure 5.

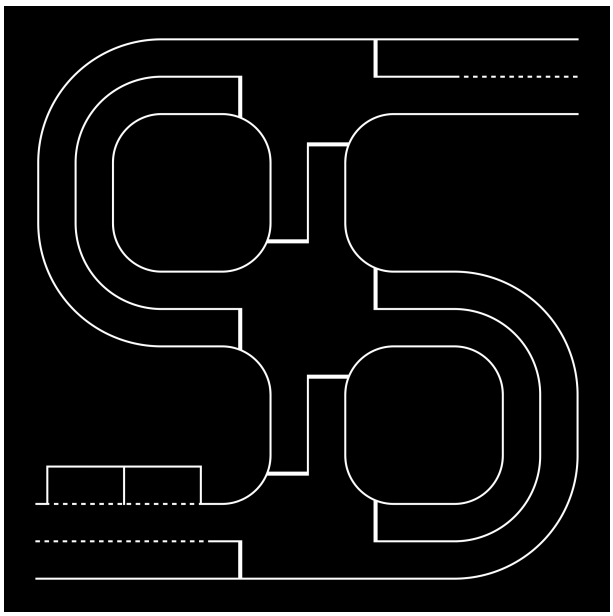

Figure 5: Test Track.

Based on the videos, the jury will decide which teams possess the right skills to continue the competition and to go to the Bosch Engineering Centre site in Cluj-Napoca (Romania) for the qualifications and possibly semifinals and finals in May.

During the race period in Romania, the teams will have to face two challenges: the technical and the speed one. The former requests that the car can correctly respect most of the road signs, such as traffic signs, traffic lights, lanes, intersections, ramps, and roundabouts. Moreover, it must detect pedestrians and overtake other cars present in the same lane. The latter asks the car to complete a determined

path in the shortest time possible, this time respecting only the lanes and the road markings. In addition to this, the teams will make a presentation in front of the jury.

Only a maximum of 8 teams will be selected to participate to the final race, in which the first 3 qualified teams will win both a money prize and the car kit, and another team, not included in the top 3, will be rewarded as the "best newcomer", meaning a team which did not take part to the competition in the previous year. All the phases of the challenge are reported in Figure 6.

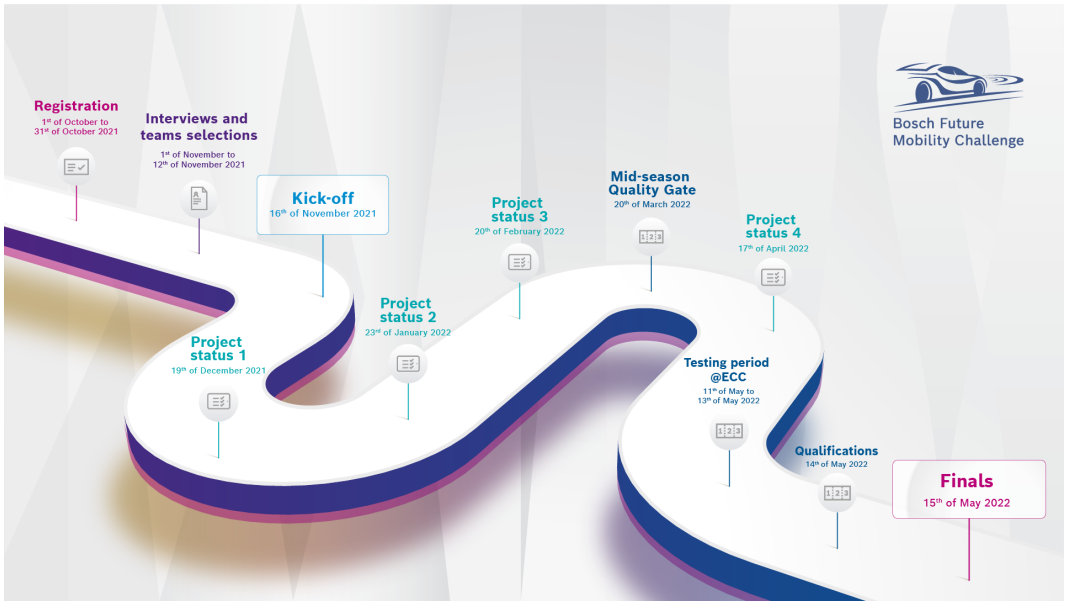

Figure 6: Bosch Future Mobility Challenge 2022 - Timeline.

The author's team managed to reach the finals and competed with other 7 talented teams from Greece, Romania, Portugal and Italy and won the Best new participating team award (Figure 7).

Figure 7: Bosch Future Mobility Challenge 2022 - Best New Participating Team Award.

2.3 The Car-Kit

Going into the details of the car kit provided by Bosch, the following components are to be found:

- Nucleo F401RE.
- Raspberry Pi 4 Model b.
- VNH5012 H-bridge Motor Driver.
- ATM103 Encoder.
- DC/DC converters.
- Servomotor.
- LiPo Battery.
- Chassis.
- Camera.
- IMU Sensor.

The fundamental components are shown in Figure 8.

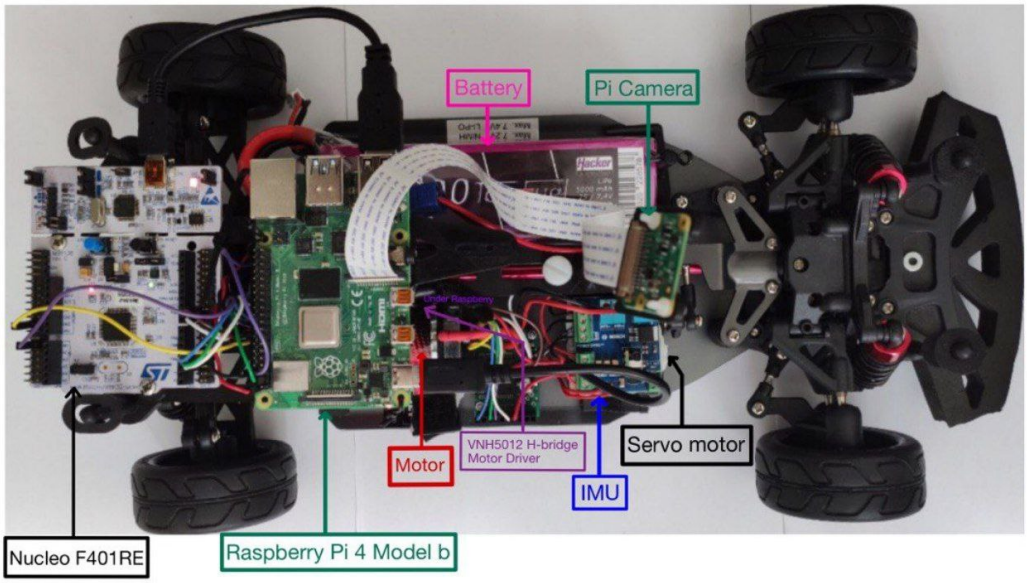

Figure 8: Car with the listed Components.

In addition to these basic elements, the team decided to furnish the car with a LiDAR sensor and an Ultrasonic sensor, placed respectively in the front and the right-hand side of the car.

2.4 The Project

To start working on the project, the teams are provided with a complete documentation necessary to understand better the structure of the project, especially the hardware side and the base Python/C++ codes for the correct communication of all the components of the car. The cited documentation is subdivided as shown in Figure 9.

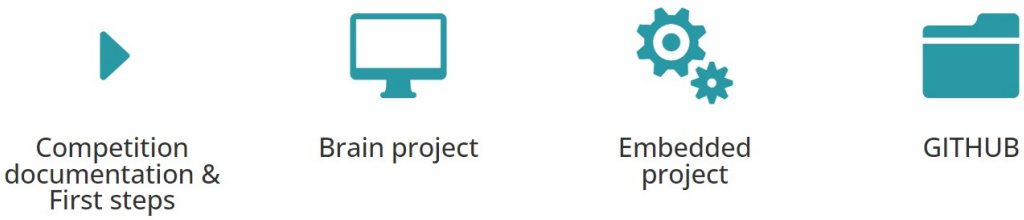

Figure 9: BFMC website - Layout of the Shared Documentation.

A brief explanation of the content of each section is reported in the following subchapters.

2.4.1 Competition Documentation and First Steps

It includes:

- Connection diagram and description with official links to the components of the car.

- Racetrack: the description of the provided racetrack and its elements, the given components, and the diagrams, as well as a starting point and directions of the knowledge required.
- V2X-Vehicle to everything: it includes localization, semaphore, environmental server, and vehicle-to-vehicle communication.
- Printed components and circuit boards.
- Hardware improvements: it includes settings for the hardware components.
- Useful links for Raspberry Pi, ROS, and Python.
- Periodic status: project plan and architecture, reports and media.

2.4.2 Brain Project

The Brain Project describes the given code for the RPi platform. It includes the start-up code and the documentation for the provided API's, which will help the interaction with the V2X systems. The project uses concepts of multi-processing and distributed system, and it implements a basic flexible structure, which can be extended with new features. This folder contains:

- Introduction: concept and architectures, in particular remote car control and camera streaming, installation and configuration, IMU displayer.
- Utils layer: camera streamer, remote control.
- Hardware layer: camera, serial handler process and camera spoofer process.
- Data acquisition layer: traffic lights, localization system, environmental server.

The computer project is already implemented on the provided Raspberry Pi, while the embedded project is already implemented on the Nucleo board. Together, they give a good starting point for the project, providing a remote keyboard control, remote camera stream, constant speed control of the given kit and others.

2.4.3 Embedded Project

This documentation describes the low-level application which runs on the micro-controller Nucleo-F401RE. It aims at controlling the car movement and providing an interface between higher level controllers and lower-level actuators and sensors.

The project has four parts:

- Tools for development containing the instructions to upload the codes related to the correct functioning of the Nucleo.
- Brain layer contains the state machine of the Nucleo (speed and steering).
- Hardware package includes the drivers for actuator and sensors.
- Signal, utils and periodics namespace: 'signal' includes libraries for processing signals, 'utils' package incorporates some util functionalities and 'periodics' layer includes some periodic tasks.

2.4.4 GITHUB

Bosch provided their own link of GitHub in which all the Python/C++ codes related to the topics described above are held. Specifically:

- `Brain` and Brain_ROS: the project includes the software already present on the development board (Raspberry Pi) for controlling the car remotely, use the API's and test the simulated servers, respectively for Raspbian and ROS.
- `Startup\_C`: the project includes some of the scripts transcribed in C++ language from the startup project.
- `Embedded\_Platform`: the project includes the software already present on the Embedded platform (Nucleo board). It describes all the low-level software for controlling the speed and steering of the car.
- `Simulator`: the project includes the software for the Gazebo simulator, which is the official on-line environment of the competition.
- `Documentation`: the project includes all the general documentation of the competition environment, guides, diagrams, car components, etc.

2.5 The Structure behind the Algorithms

The tasks to perform by the end of the competition are the following:

- Lane Keeping and Following.
- Intersection Detection and crossing.
- Correct manoeuvres under the detection of the following traffic signs: stop, priority, crosswalk, parking, roundabout, highway entrance and highway exit, one-way, no entry.
- Parallel and perpendicular parking.
- Object Detection: pedestrian and overtake of a static and/or moving vehicle.
- Navigation by means of nodes and localization system (GPS).

The brain of the car must be inserted in the Raspberry Pi which, basing on the tasks to perform, sends the commands to the Nucleo which, in turn, acts on the motor and on the servo motor to regulate both the speed and steer. More in details, in order to process the image, the Raspberry takes as input the camera frame and the IMU data for the position of the vehicle, runs the specific control algorithms and sends the corresponding output commands to the Nucleo; for example, an increased speed in presence of a ramp which signs the entrance to the highway, a decreased speed and a specific steer when traveling along a tight curve and a zero speed when the traffic light turns red. The correlation between all the project components, the sensors, the algorithms and the vehicle actuation is represented in the project architecture shown in Figure 10.

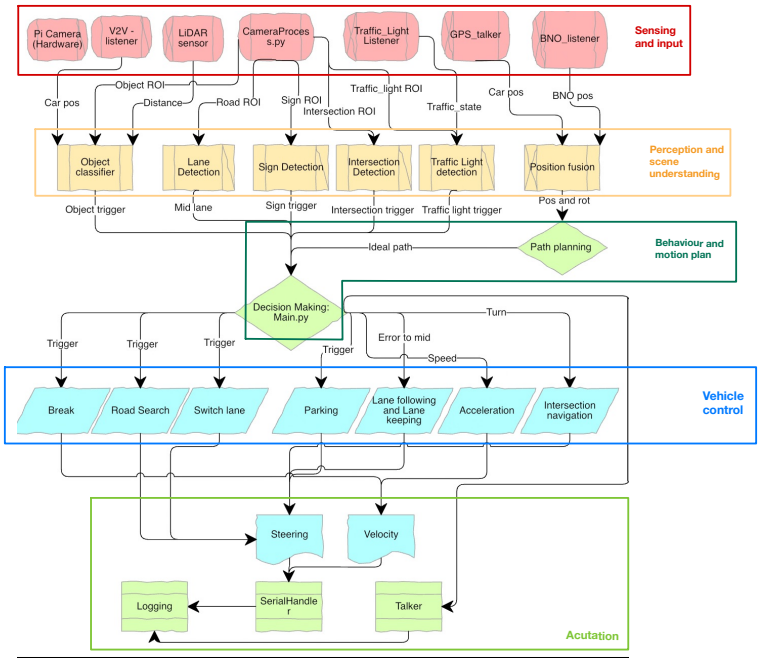

Figure 10: Architecture of the completed project.

The project presented in this chapter sinks the roots for the work developed by three members of team PoliTron: specifically, Cristina Chicca deals with the image processing part, Gianmarco Picariello's work consists in the development of MPC controllers in this context and Claudia Viglietti's thesis concerns optimization algorithms for path planning.

3 Files communication

Before diving into the algorithms dealing with the paper matter, it is of particular interest to define appropriately how the files shown in Figure 10 communicate, both by means of Python3 and using a given server responsible for the car localisation system.

3.1 Parallelism, Thread and Processes

The whole project is composed by processes and threads which ensure that all the algorithms present on the Raspberry Pi run correctly and in parallel since the car, to perform a correct self-drive, needs to execute them concurrently: for example, it has to always follow the lane while respecting traffic and road signs, checking for pedestrians crossing the street etc.

Python multiprocessing library offers two ways to implement process-based parallelism:

- Process: used when functions-based parallelism is required.
- Pool: offers a convenient means of parallelizing the execution of a function across multiple input values, distributing the input data across processes (data-based parallelism).

In this project case, the **Process** method has been used: when it is run, it has a specific address on the memory and all the used variables are accessible only by the same process, so they cannot be read by another process unless a pipe is used.

A **Pipe** is a method to pass information from one process to another one: it offers only one-way communication and the passed information is held by the system until it is read by the receiving process. What is returned from the *pipe()* function is a pair of file descriptors (r,w) usable for reading and writing respectively.

In addition to the Process module, the multiprocessing module offers the threading module. The **Thread** class represents an activity that is run in a separate thread of control. This function represents the capability of Python to handle different tasks at the same moment: to sum up, it is a separate flow of execution in the sense that Python script appears to have more threads happening at once. Its syntax is the following:

Thread(group=None, target=None, name=None, args=(), kwargs=, *, daemon=None)

In particular, *target* is the callable object to be invoked by the *run()* method, *name* is the thread name and *args* is the argument tuple for the target invocation. Moreover, the *.daemon* property ensures that the daemon thread does not block the main thread from exiting and continues to run in the background.

Multiple threads work in parallel as shown in Figure 11.

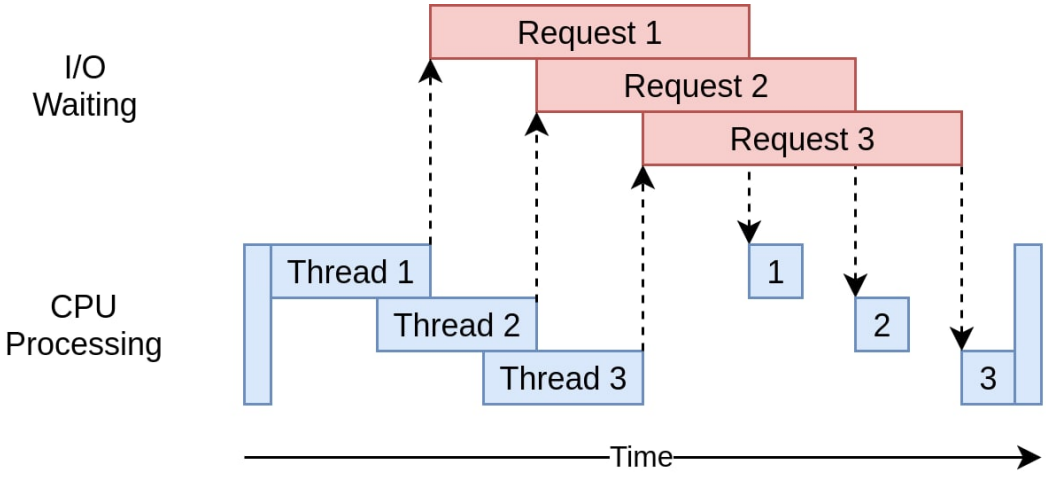

Figure 11: Threads parallelism.

In order to give an idea of how all these functions are related to one another inside a working script, an example in the context of autonomous drive follows: it is necessary to set the commands to send to the STM32 Nucleo microcontroller on a dedicated process and, by means of a pipe, these commands are sent to the *SerialHandlerProcess*, which deals with the communication with the Nucleo. It implements the *WriteThread* function to send commands to Nucleo and the *ReadThread* function to receive data from Nucleo. The commands are generated after having processed the images which come from the *CameraProcess* implementing the Raspberry Pi Cam and they are sent to the *LaneDetectionProcess*, responsible for the lane detection. This means that *LaneDetectionProcess* has to receive two pipes, one for receiving the images and one for sending the commands.

The project contains many processes, each defining a particular algorithm for the self-driving control: the definitions explained until now are useful to outline a structure in common with all these processes, which is shown below.

```
class Name(WorkerProcess):

    def __init__(self, inPs, outPs):
        """ Process used for sending images over the network to a targeted
            IP via UDP protocol (no feedback
        required). The image is compressed before sending it.

        Used for visualizing your raspicam images on remote PC.

        Parameters
        ----------
        inPs : list(Pipe)
            List of input pipes, only the first pipe is used to transfer the
                captured frames.
        outPs : list(Pipe)
            List of output pipes

        In this section you can also define the variables initialization.
        """

        super(MainLaneDetectionProcess, self).__init__(inPs, outPs)
```

```
    def run(self):
        """ Apply the initializing methods and start the threads. """

        super(MainLaneDetectionProcess, self).run()

    def _init_threads(self):
        """ Initialize the sending thread. """

        # Thread for elaborating the received frames:
        receiveFrameT = Thread(name='receiveFrameThread',
            target=self._generate_Output,
                               args=(self.inPs, self.outPs,))
        receiveFrameT.daemon = True
        self.threads.append(receiveFrameT)
```

After creating the class whose name is subjective (in this case it is called *Name*), the utilized functions and methods are the following:

- `__init__`: takes as input `self`, `inPs` and `outPs` which correspond to the list of input pipes and the list of output pipes respectively. This function is called when a class is "instantiated", meaning when the class is declared and any argument withing this function will also be the same argument when instantiating the class object. These initial arguments are the data manipulated throughout the class object. Under this function, some instance attributes are defined and assigned to `self` to be manipulated later on with other functions.
- `super()`: it inherits, uses code from the base (existing) class (i.e., *WorkerProcess*) to define the structure of the new class (i.e., *Name*) – it guarantees the access methods from a parent class within a child class reducing repetitions in the code.
- `run()`: function that initializes the sending thread for the processing of received frames.
- `.append()`: adds a single item to the existing list. It does not return a new list of items, but it will modify the original list by adding the item to the end of the list. After executing the method `append` on the list, the list size increases by one.

All of them send their output to a particular process called *MovCarProcess*: it is responsible for setting the correct values of steer and speed of the car according to the road situation, e.g the value of the lane curve, the detected traffic sign, the intersection etc.. These values are integers representative of the manoeuvre: for example, a value of 999 corresponds to speed equal to 0 in the *SpeedThread*. Summarizing, *MovCarProcess* sets the representative value according to the output received from the control processes, whereas *SpeedThread* and *SteerThread* contain the actual command sent to the Nucleo for, respectively, speed (action 1) and steer (action 2). An example is given by the code shown below, in which the *MovCarProcess* sets the value by means of which the car stops in presence of a STOP or CROSSWALK sign and both *SpeedThread* and *SteerThread* actually build the physical command.

```
""" Extract from MovCarProcess """
    if STOP or CROSSWALK:
```

```
        value = 999

""" Extract from SpeedThread """
    #Stop
    if curveVal == 999:
        command = {'action': '1', 'speed': 0.0}

""" Extract from SteerThread """
    #Stop
    if curveVal == 999:
        command = {'action': '2', 'steerAngle': 0.0}
```

Similarly, these threads will set speed and steer values different from 0 whenever the car has to travel along the path, in absence of road and traffic signs that would impede it.

3.2 The Main.py file

All the processes which have to be run on the Raspberry Pi, including their inputs and outputs, the way in which they communicate, are described inside the *main.py* file. As every main function, it has the job to put together the functions involved in the autonomous-driving solution, searching them from their specific folder.

```
    ArcShRead, ArcShSend = Pipe(duplex=False) # for serial handler

    FrameRead1, FrameSend1 = Pipe(duplex=False) # Frame towards Lane
        Detection
    FrameRead2, FrameSend2 = Pipe(duplex=False) # Frame towards
        Intersection Detection
    FrameRead3, FrameSend3 = Pipe(duplex=False) # Frame towards Sign
        Detection
    FrameRead4, FrameSend4 = Pipe(duplex=False) # Frame towards
        Localization Process

######## IMAGE PROCESSING ALGORITHMS ##########
    curveValRead, curveValSend = Pipe(duplex=False)
    IntersectionRead, IntersectionSend = Pipe(duplex=False)
    SignDetRead, SignDetSend = Pipe(duplex=False)

######## LOCALISATION ALGORITHMS ##########
    LocalizationRead1, LocalizationSend1 = Pipe(duplex=False)

######## PROCESSES ##########
    AshProc = SerialHandlerProcess([ArcShRead], []) #receives the data from
        MovCar and sends it to the Nucleo
    allProcesses.append(AshProc)

    AcamProc = CameraProcess([], [FrameSend1, FrameSend2, FrameSend3,
        FrameSend4])
    allProcesses.append(AcamProc)

    ALaneProc = MainLaneDetectionProcess([FrameRead1], [curveValSend])
    allProcesses.append(ALaneProc)

    AInterProc = IntersectionDetectionProcess([FrameRead2],
```

```
    [IntersectionSend])
allProcesses.append(AInterProc)

ASignProc = SignDetectionProcess([FrameRead3],[SignDetSend])
allProcesses.append(ASignProc)

AtrajProc = RaceTrajectoryProcessSO([FrameRead4], [LocalizationSend1])
allProcesses.append(AtrajProc)

AEnvProc = EnvironmentalProcessSO([LocalizationRead1], [])
allProcesses.append(AEnvProc)

AcurveValProc = MovCarProcess([curveValRead, IntersectionRead,
    SignDetRead, LocalizationRead1],[ArcShSend])
allProcesses.append(AcurveValProc)
```

The example above shows an extract from the *main.py* file: a **pipe** is defined for every process which has to receive the frame from the camera as input (in this case, there are 4 processes which require it) and also, a pipe for localisation and serial handler data is defined. Then, every process is declared, in the first brackets the inputs are listed, whereas in the second brackets the outputs are listed. *CameraProcess* has no input but only the frames to send as output, whereas *SerialHandlerProcess* has the output of *MovCarProcess* as input and no output. It is important to highlight that not all the processes receive as input the camera frames: *EnvironmentalProcess*, responsible for sending the encountered objects to the server, receives as input the coordinates of the car from the *RaceTrajectoryProcess*, whereas *MovCarProcess* receives the inputs from the other processes (car localisation and objects detection) and sends the commands to *SerialHandlerProcess*.

3.3 Server Communication and UDP

The car has an indoor localisation system which detects and sends by **UDP connection** the relative position of itself and other cars present on the race track.

The UDP communications describe the programming for the User Datagram Protocol provided in the TCP/IP to transfer datagrams over a network. Informally, it is called "Send and Pray" because it has no handshake, session or reliability, meaning it does not verify that the protocol has reached the destination before it sends data. UDP has a 8-byte header that includes source port, destination port, packet length (header and data) and a simple (and optional) checksum.

The checksum, when utilized, provides limited integrity to the UDP header and data since it is simply an algorithm-based number created before data is sent to ensure data is intact once received: this procedure is done by running the same algorithm in the received data and comparing the number before and after the reception.

UDP avoids the overhead associated with connections, error checks and retransmission of missing data, it is suitable for real-time or high performance applications that does not require data verification or correction. In fact, the IP network delivers datagrams that can be up to 65507 bytes in length but does not guarantee that they are delivered at the destination and in the same order as they are sent. Moreover, UDP provides pre-process addressing through ports where IP provides addressing of a specific host. The process is described as follows:

1. These ports are 16-bit values used to distinguish different senders and receivers at each end point.

2. Each UDP datagram is addressed to a specific port at the end host and incoming UDP datagrams are demultiplexed between the recipients.

The advantage of using UDP is the absence of retransmission delay, meaning it is fast and suitable for broadcast. The disadvantage regards no guarantee of packets ordering, no verification of the readiness of the receiving computer and no protection against duplicate packets. Anyway, UDP is often used for streaming-type devices such as lidar sensors, cameras and radars since there is no reason to resend data if it is not received. Moreover, due to high data rates, resending past and corrupted data would slow things down tremendously. A comparison between TCP and UDP is given by Figure 12.

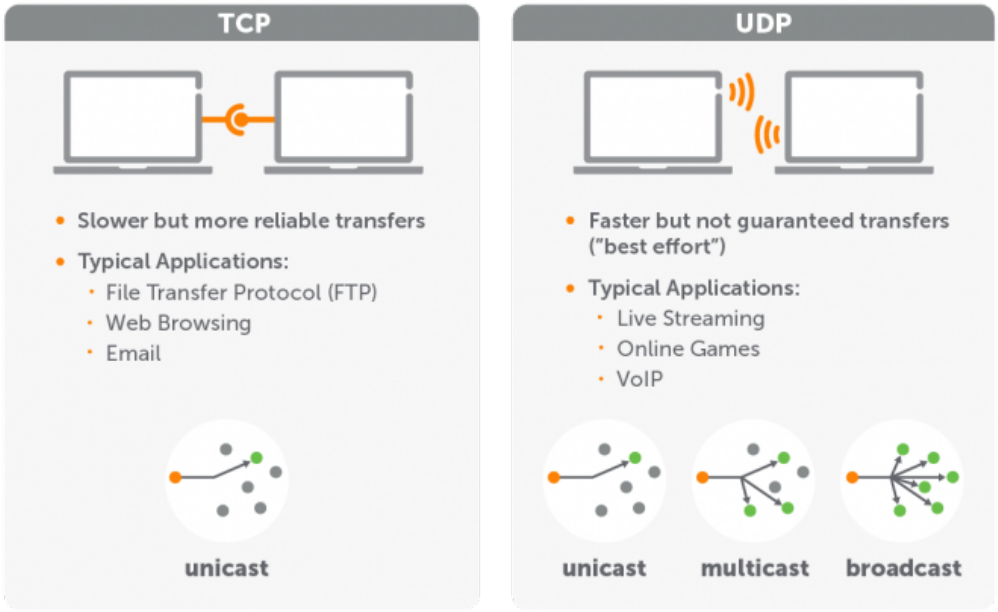

Figure 12: Comparison between TCP and UDP protocols.

The connection between the car and the server is validated by means of the **API communication**, which ensures the reading of the car given ID together with a certain port responsible for the communication of the coordinates of all the moving obstacles. An API communication is a type of Application Programming Interface which adds communication channels to a particular software. It allows two pieces of software hosted on the cloud to connect to each other and transfer information.

An example of the UDP protocol used inside the project is given by a file responsible for reading the position of the car in real time (*position_listener.py*).

```
import sys
sys.path.insert(0,'.')

import socket
import json
from complexDealer import ComplexDecoder
```

```
class PositionListener:
   """PositionListener aims to receive all message from the server.
   """
   def __init__(self, server_data, streamPipe):

      self.__server_data = server_data

      self.__streamP_pipe = streamPipe

      self.socket_pos = None

      self.__running = True

   def stop(self):
      self.__running = False
      try :
         self.__server_data.socket.close()
      except: pass

   def listen(self):
      while self.__running:
         if self.__server_data.socket != None:
            try:
               msg = self.__server_data.socket.recv(4096)

               msg = msg.decode('utf-8')
               if(msg == ''):
                  print('Invalid message. Connection can be interrupted.')
                  break

               coor = json.loads((msg),cls=ComplexDecoder)
               self.__streamP_pipe.send(coor)
            except socket.timeout:
               print("position listener socket_timeout")
               # the socket was created successfully, but it wasn't received
                   any message. Car with id wasn't detected before.
               pass
            except Exception as e:
               self.__server_data.socket.close()
               self.__server_data.socket = None
               print("Receiving position data from server " +
                   str(self.__server_data.serverip) + " failed with error: "
                   + str(e))
               self.__server_data.serverip = None
               break
      self.__server_data.is_new_server = False
      self.__server_data.socket = None
      self.__server_data.serverip = None
```

Similarly to the Process object, the class `PositionListener` is composed by the main functions `__init__`, `stop` and `listen`. In this case, the variables of interest are `server_data`, `streamPipe` and `socket`.

A network socket is a software structure within a node of a computer network that serves as an endpoint for sending and receiving data. The structure and properties of a socket are defined by an API for the networking architecture. Sockets are created

only during the lifetime of a process of an application running in the node.

The function `listen` performs the following steps:

1. After the subscription on the server, it is listening the messages on the previously initialized socket.
2. It decodes the messages and saves in 'coor' member parameter.
3. Each new message will update the 'coor' parameter and the server will send the result (car coordinates) of last detection. If the car has been detected by the localization system, the client receives the same coordinates and timestamp.

The UDP socket programming fundamentals are represented by Figure 13.

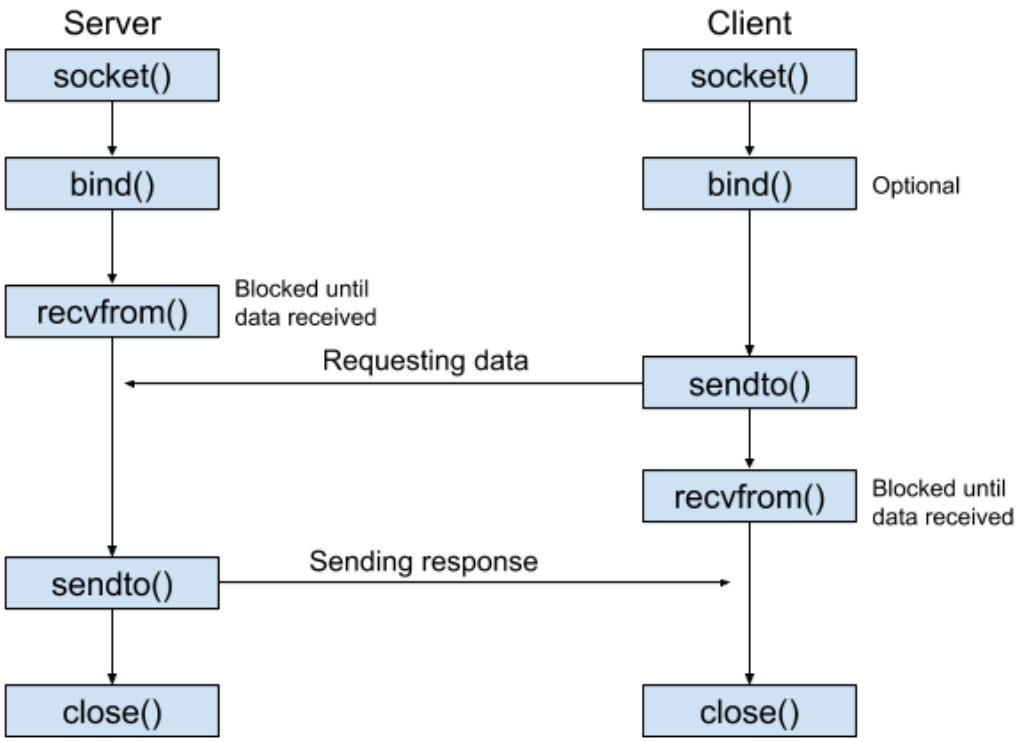

Figure 13: Fundamentals of UDP socket programming.

4 Technology behind autonomous driving

Autonomous vehicles (AVs) use "sense-plan-act" design. AVs are equipped by sensors like camera, LiDar, ultrasonic, radar and infrared to sense the environment (Fig. 14). A range of sensors in combination can be complementary and compensate for any weaknesses in any other sensor.

Sensors can degrade their performance because of their limitations and inadequacies. Errors are due to drifting errors, surface irregularities, wheel slipping, low sensor resolution or uncertainty in readings. Better is the accuracy of the sensors fewer are the limitations and higher are the costs [6].

To plan, autonomous cars can use a blend of the Global Positioning System (GPS) and Inertial Navigation Systems (INS) so that the vehicle can localize its position. Both GPS and INS can have some uncertainties and they can be inaccurate, for this reason it is important to take into account their limits. After simulations and field testing, important parameters are set and control is managed through rule-based controllers. The drawback is the difficulty to generalize new scenarios, the huge time required to tune the parameters and the non linear behavior of driving that imply linearization of the vehicle model. Technology advanced and the number of sensors can be reduced with the adoption of Convolutional Neural Networks (CNNs) provided by raw camera inputs.

Increasing the use of deep learning, it allows to developers to teach to vehicle the system to accomplish and the control achieves numerous benefits such as the ability to adapt to new scenarios [7], [8].

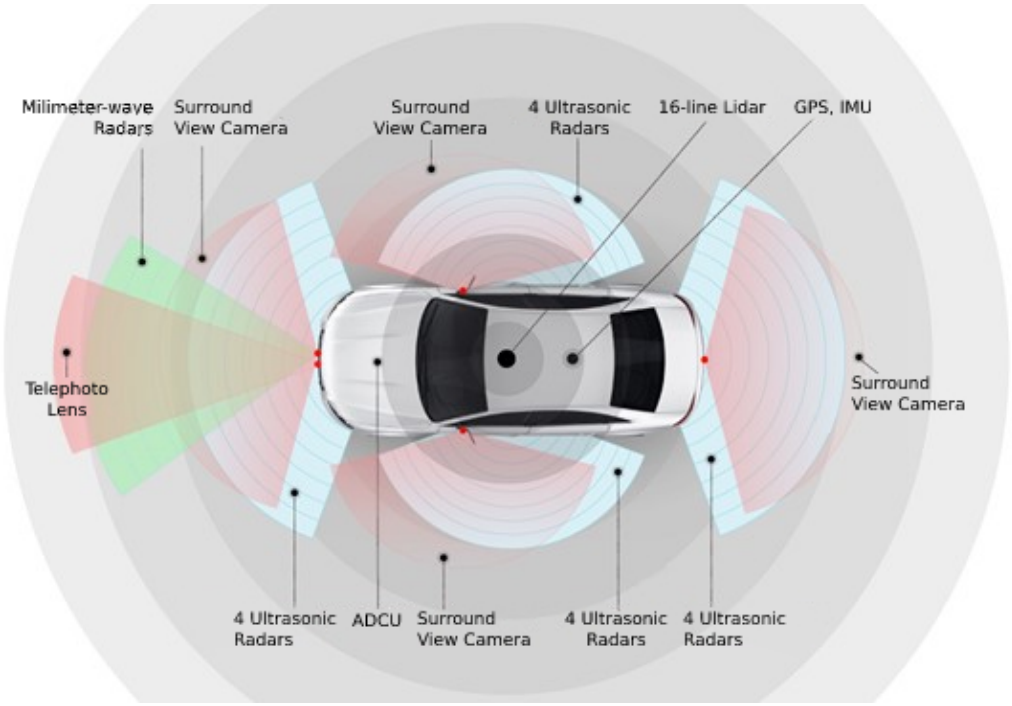

Figure 14: Autonomous Driving Sensors

4.1 Sensors

Sensors are devices, module, machine or subsystem that produce an output signal for the aim of sensing events or changes in the environment. They are vital because

they grant to autonomous vehicles to plan their routes securely, supervise their surroundings and identify oncoming impediments. Thanks to sensors, the automation system combines automotive software and hardware and it takes full control of the vehicle [9]. Sensors are divided into:

- Internal state, or proprioceptive sensors: they detect internal data wheel load, angular rate, force and stores the dynamical state of a dynamic system. Some examples of internal states are: encoders, Inertial Measurement Units (IMU), gyroscopes.
- Exteroceptive sensors, or external state sensors: they receive and collect information from the system environment. For instance, they perceive information about light intensity or distance measurements. An example of exteroceptive sensors are cameras, LiDar, ultrasonic sensors, radar.

They can have two kind of nature:

- Passive: they obtain energy from environment and they provide the corresponding output. An example of passive senors are the vision cameras.
- Active: they release energy into the environment and they detect the correlative environmental feedback. An example of active senors are LiDAR and radar sensors.

They are also split according to the wireless technology transmission range: short-range, medium-range and long-range.

4.2 Detailed study about sensors

This section analyzes benefits and drawbacks of a generic sensor that can be used in autonomous vehicle and it goes in deep into the ones used for the Bosch Future Mobility challenge.

4.2.1 Camera

In autonomous vehicle, cameras are the most used technology to analyze the surrounding. They generate images of the approaching environment, such as a pedestrian crosswalk, and they can operate in different weather condition. They can be classified as:

- Visible cameras (VIS): similar to human eyes, they capture wavelengths that ranges from 400 to 780 nm [10]. Combination of more visible cameras let stereo vision to be performed. They are used for their ability to distinguish colors, high resolution and low cost in spite of their low estimated depth accuracy.
- Infrared cameras (IR): they work with infrared wavelengths ranging between 780 nm and 1 mm and they can be extended to the near-infrared (NIR: 780 nm–3 mm) and the mid-infrared (MIR:3–50 mm; known as thermal cameras) [10]. They are less susceptible to lighting or to weather conditions than visible camera so they can overcome situation where VIS fails and they can detect warm bodies as pedestrians.

Bosch gives to the team the Raspberry camera board: Pi Camera module v2.1 (Fig. 15). It is a high quality 8 megapixel camera provided with fixed focus lens. It is capable of 3280 x 2464 pixel static images and it supports 1080p30, 720p60 and 640x480p90 video using the Sony IMX219PQ image sensor.

Team used camera for lane keeping, intersection detection, traffic sign detection, traffic light detection and object detection.

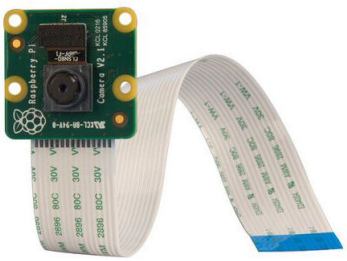

Figure 15: Pi Camera module v2.1

4.2.2 LiDar

The first attempt of the light detection and rangig (LiDar) was developed in the 1930s to measure air density profiles in the upper atmosphere by defining the scattering intensity from searchlight beams. In 1938, for the first time, pulses of light were utilised to calculate cloud base heights. In 1953, the acronym LiDar was introduced by Middleton and Spilhaus. The modern LiDar technology was born with the invention of the laser in 1960. The first commercial LiDar had 2000 to 25,000 pulses per second for topographic mapping applications [11].

Lidar is a distant sensing technique that targets a surface or an object with a laser light pulses with lengths of a few to several hundred nanoseconds and particular spectral properties. The equipment measures the time between emission and reception of the light pulses permiting distance estimate. It has airborne, terrestrial and mobile applications. The point cloud data (PCD) are the data that LiDar produces and they give object intensity information. Several systems make use of a beam expander to decrease the divergence of the light beam. LiDars use mirror telescope at the receiver end with lenses that can be used for small-aperture receivers. Emitter and receiver optics can have different geometric arrangement that determine the degree of signal compression. With short distance, only a part of the LiDar return signal is measured. Changing distance, beam diameter, shape, divergences this part changes.

There are several kind of LiDar, some of them are explained below:

- Elastic-backscatter LiDar: in its more manageable form it applies one laser releasing a single wavelength and one detector calculating the radiation flexibly

backscattered from the atmospheric particles. For example, it gives information about cloud layers.

- Differential-absorption LiDar (DIAL): it detects atmospheric gases with high sensitivity. It uses single broad absorption bands or absorption lines of gases. Differential-absorption LiDar generates two wavelengths where one is absorbed more powerfully than the other to determine the differential molecular absorption coefficient. The number concentration of the gas atoms can be evaluated if the differential absorption cross section is known. For instance, DIAL is used for the observation of water vapor. In general, differential-absorption LiDar must consider the Doppler broadening of the backscattered light and it requires spectral purity of the emitted laser light and high stability. DIAL uses the temperature-dependent strength of absorption lines of oxygen for temperature profiling where the differential absorption cross section is measured knowing the number concentration of the gas. If more than just a few nanometers spectrally separate the two DIAL wavelengths, differential backscattering is transformed into the bigger error source.

- Doppler LiDar: Direct-detection Doppler LiDars apply narrow-band spectral filters to evaluate frequency shift and it exploits the molecular backscatter component. Coherent Doppler LiDar detects the radiation backscattered from the moving particles and it emits single-mode single-frequency laser radiation. A local oscillator radiation is mixed generating the return signal for the sensor. Heterodyne detection is used to determine the sign of the shift.

In autonomous vehicle, LiDar is generally based on TOF, a pulsed laser that emits pulse singularly or continuously to the target triggering instantaneously internal timing circuit. To obtain the distance of the object, the calculator evaluates the time Δt between the laser pulse getting to the target and coming back to the receiver from the objective. Once the target is aimed, the laser emits light; at the same time, the emitting signal collected by the sampler allows the counter to counting and the clock oscillator loads the clock pulse to the counter. The echo signal gets into the receiving optical system, it is amplified by the amplifier, the photoelectric detector converts it into electric pulse and the counter stops counting. The clock pulse number entering the counter is calculated to get the target distance [12].

The challenge allows hardware improvement, team Politron decided to add a TF-Luna LiDAR Module. It measures the distance and it regularly releases near infrared modulated waves using Time of Flight (ToF) principle (Fig 16). In order to procure the relative distance D:

$$D = \frac{c}{2}\frac{1}{2\pi f}\Delta\phi \tag{1}$$

where f is the clock pulse frequency and c is the speed of light, time is measured by evaluating the phase difference $\Delta\phi$ between original and reflection waves.

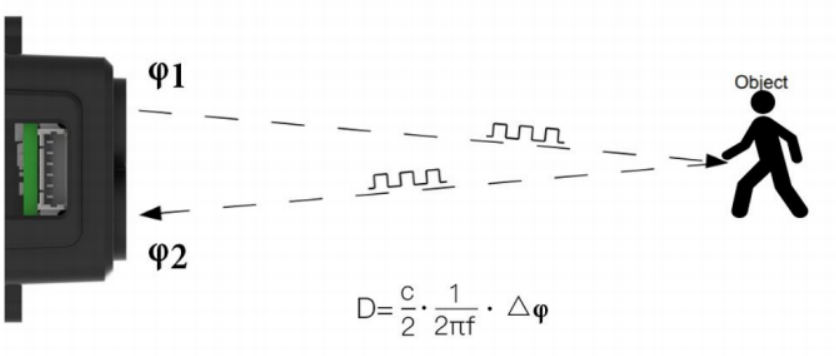

Figure 16: Schematic of ToF principle

In the competition, LiDar was mounted on the front of the car (Fig. 17), it detected the in-front obstacles like pedestrians and other vehicles.

Figure 17: LiDar mounted on the car of the team.

The algorithm perform a data fusion in *MovCarProcess* between *ObjectDetectionProcess*, the process that detects the objects in the environment and LiDar data. The team performed a sort of classification of the detected objects also according to the distance perceived by the LiDar: for example, pedestrians are detected with 20 cm distance, so whenever the LiDar detects an object within 20 cm, it stops since it is a pedestrian, whereas other cars on track are detected within 75 cm distance, since the car has to consider a safe margin in order to perform the overtake manoeuvre.

In the following code, it is shown what was explained before:

```
    def _lidarMeasure(self,):
        try:
            ser = serial.Serial("/dev/serial0", 115200, timeout=0) # mini
                UART serial device
            if ser.isOpen() == False:
                ser.open() # open serial port if not open
            distance, strength, temperature = self.read_tfluna_data(ser)

            if distance < 20.0: #cm
                flag_distance = 99
            elif distance >= 20.0 and distance < 30.0: #cm
                flag_distance = 2
            elif distance >= 30.0 and distance < 75.0: #cm
                flag_distance = 1
            else:
                flag_distance = 0
            ser.close()
            return flag_distance

        except Exception as e:
            print('Lidar communication error')
            print(e)

    def read_tfluna_data(self, ser):

        while True:
            counter = ser.in_waiting # count the number of bytes of the
                serial port
            if counter > 8:
                bytes_serial = ser.read(9) # read 9 bytes
                ser.reset_input_buffer() # reset buffer

                if bytes_serial[0] == 0x59 and bytes_serial[1] == 0x59: #
                    check first two bytes
                    distance = bytes_serial[2] + bytes_serial[3] * 256 #
                        distance in next two bytes
                    strength = bytes_serial[4] + bytes_serial[5] * 256 #
                        signal strength in next two bytes
                    temperature = bytes_serial[6] + bytes_serial[7] * 256 #
                        temp in next two bytes
                    temperature = (temperature / 8.0) - 256.0 # temp scaling
                        and offset
                    return distance, strength, temperature
```

4.2.3 Ultrasonic

Ultrasonic sensor or sonar is an electronic device composed by two main components:

1. transmitter: it uses piezoelectric crystals to emit sound.

2. receiver: it acquires the reflected sound.

Ultrasonic sensors make use of echolocation to determine the proximity of an object in the range of the sensor. It measures also the distance of the object evaluating

the time for the emitted signal to come back. It emits ultrasonic sound waves and transforms the reflected sound into an electrical signal. They lose accuracy due to noise interference or its blind zone at close proximity. Ultrasonic sensors can be divided into:

- Ultrasonic proximity sensors: they emit and receive sound waves at high frequency. They are composed by a sonic transducer which allows for alternate transmission and reception of sound waves. They sense the presence of any object, regardless of its material or surface properties. They are used for intermediate distances object detection and they can work in bad operating conditions.

- Diffuse or Reflective sensors: when they detect an object, the ultrasonic waves comes back to the sensor. Any sound reflecting, stationary object is used as a reflector. The ultrasonic is in not active state as long as the measured propagation time matches to the distance from the sensor to the reflector. The device is in active state when an object comes within the sensing distance and the propagation time changes. They have the transmitter and receiver box in the same housing [13].

For automotive application, ultrasonic sensors transmit sonic waves in the range of 40 kHz to 70 kHz, a range out of the audible one for humans which does not hurt human ears. This is important because, for example, parking sensors of the car can produce more than 100 dB of sound pressure [13]. The majority of ultrasonic sensors are based on ToF principle, already explained in the section dedicated to LiDar sensors.

A second hardware improvement for the team was to add an HC-SR04 Ultrasonic Sensor Module. It utilizes sonar to determine the distance to an object trough the following formula:

$$D = ((S) * time)/2$$

where D=Distance to an object, S=speed of sound in the air and time=time between the transmission and reception of the signal

This sensor reads from 2cm to 400cm (0.8inch to 157inch) with an accuracy of 0.3cm (0.1inches). It is composed by an ultrasonic transmitter, which emits a high-frequency sound (40 kHz), and receiver, which receives the reflected sound (echo), modules and it has the following sensor features [14]:

- Power Supply :+5V DC
- Quiescent Current : <2mA
- Working Current: 15mA
- Effectual Angle: <15°
- Ranging Distance : 2cm – 400 cm
- Resolution : 0.3 cm
- Measuring Angle: 30 degree
- Trigger Input Pulse width: 10uS TTL pulse
- Echo Output Signal: TTL pulse proportional to the distance range

In competition, the HC-SR04 Ultrasonic Sensor Module is mounted on the right side of the car, it detects the right-hand side obstacles like a vehicle and correctly performing the overtake and parking manoeuvre.

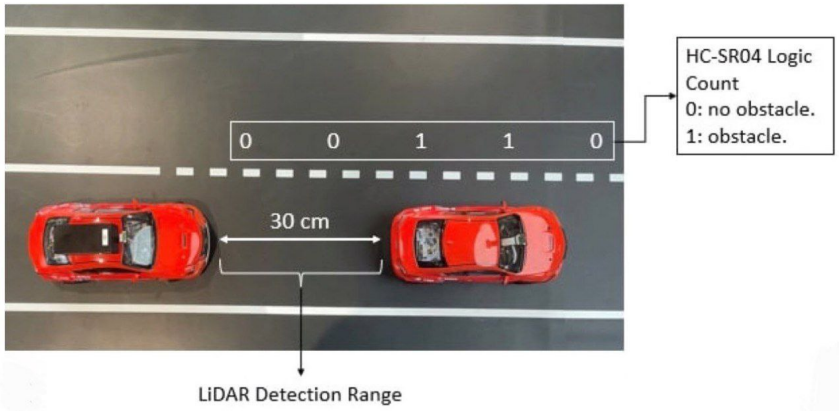

Figure 18: Overtake manoeuvre

The overtake manoeuvre is accomplished by both the LiDAR and the ultrasonic sensor. Manoeuvre triggered when the followings are satisfied at the same time:

- Dashed line.
- Obstacle (car) in front detected by the LiDAR.

Then, it changes the lane and perform the overtake checking the lateral obstacle presence, and finally ends the manoeuvre when the lateral sensor returns no obstacles anymore (Fig. 18). The parking manoeuvre will be analysed in the Parking chapter.

In the following code it is shown the overtake manoeuvre:

```
elif OVERTAKE and OVERTAKE_FLAG == True:
    cnt = cnt + 1
    if cnt == 1:
        print("Starting OVERTAKE manoeuvre")
        if HIGHWAY:
            self.Car_detected = 1
    if self.Ultrasonic == 1:
        # The car has been detected by the sensor, I'm
            overtaking
        car_detected = car_detected + 1
    elif self.Ultrasonic == 0 and car_detected > 7 and not
        END_OVERTAKE:
        # If I've detected the car and now it's gone, I can
            end the overtake manouvre
        END_OVERTAKE = True
        for x in range(6):
            if (vec_s[x] == 1):
                END_OVERTAKE = False
                break
    # Starting overtake:
    if cnt < 25:
        valore = 2001 # turning left
```

```
# Overtaking:
elif cnt >= 25 and cnt < 1000 and not END_OVERTAKE:
    valore = self.curveVal # overtaking, going straight on
elif cnt >= 25 and cnt < 1000 and END_OVERTAKE:
    cnt = 1000 # I "reset" the same counter with a much
        higher value to end the manouvre in order to use
        an unique counter
    valore = self.curveVal
# Ending overtake:
elif cnt >= 1000 and cnt < 1025:
    valore = 2002 # turning right
elif cnt >= 1025:
    valore = self.curveVal
    print("OVERTAKE manoeuvre finished")
    cnt = 0
    car_detected = 0
    OVERTAKE = False
    END_OVERTAKE = False
    NORMAL = True
else:
    print("Overtake error")
```

4.2.4 ATM103 Encoder

An encoder in digital electronics is a device that measure rotation. Connected to appropriate electronic circuits and with appropriate mechanical connections, the encoder is capable of measuring angular displacements, rectilinear and circular movements as well as rotational speeds and accelerations. There are various techniques for motion detection: angular capacitive, magnetic, inductive and photoelectric. It is often used for parking in order that car can understand when it is parallel to the street and finish the manoeuvre. Team Politron did not use it.

4.2.5 IMU Sensor

IMU sensor is a device to measure orientation, gravidational force and velocity. At the beginning, technology consisted of sensor accelerometers to evaluate the inertial acceleration and gyroscopes to measure angular rotation. In this case, IMU technology has six DOF because both sensors have three degrees of freedom. Each sensor can measure angles and in order to obtain more accurate output, both data can be calibrated.

Nowadays, IMU technology progresses with the magnetometer that evaluates the bearing magnetic direction improving the reading of gyroscope. In this case, IMU technology has 9 DOF because also magnetometer sensor has three degrees of freedom. It is used for dynamic orientation calculation in the short and long run when less drift errors occur but in the environment ferromagnetic metal can be present and the measurement could be altered by magnetic field disturbances [15].

Bosch provides to teams the smart IMU sensor BNO055 (Fig. 19). It is a System in Package (SiP) solution that integrates a triaxial 14-bit accelerometer, an accurate close-loop triaxial 16-bit gyroscope, a triaxial geomagnetic sensor and a 32-bit microcontroller running the BSX3.0 FusionLib software. It is really small [1].

Figure 19: IMU sensor BNO055

Team Politron used IMU sensor module to detect the orientation of the car and thus to correct its positioning inside the lane. When the code is run, the car must be positioned perfectly straight in the lane in order to set the angle 0 of the IMU in such a way that the sensor can detect the right angulation of the car respect to the street and it can correct its position. Beside this, it is used to detect the ramp due to the change in the inclination level.

Below an example of code used in competition:

```
    def IMU_initialization(self):
        self.SETTINGS_FILE = "RTIMULib"
        print("Using settings file " + self.SETTINGS_FILE + ".ini")
        if not os.path.exists(self.SETTINGS_FILE + ".ini"):
            print("Settings file does not exist, will be created")
        self.s = RTIMU.Settings(self.SETTINGS_FILE)
        self.imu = RTIMU.RTIMU(self.s)
        print("IMU Name: " + self.imu.IMUName())
        if (not self.imu.IMUInit()):
            print("IMU Init Failed")
            self.stop()
            sys.exit(1)
        else:
            print("IMU Init Succeeded")
        self.imu.setSlerpPower(0.02)
        self.imu.setGyroEnable(True)
        self.imu.setAccelEnable(True)
        self.imu.setCompassEnable(True)

        self.poll_interval = self.imu.IMUGetPollInterval()
        print("Recommended Poll Interval: %dmS\n" % self.poll_interval)

    def _IMUMeasure(self,):
        if self.imu.IMURead():
            self.data = self.imu.getIMUData()
            self.fusionPose = self.data["fusionPose"]
            self.accel = self.data["accel"]
            self.roll = math.degrees(self.fusionPose[0])
            self.pitch = math.degrees(self.fusionPose[1])
            self.yaw = math.degrees(self.fusionPose[2])
            self.accelx = self.accel[0]
```

```
            self.accely = self.accel[1]
            self.accelz = self.accel[2]

            print("roll = %f pitch = %f yaw = %f" % (self.roll, self.pitch,
                self.yaw))

    def _straight_correction(self, starting_yaw):
        # Retreive the orientation:
        if starting_yaw <= 45 or starting_yaw >= 315:
            yaw_ref = self.yaw_ref_N
        elif starting_yaw >= 135 and starting_yaw <= 225:
            yaw_ref = self.yaw_ref_S
        elif starting_yaw > 45 and starting_yaw < 135:
            yaw_ref = self.yaw_ref_E
        elif starting_yaw > 225 and starting_yaw < 315:
            yaw_ref = self.yaw_ref_O

        # Keep the straight manoeuvre close to the reference
        if yaw_ref == 0:
            if self.Yaw > 5:
                valore = 15
            elif self.Yaw < 355:
                valore = -15
            else:
                valore = 2000

        else:
            if self.Yaw < yaw_ref - 5:
                valore = 15 #turn right
            elif self.Yaw > yaw_ref + 5:
                valore = -15 #turn left
            else:
                valore = 2000 #go straight
        return valore
```

4.3 Localisation

In autonomous driving, the vehicle must be provided of sensors, actuators, computers and algorithms as localization, planning, control and perception are needed. Localisation system gives the geographic position of the vehicle using the Global Positioning System, dead reckoning and roadway maps [16]. For a correct function of the autonomous operation, planning, control and perception system need location from localization system. Localization system must consider also particular weather and driving conditions such as obscured road, fog, etc. The global positioning system (GPS) has multipath, low accuracy and signal blockage issues but it is cheap and for this reason it is often used to provide solutions for localisation. To have a robust localisation system, two solution can be considered:

1. Development of advanced sensors (LiDAR, camera or RADAR etc.).

2. Fusing sensors data with network infrastructure.

To access to various vehicle information such as traffic information, close cars or weather, the vehicle can connect through V2V (vehicle-to-vehicle) system that is

composed by wireless connectivity embedded to enhance robustness to tackle the line-of-sight problems and localization accuracy [17]. The vehicle-to-vehicle technique takes sensor information from neighbouring vehicles into consideration to estimate location forming a network (VANET) of vehicles where each car is informed about location and movement of neighbouring vehicles. This methods is very cheap respect to LiDar measurement based systems. More vehicles cooperatively calibrate their position and identify nearby vehicles through their GPS receivers and ranging sensors. The system works until surrounding vehicles do not or cannot participate in the system. Each car obtains some pieces of positioning information different degrees of accuracy and combine them to minimize estimation errors [18]. The accuracy of localization of V2V cooperative localization system is determined by the number of the shared position and vehicle connected in the surrounding area (Fig.20). Same reasoning can be done to receive information from infrastructure or sensor nearby the car with V2I and V2S techniques.

Figure 20: V2V real life representation

4.3.1 Localisation and mapping for BFMC2022

The vehicle used for the competition has an indoor localization system with the goal to detect and to transmit the relative position for each car model on the track. Localization system has three principal components:

- Server: it evaluates the position of the cars collecting data from camera client and it sends their coordination to the Robot clients.
- Robot: it receives the coordination of cars from server.
- Camera client: it sends the position of the car to the server.

And it has the following specific:

- The frequency of the given messages is 3-4 Hz.
- The error of the system is about $\sim$ 10 cm.

- The delay of the received messages is ~ 0.4 seconds.
- The last detected position is stored and served. If the position is older then 2 seconds it is discarded.

To navigate in the track, a digital map in XML format is provided. It contains two important information: nodes and connections (Fig.21). The distance between two nodes is approximately 40 cm, every node is positioned in the middle of the lane and it has three features: Id, X coordinate and Y coordinate.

Nodes table			**Connections table**		
id	X	Y	Id_1	Id_2	Dotted
1	3.6	2.4	1	2	TRUE
2	4.0	2.4	2	3	FALSE
3	4.4	2.4	3	4	FALSE
4	4.8	2.4			

Figure 21: Table of Node and Connections [1].

The relation between two nodes is described in the connection where the start node id, the end node id and the type of connection (straight or dotted road: TRUE or FALSE) data are given. In the intersection case, there will be three or four points with the same coordinates, each one for a different direction [1].

In the competition, the networkX library is used. It is a package for the creation, manipulation and study of the structure, dynamics, and functions of complex networks, including [19]:

- Data structures for graphs, digraphs, and multigraphs.
- Many standard graph algorithms.
- Network structure and analysis measures.
- Generators for classic graphs, random graphs, and synthetic networks.
- Nodes can be "anything" (e.g., text, images, XML records).
- Edges can hold arbitrary data (e.g., weights, time-series).
- Open source 3-clause BSD license.
- Well tested with over 90% code coverage.
- Additional benefits from Python include fast prototyping, easy to teach, and multi-platform.

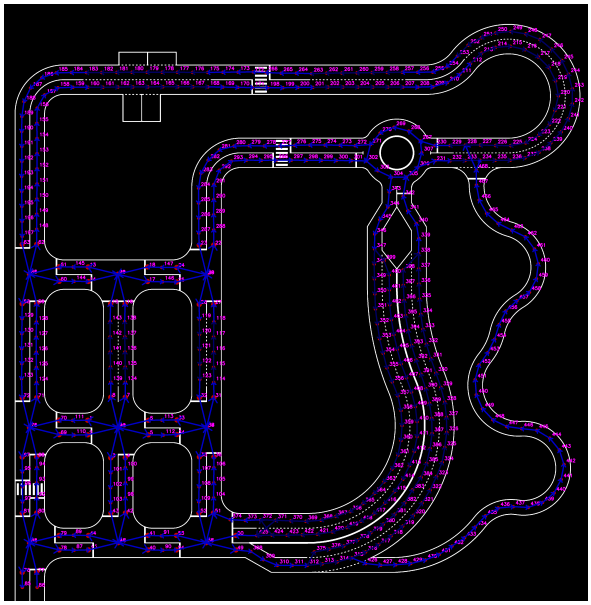

Figure 22: Competition track with nodes and connections [1].

To save and use the map (Fig. 22), it is useful to download it in the same folder where the localisation codes are present. The function *_saveGraph()* is created to accomplish the goal. Furthermore, to read and to process the attributes of the nodes it is needed an iteration through them, extracting the data from them.

```
    def _saveGraph(self):
        '''Function created to read .XML file, save node and connection
            information'''

        x = []
        y = []
        line = []
        source_node = []
        bool_val = 0

        # read graph
        G = nx.read_graphml('./src/data/Competition_track.graphml')
        pos = nx.circular_layout(G)

        for (node, node_pos) in pos.items():
            node_pos[0] = G.nodes[node]['x']
            x.append(node_pos[0])
            node_pos[1] = G.nodes[node]['y']
            y.append(node_pos[1])

        # print graph
        plt.clf()
        nx.draw_networkx(G, pos)
        plt.show()
        print('\n')

        # save edges
        for n, data in G.edges.items():
            if data['dotted']:
```

```
            bool_val = 1
            line.append(bool_val)
        else:
            bool_val = 0
            line.append(bool_val)

    # save nodes
    for node in G.nodes(data=True):
        source_node.append(node[0])

    return x, y, line, source_node
```

Moreover, there is a module that gets the position and orientation of the car itself on the map from the localization system in the following way. At first, on the broadcast port '12345', server streams its TCP port, client sends its own ID and it connects to the server on the communicated TCP port. Afterwards, server responses with an encrypted message with its private key that client translates with the server public key and it checks that the messages are equal. If they are, the connection between server and client begins and the server shares its position and orientation on the map in order to validate the position on the track at each moment. Bosch released a code where the vehicle uses application programming interface (APIs) for communications. API is a type of software interface that more computers use to communicate with each other. It validates and connects server with the car given ID where the server gives back the position of the car on the track. Localisation system scripts are created to intercept all data, they are created by Bosch engineers [20] and they are adapted to the requirement of the project:

- Locsys.py script: it is a thread that receives from the Trajectory process (that will be explained in the section dedicated at the trajectory planning) the id number, the server public key and beacon. GPS Tracker connects to the server and it receives coordinate of the car on the race. In 'setup state', it creates the connection with server. In 'listening state', it receives the messages.

```
def listen(self):
    """ Listening the coordination of car"""
    coord = self.__position_listener.listen() # listen messages
        from server and receive coordinates
    while self.__running:
        if self.__server_data.socket != None:
            try:
                self.q1.put(coord)
            except Exception as e:
                print('Errore in thread LOCSYS')
```

- Server_data.py script: it is the first function called in *locsys.py*. It needs server connection and it includes parameters of the server that are updated in *ServerListener* and *SubscribeToServer* classes.

- Server_listener.py script: its goal is to find the server. It stands until a broadcast message that contains a port where the server listens the car client arrives on predefined protocol. It ends the listening when the message is correct and a subscriber object tries to connect on server. It has a function *find()* that

creates a socket with predefined parameters, where it waits the broadcast messages (a port number where the server listen the car clients that it converts to an integer value).

```
def find(self):
  try:
     #: create a datagram socket for intramachine use
     s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
     #: used to associate socket with a specific network interface
         and port number
     s.setsockopt(socket.SOL_SOCKET, socket.SO_BROADCAST, 1)
     s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)

     s.bind(('', self.__server_data.beacon_port))

     #: Listen for server broadcast
     s.settimeout(1)

     while ((not self.__server_data.is_new_server) and
         self.__running):
        try:
           # waiting for the beacon.
           # Receive data from the socket. Buffer size = 1500 bytes
           data, server_ip = s.recvfrom(1500, 0)

           # convert the received message
           subscriptionPort = int(data.decode("utf-8"))

           # actualize the parameter of server_data with new IP
               address and communication port
           self.__server_data.serverip = server_ip[0]
           self.__server_data.carSubscriptionPort = subscriptionPort

           self.__server_data.is_new_server = True
```

- Server_subscriber.py script: it has the role to subscribe on the server, to create a connection with the parameter of *server_data.py* and to verify the server authentication. It sends the identification number of car after creating a connection and the server authorizes it through two messages. The authentication is based on the public key of server stored in 'publickey.pem' file. It has a function *subscribe()* that connects to the server and send the car id. After sending the car identification number it checks the server authentication.

```
def subscribe(self):
   try:
      # creating and initializing the socket
      sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
      sock.connect((self.__server_data.serverip,
          self.__server_data.carSubscriptionPort))
      sock.settimeout(2.0)

      # sending car id to the server
      msg = "{}".format(self.__carId).encode('utf-8')
```

```
            sock.sendall(msg)

            # receiving response from the server
            msg = sock.recv(4096)
            # receiving signature from the server
            signature = sock.recv(4096)

            # verifying server authentication
            is_signature_correct = verify_data(self.__public_key, msg,
                signature)
            # Validate server
            if msg == '' or signature == '' or not
                is_signature_correct:
                msg = "Authentication not ok".encode('utf-8')
                sock.sendall(msg)
                raise Exception("Authentication failed")

            msg = "Authentication ok".encode('utf-8')

            sock.sendall(msg)

            self.__server_data.socket = sock
            self.__server_data.is_new_server = False
```

- Position_listener.py script: it aims to receive all messages from the server. After the subscription on the server, the function *listen()* tunes the messages on the previously initialed socket, it decodes and stores it in 'coor' member parameter that is update by each new message. The server transmits acquired coordinate and timestamp of the detected car.

```
    def listen(self):

      while self.__running:
         if self.__server_data.socket != None:
            try:
               msg = self.__server_data.socket.recv(4096)
               msg = msg.decode('utf-8')

               if(msg == ''):
                  print('Invalid message. Connection can be
                      interrupted.')
                  break

               coor = json.loads((msg),cls=ComplexDecoder)
               self.__streamP_pipe.put(coor)

      self.__server_data.is_new_server = False
      self.__server_data.socket = None
      self.__server_data.serverip = None
```

4.3.2 V2X for BFMC2022

At the competition, teams shared the MAC of the car computer because all the cars must connect to the LAN of Bosch in order to have V2X communication.

- Semaphores

  Connecting to the LAN, cars receive each semaphore broadcast messages with a frequency of 10 Hz, including semaphore ID and its state:

0	1	2
RED	YELLOW	GREEN

Table 1: Semaphore State [1]

In Bosch location, traffic lights stream, using UDP protocol, their position on the network in the 5007 port and they also broadcast their state. The thread called *trafficlights.py* gets all the data and saves it as its attributes. As long as the thread runs, for each semaphore ID received from the socket the corresponding state (color) is associated. On the race track, four semaphore were present: one at the starting point and the other three placed in the intersections. Team Politron creates a process called *TrafficStateProcess* that uses the *trafficlights.py* thread, in fact it imports the file to receive the state of 4 semaphores on the track. It defines the position of each semaphore a priori known and the color lists as red, yellow and green. It receives the team car coordinate from localisation and if the car is near to a specific semaphore it sends its state. Below, *runListener* function in the *TrafficStateProcess* is shown:

```
        def runListener(self,inPs,outPs):

            try:
                # Semaphore colors list
                colors = ['red','yellow','green']

                # Create listener object
                Semaphores = trafficlights.trafficlights()

                # Start the listener
                Semaphores.start()

                while True:
                    # Receive car coordinates
                    coora = inPs[0].recv()

                    # Save car coordinates (KitKat is the car name)
                    x_KitKat = coora['coor'][0].real
                    y_KitKat = coora['coor'][0].imag

                    # Sempahore position on the track
                    x_semaforo1 = 3.05
                    y_semaforo1 = 11.41
                    x_semaforo2 = 2.1
                    y_semaforo2 = 10.83
                    x_semaforo3 = 0.82
                    y_semaforo3 = 14.29
                    x_semaforo4 = 3.63
                    y_semaforo4 = 10.4
```

```
# Sending the state for each semaphore

if abs(x_KitKat-x_semaforo1) <= 0.4 and
    abs(y_KitKat-y_semaforo1) <= 0.4:
    self.SemaphoreState = Semaphores.q1.get()
elif abs(x_KitKat-x_semaforo2) <= 0.4 and
    abs(y_KitKat-y_semaforo2) <= 0.4:
    self.SemaphoreState = Semaphores.q2.get()
elif abs(x_KitKat-x_semaforo3) <= 0.4 and
    abs(y_KitKat-y_semaforo3) <= 0.4:
    self.SemaphoreState = Semaphores.q3.get()
elif abs(x_KitKat-x_semaforo4) <= 0.4 and
    abs(y_KitKat-y_semaforo4) <= 0.4:
    self.SemaphoreState = Semaphores.q4.get()
else:
    self.SemaphoreState = 100

outPs[0].send(self.SemaphoreState)

# Stop the listener
Semaphores.stop()
```

- Environmental server

  At the location, cars sent to local environmental server at TCP port "23456" the position and the id of the encountered obstacle (as shown in table 4.3.2). In order to connect to the server, cars sent its own ID and the ID crypted with the car private key in its turn decrypted by server with corresponding public key. If the ID's corresponded, the client was validated by the server. Consequently, server replied with encrypted messages with own private key that client decrypted with the server public key and it checked if the message were the same. If it were, connection was initiated.

ID	Obstacle
1	TS - Stop
2	TS - Priority
3	TS - Parking
4	TS - Crosswalk
5	TS - Highway entrance
6	TS - Highway exit
7	TS - Round About
8	TS - One way road
9	Traffic light
10	Static car on road
11	Static car on parking
12	Pedestrian on crosswalk
13	Pedestrian on road
14	Roadblock
15	Bumpy road

Table 2: ID assignment for each obstacle [1]

As in localization, environmental module is composed by several scripts in order to reach its goal. Firstly, *environmental.py* script is a thread that connects on the server and sends messages, which incorporates the coordinate of the encountered obstacles on the racetrack. It creates the connection with server in the setup state, moreover in streaming state it sends the messages to the server. Then *serverdata.py* script is a class that contains parameters updates in *ServerListener* and *SubscribeToServer* classes. The former class goal is to find the server. It stands until a broadcast message that contains a port where the server listens the car client arrives on predefined protocol. It ends the listening when the message is correct and a subscriber object tries to connect on server. It has a function *find()* that creates a socket with predefined parameters, where it waits the broadcast messages (a port number where the server listen the car clients that it converts to an integer value). *SubscribeToServer* class has the role to subscribe on the server, to create a connection with the parameter of *server_data.py* and to verify the server authentication. It sends the identification number of car after creating a connection and the server authorizes it through two messages. The authentication is based on the public key of server stored in 'publickey.pem' file. It has a function *subscribe()* that connects to the server and send the car id. After sending the car identification number it checks the server authentication. Instead, *environmental_streamer.py* aims to send all message to the server.

The author created a process called *EnviromentalProcess* that receives car coordinates from localisation system and send object coordinate (_enviromental) to the server. It defines a vector_flag in order not to send several times coordinates and ID required by the competition, once is sufficient. It acquires sign flag when in *TrafficSignProcess* a sign is detected, it obtains object flag when *ObjectDetectionProcess* senses a specific object. Sign flag and object flag is equal to a specific number according to the sign or object that the car detects. Making the use of *environmental.py*, it connects to the server and send ID and coordinates of the obstacle encountered. Below, *_environmental* function of the *EnviromentalProcess* is shown:

```
        def _environmental(self, inPs,gpsStS_env):
            try:
                vector_flag=[0,0,0,0,0,0,0,0,0,0,0,0,0,0]
                while True:
                    try:
                        #Receive from localisation car coordinate
                        coora = inPs[0].recv()
                        # Save coordinate
                        x_gps = coora['coor'][0].real
                        y_gps = coora['coor'][0].imag
                        # Receives sign from image processing
                        sign = inPs[1].recv()
                        # Receives object from object detection
                        object_ = inPs[2].recv()
                        # Dynamic car in nearby
                        car_in_m = inPs[3].recv()

                        if sign == 1 and vector_flag[0] == 0:
                            #STOP
                            vector_flag[0] = vector_flag[0] +1
```

```
                a = {'obstacle_id': sign, "x": x_gps,
                    "y":y_gps}
                gpsStS_env.send(a)

            if sign == 111 and vector_flag[0] == 0:
                #STOP
                vector_flag[13] = vector_flag[13] +1
                a = {'obstacle_id': 1, "x": x_gps,
                    "y":y_gps}
                gpsStS_env.send(a)

            if sign == 2 and vector_flag[1] == 0:
                #PRIORIY
                vector_flag[1] = vector_flag[1] +1
                a = {'obstacle_id': sign, "x": x_gps,
                    "y":y_gps}
                gpsStS_env.send(a)

            if sign == 3 and vector_flag[2] == 0:
                #PARKING
                vector_flag[2] = vector_flag[2] +1
                a = {'obstacle_id': sign, "x": x_gps,
                    "y":y_gps}
                gpsStS_env.send(a)

            if sign == 4 and vector_flag[3] == 0:
                #CROSSWALK
                vector_flag[3] = vector_flag[3] +1
                a = {'obstacle_id': sign, "x": x_gps,
                    "y":y_gps}
                gpsStS_env.send(a)
```

- V2V

  The car of each team receives ID and position of dynamic obstacle through Wi-Fi UDP messages with 4 Hz of frequency and 10 cm radius of accuracy. The thread called *vehicletovehicle.py* gets all the data and saves it as its attributes. As long as the thread runs, it receives indoor positioning and orientation of the moving obstacle/car that are streaming their position into the network. The author wrote a process called *V2VProcess* that receives from server the coordinates of a car different from the car of the team and it receives team car coordinates from localisation system. If they are near to each other, it sends the other car coordinates to *MovCarProcess*. *MovCarProcess* is the process that sends commands to the Nucleo board. *V2VProcess* was not used in the competition because there was too much process in parallel and the car had problem to compute all the task at the same time. Below, *_runListenerV2V* function of the V2VProcess is shown:

```
    def _runListenerV2V(self,inPs,outPs):
        try:
            start_time = time.time()
            vehicle = vehicletovehicle.vehicletovehicle()
            # Start the listener
            vehicle.start()
```

```
while True:
    # Receives team car coordinate
    coora = inPs[0].recv()
    x_KitKat = coora['coor'][0].real
    y_KitKat = coora['coor'][0].imag
    #Receive the other car coordinates
    self.coorV2V = vehicle.pos
    x_V2V = self.coorV2V['coor'][0].real
    y_V2V = self.coorV2V['coor'][0].imag
    # Sending V2V position
    if math.sqrt(math.pow((x_V2V - x_KitKat),2)
        + math.pow((y_V2V - y_KitKat),2)) >= 0
        and math.sqrt(math.pow((x_V2V -
        x_KitKat),2) + math.pow((y_V2V -
        y_KitKat),2)) <= 1:

        outPs[0].send(self.coorV2V)

    else:
        NO=0
        outPs[0].send(NO)
```

5 Trajectory and path planning

5.1 Trajectory planning

An autonomous vehicle must move from a starting point to a final one. To do that, a predefined path according to the vehicle or path limits is needed. Cars have limited maneuvers due to limited steering angle or inaccessibility of certain places, such as smaller streets or streets with work in progress. The first step is to plan a trajectory that designs a predefined path, that starts from point A and ends in point B, according to a defined time law. It is important to underline the difference between path and trajectory: the former is the place of points that the vehicle has to follow to reach the end point (a purely geometric meaning), the latter is a path in which a timing law is specified [21]. Trajectory planning (Fig.23) consists in path planning and movement planning based on velocity, time and kinematics. A trajectory planner is a tool that computes a set of reference values, inputs of the control block, useful to bring the autonomous plant to the desired configuration given the desired kinematic, dynamic constraints and the path. Trajectory planner must react quickly to an environment change, it must plan a feasible motion for the autonomous vehicle, it has to control all kind of car behaviour [22].

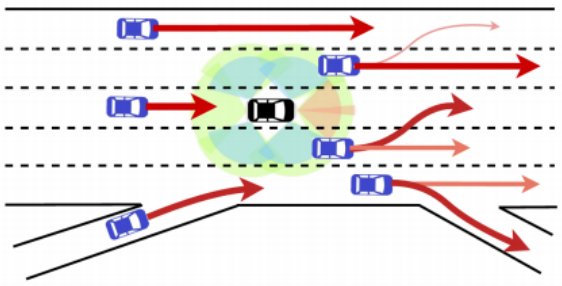

Figure 23: Example of trajectories in autonomous vehicle [2].

5.1.1 Trajectory planning used in competition

The process that deals with trajectory planning is called *RaceTrajectoryProcess*. Its principal function is called *_localisation()* and its first task is to saves the graph calling the function *_saveGraph()*. As explained in localisation section, the team car, in order to understand its position, saves the graph that represents the competition track. Furthemore, it receives the coordinates of the car from localisation system and the function *_understandPosition()* reports the exact position of the car in the track, according to the GPS uncertainties (at most 10 cm). Subsequently, the process calls the function *_behaviour()*, to instruct the car on which operation it has to execute. Finally, it stores the position of the traffic signs, of which the teams is already aware, and it sends the car position near to the stored traffic sign to *EnvironmentalProcess*.

- *_localisation* script is presented below:

```
def _localisation(self, inPs, q1, outPs):
    try:
        # localisation of the car in the track
        self.x, self.y, self.line, self.source_node =
            self._saveGraph()
        while True:
```

```
            stamps= inPs[0].recv()
            # car's coordinate
            coora = q1.get()
            x_gps = coora['coor'][0].real
            y_gps = coora['coor'][0].imag
            #car understands its position on the track
            self._understandPosition(x_gps, y_gps)
            #car chooses its behaviour
            self._behaviour()
            #Sign on the track
            self._thereIsASign()
            outPs[0].send(self.behaviour) # send behaviour to
                movecar
            outPs[1].send(self.sign) #send sign already saved on
                the map to MovCarProcess
            outPs[2].send(self.sign) #send sign already saved on
                the map to EnvironmentalProcess
            outPs[3].send(coora)
            outPs[4].send(coora)
```

- The *_understandPosition()* function receives the coordinates of the car and compares them with the coordinates of the nodes present in the graph. In order to determine the nearest node the calculation of the euclidean distance from each of them is evaluated:

$$distance = \sqrt{(x_n - x_c)^2 + (y_n - y_c)^2}$$

and the smallest one is selected. In the formula, x_c and y_c are the coordinates of the car instead x_n and y_n the coordinate of the node. Nodes has no a precise numerical order in the graph, thus team decided to iterate, each time, to all the node present in the graph to be sure to select the right one. In the competition, the car had no problem to understand its position and it was relatively fast to fulfill its goal.

```
    def _understandPosition(self, x_gps, y_gps):
        '''This function verifies that the car coordinates coming from
            server correspond to a position in the XML file'''
        cnt = 0
        dist_min = 10000
        ID_min = 0
        x_min=0
        y_min=0
        for cnt, val in enumerate(self.x):
            # the statement if is used to find the corresponding node
                of the car position with some uncertainty
            dist = math.sqrt((x_gps-self.x[cnt])**2 + (y_gps -
                self.y[cnt])**2)
            if dist < dist_min:
                dist_min = dist
                x_min = self.x[cnt]
                y_min= self.y[cnt]
                ID_min = self.source_node[cnt]

        self.current_node = ID_min
```

```
        self.current_line = self.line[cnt]
```

- The *_behaviour()* function is the focal function for the trajectory planning. Knowing the map and the path (Fig.24) selected for the competition, a behavior that the car has to execute is set at each node; four vectors formed by nodes that help to understand the behaviour are established.

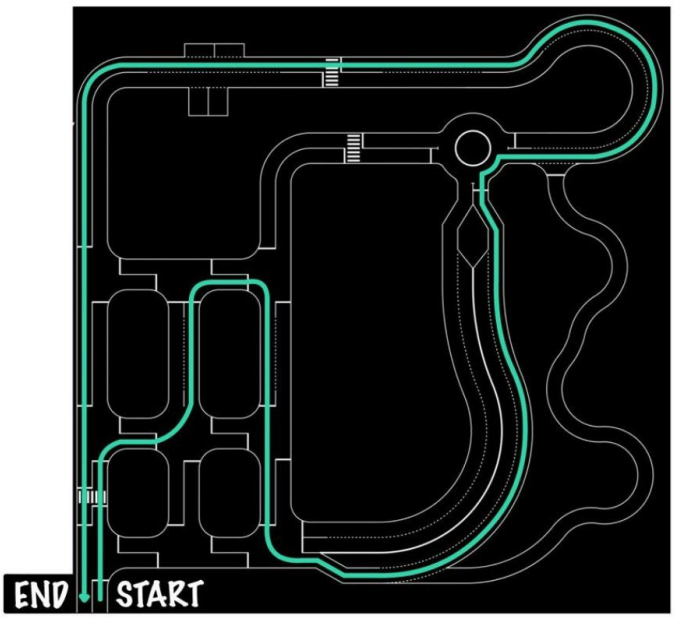

Figure 24: Competition track with the designed path.

The team selected a path to follow in order that the car accomplish the required tasks. If the car is in correspondence of a certain node, the script sends the behaviour to *MovCarProcess*. For instance, if the car is in the node number '68', it turns right because this node is in the vector that establishes that the car must turn right.

```
    def _behaviour(self):
        ''' According to the position it send 'go straight', 'right',
            'left' or 'normal' state. From 1-30 the car is in the
            intersection, otherwise no start at node 39, end at 111'''

        self.current_node = int(self.current_node)
        list_gostraight=[77,32,63,72,81]
        list_right=[68,16,25]
        list_left=[2,62]
        list_final = [131,132,133,126,125,124]

        self.behaviour=0 #normal state: lane keeping
        for i in list_gostraight:
            if self.current_node==i:
                # go straight
                self.behaviour = 2
        for j in list_right:
```

```
            if self.current_node ==j:
                # turn right
                self.behaviour = 4
        for z in list_left:
            if self.current_node == z:
                # turn left
                self.behaviour = 3
        for l in list_final:
            if self.current_node == l:
                # end
                self.behaviour = 5
        if self.behaviour == 0 and ((313 < self.current_node and
            self.current_node < 336) or (375 < self.current_node and
            self.current_node < 398) or (256 < self.current_node and
            self.current_node < 263) or (201 < self.current_node and
            self.current_node < 208)):
            # NORMAL_DASHED
            self.behaviour = 1
```

- The *_thereIsASign()* function saves the already known position of the signs, then this information will be sent to *MovCarProcess* and to *Environmantal-Process*.

```
    def _thereIsASign(self):
        self.sign= -1

        node_STOP = [90,91,54,53, 94, 95, 68, 67]
        node_PRIORITY = [196,197,63,62,148,149]
        node_PARKING = [175,176,167,168]
        node_CROSSWALK = [80,92,81,97]
        node_HIGHWAY_ENTRANCE = [49,308,309,50,425]
        node_HIGHWAY_EXIT = [338,339,340,345,346]
        node_ROUNDABOUT = [341,342,343,344]
        node_ONE_WAY_ROAD = [7,8]

        for i in node_STOP:
            if self.current_node == i:
                self.sign = 1
        for j in node_PRIORITY:
            if self.current_node == j:
                self.sign = 2
        for k in node_PARKING:
            if self.current_node == k:
                self.sign = 3
        for l in node_CROSSWALK:
            if self.current_node == l:
                self.sign = 4
        for m in node_HIGHWAY_ENTRANCE:
            if self.current_node == m:
                self.sign = 5
        for n in node_HIGHWAY_EXIT:
            if self.current_node == n:
                self.sign = 6
        for o in node_ROUNDABOUT:
            if self.current_node == o:
```

```
            self.sign = 7
    for p in node_ONE_WAY_ROAD:
        if self.current_node == p:
            self.sign = 8
```

When 'go straight', 'right', 'left' or 'normal' state is sent to *MovCarProcess*, the car must follow specific commands selected in the process:

- Go straight: the car must go straight at the intersection using IMU sensor that can correct car position.
- Right: the car has to turn right at the intersection. Knowing the size and the angulation of the curve, the steering angle is set and the car turns right until the IMU sensor detects 90° respect to the position in which the manoeuvre starts (Fig.25).
- Left: the car must turn left at the intersection. Knowing the size and the angulation of the curve, the steering angle is set and the car turns left until the IMU sensor detects 270° respect to the position in which the manoeuvre starts (Fig.25).
- Normal: the car keeps the lane.

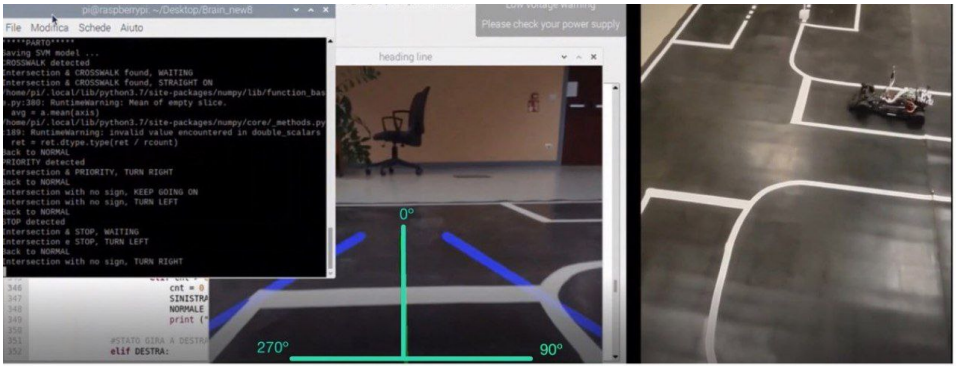

Figure 25: Car of the team at intersection with grades of IMU sensor specified.

During the competition two challenges were afforded: a technical one has to be executed in 10 minutes, instead the speed challenge in 3 minutes. For the former, the car had a velocity of 0.12 m/s if the steering angle was less than 25° otherwise 0.10 m/s. For the latter, the car had a velocity of 0.20 m/s if the steering angle was less than 45° otherwise 0.10 m/s. The decision of the team was based on the ability of the car to recognize the lane and act consequently; increasing speed, the car had difficulties to act enough rapidly.

5.2 Path planning

Path planning is a fundamental matter for autonomous vehicle because it has a huge effect on driver and passengers safety, it is the core of the autonomous vehicle abilities and obstacle avoidance [23]. The main goal of path planning is to find the shortest feasible path preventing uncertainties, taking into account vehicle dynamics, obstacle maneuvering capabilities, and traffic rules (Fig. 26); it is often based on a digital map and it selects the route that the car must take according to the surrounding environment. A proper planning leads to an increase in system efficiency and a reduction in power consumption [24]. Path planning can be global and local: the former gives the constraint to the local path planning so that it can achieve an optimal path based on a given requirements, for example a path to reduce pollution, to decrease the amount of time and less dangerous manoeuvres. If the planned path cannot be accomplished, the global planning must relaunch to get a new feasible path. Local path planning uses on-board sensor to understand the surroundings information in order to locate the position of the vehicle and obstacles on the map: in this way it can smoothly plan the best path from the starting node to target one [25].

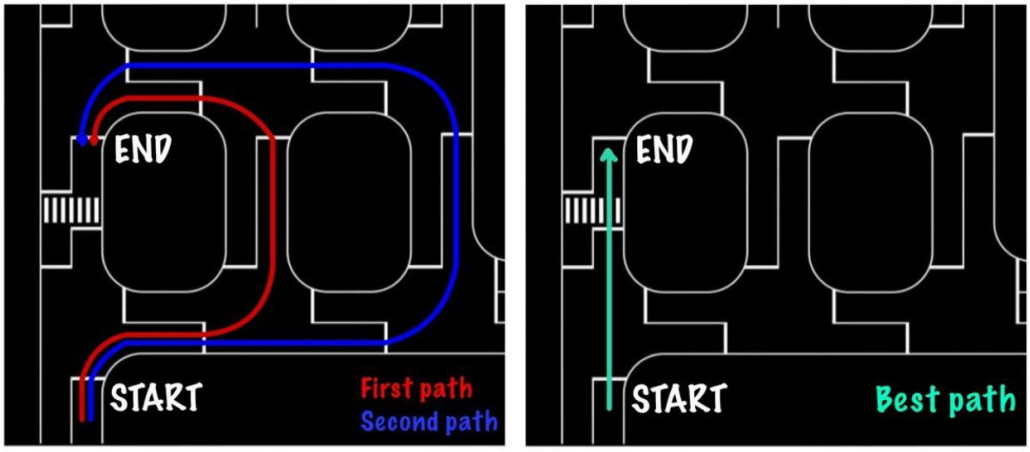

Figure 26: Example of different paths on the same map.

Path Planning consists of three main steps:

1. It generates a sequence of manoeuvres from the initial position to the final one.

2. It chooses the direction to follow at each movement.

3. It has to forward planning.

There are multiple tests in planning a path for an autonomous vehicle such as creating an offline map that represents the real world in order to help the system to understand the position of the vehicle; planning rapidly a path through points of the map, taking into account obstacles position; finding the appropriate acceleration and direction in order to get the more comfortable and safer path [23].

Path planning can be classified in the following way [26]:

- Space configuration: the surrounded environmental of the vehicle is divided in cells and, if they are collision-free, a solution is applied. The goal is to discover the right configuration of linked cells avoiding collision. These algorithms are fast but they often give unfeasible solutions.

- Path-finders: these algorithms are based on the search of a path among nodes in a graph. The main purpose is to find the optimal path in terms of cost function. Cost function must take in consideration the possibility of collisions and it has to take into account the comfort of a path. The main path-finders algorithms are: A*, Dijkstra used when the environment is previously known and Rapidly Random Tree (RRT) employed in unknown environments.

- Attractive and repulsive forces: they creates artificial forces that direct the car to the desired destination avoiding obstacles or specific areas. The sum of the forces is the state of the motion of the vehicle. The solution of this algorithm can be unstable or the algorithm can block into local minima.

- Curves: in this algorithm, a set of parametric or/and semi-parametric curves are generated according to the state of the car, the road and specific mathematical form. It can be applied through two main methods: point-free scheme and point-to-point scheme. In the former, the vehicle is able to follow a feasible kinematic/dynamic trajectory with a given legal maneuver thanks to the generated curves, otherwise in the latter, curves are used to arrange the trajectory between two points. Curves represent a local path and it is difficult to generate them in a correct way.

- Artificial intelligence schemes: they use human reasoning to solve problems and there are many techniques to find an optimal path. Accurate mathematical models are not needed.

In this project, path-finders algorithms are studied in deep. They can be divided in more categories [3]:

- Sampling-Based Algorithms (Fig. 27): they are divided into active and passive algorithms. The former have a processing procedure to produce the best possible path. LaValle [3] proposed the Rapidly exploring Random Tree (RRT) method that is an example of active sampling-based: it is able to solve multi-DOF problems and it handles path planning of different kinematic constraints; RRT algorithms are based on rapidly search of the configuration space to create the path that connects the initial node to the final node. The passive algorithms are the combination of search algorithms to collect the best feasible path among all suitable paths existing in the net map. They make use of the Probabilistic Road Maps (PRM) approach which is a technique that connects some sampled points in the map to create a graph composed by feasible routes with collision-free edges. PRM uses graph search algorithm to select the optimal path.

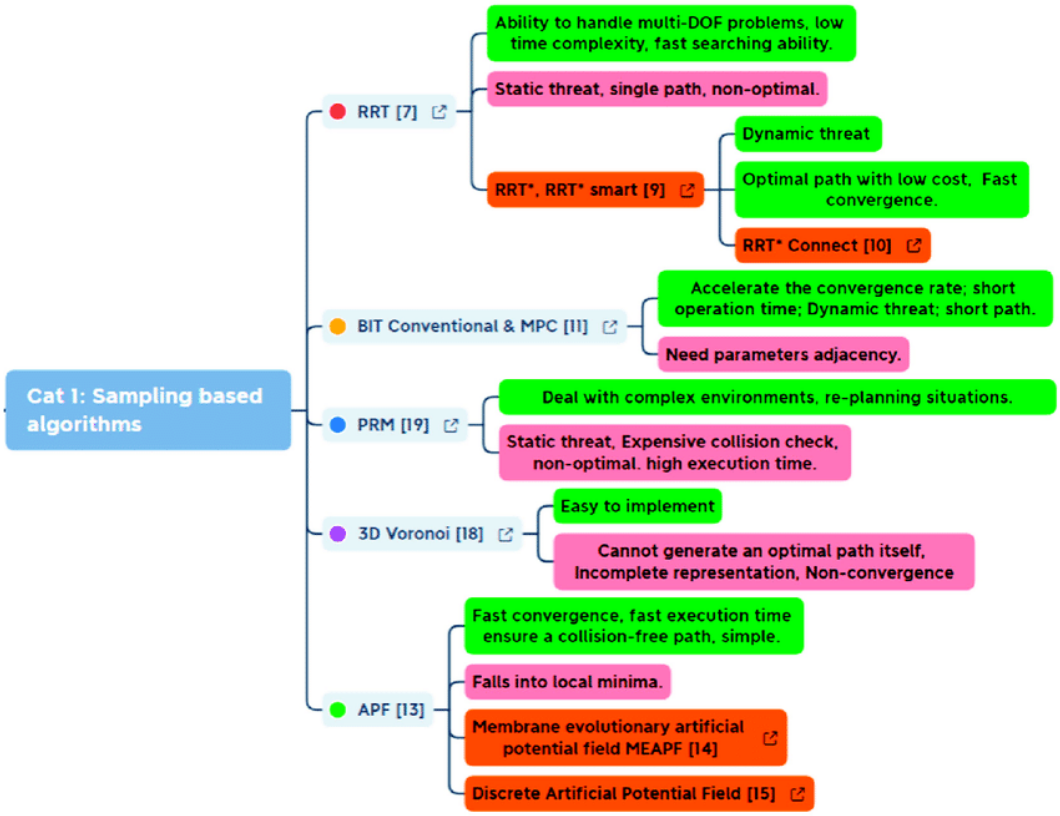

Figure 27: Sampling based approaches with the main advantages and drawbacks [3]

- Node-Based Optimal Algorithms (Fig. 28): they find an optimal path when information sensing and processing procedures are already executed and they are based on a principle of exploring among a set of cells in the map. Node-Based Optimal algorithms are divided into two categories: grid search algorithms and graph search algorithms. The former consists of mapping the environment that is divided into a set of cells in which a cell represents the presence of an object at a specific position; if it gives evidence form information, like the mass or the size value, it is called evidential occupancy grid. The graph search algorithms is useful to complete the first type. Dijkstra algorithm is a dynamic programming while A* is the heuristic search algorithms; they are both improved by searching with the least cost.

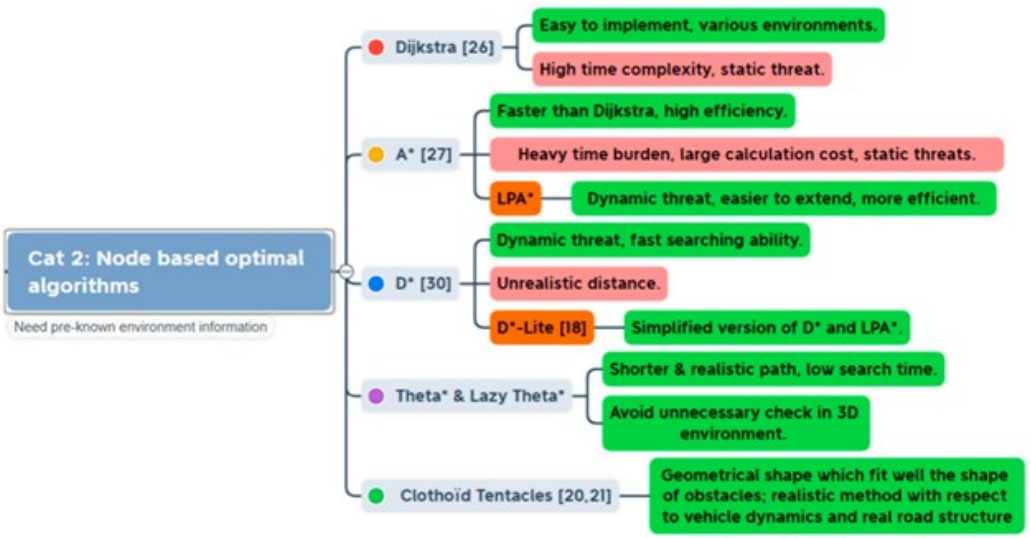

Figure 28: Node-based optimal approaches with the main advantages and drawbacks [3]

In particular, Dijkstra algorithm and A* are implemented in the following subsections.

5.2.1 Dijkstra Algorithm

In 1959, Edsger Dijkstra proposed the Dijkstra algorithm. It is a shortest path algorithm with a simple implementation and it easily adapts to the topology change. Dijkstra algorithm uses directed graph to search the shortest path from the source node to a destination one in a map with a unique source. At first, it processes the shortest path from source to adjacent nodes, the latter is called intermediate node. Next, it repeats the action between intermediate node to its adjacent nodes. It finishes the iteration when every node is traversed. It finds the shortest path from source node to any destination node in the path because it is composed by subpath that are the shortest paths from the source node and the terminal node. The Dijkstra algorithm can be considered as a kind of greedy algorithm and considering the assumptions [27]:

- *G=(V,E)* is a direct graph where V=set of nodes and E=set of arcs.
- n nodes of *G=(V,E)*.
- e arcs of *G=(V,E)*.
- *Dist(X)* is the distance between the source node v to X node.
- W is the weight of the arc.
- S is the set of nodes inside a shortest path.
- V - S denotes the set of nodes that is not inside in any shortest path.

The algorithm can be described as follows (Fig.29):

1. It selects the source node v and set S = S $\cup$ v.

2. In U = V - S, it finds the node i that is adjacent to the source node v and has smallest weight W of the arc that connects the two nodes. The node i is added into S.

3. Using i as the new intermediate node, it repeats the step *2* to find the adjacent node j. If j can be reached in different way, for instance, from source node to node j directly or passing through other nodes, the minimum distance is selected and stored as Dist(j).

4. Step *2* and step *3* are repeated n-1 times in order to obtain the shortest path.

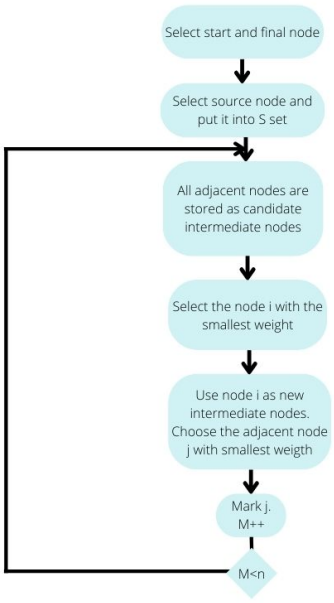

Figure 29: Flowchart of Dijkstra algorithm

The author created a process called *PathPlanning* to implement the Dijkstra algorithm. It is composed by the main function *_pathplanning()* that calls the function *_saveGraph()* to save the graph of the map and it finds the shortest path. Next, it receives the position of the car from localisation system and it calls the function *_understandPosition()* to understand the position of the car in the track. At the end, it uses the function *_behaviourPath()* to comprehend the right behaviour of the car and it sends it to *MovCarProcess*.

- *_pathplanning()* script is presented below:

```
def _pathplanning(self, inPs, outPs):
    try:
        self.x, self.y, self.line, self.source_node =
            self._saveGraph()
        while True:
            coora = q1.get() # car's coordinate
            x_gps = coora['coor'][0].real
            y_gps = coora['coor'][0].imag

            self._understandPosition(x_gps, y_gps)
```

```
                self._behaviourPath()
                outPs[0].send(self.behaviour) # send behaviour to
                    Movecar
            except Exception as e:
                print('Error in PathPlanning process')
```

- The *_saveGraph()* function is similar to the one explained in the section dedicated to the localisation and mapping for BFMC2022. It aims to save the graph of the map. In order to read and to process the attributes of the nodes it is needed an iteration and an extraction of the data from them. In this process, it uses the NetworkX library, particularly *networkx.dijkstra_path* function that uses Dijkstra's method to compute the shortest weighted path between two nodes in a graph. The NetworkX is a Python package for the creation, manipulation and study of the structure, dynamics, and functions of complex networks [19].

```
    def _saveGraph(self):
        '''Function created to read .XML file, save node and
            connection information'''

        # read graph
        G = nx.read_graphml('./src/data/Competition_track.graphml')
        pos = nx.circular_layout(G)

        for (node, node_pos) in pos.items():
            node_pos[0] = G.nodes[node]['x']
            x.append(node_pos[0])
            node_pos[1] = G.nodes[node]['y']
            y.append(node_pos[1])

        # print graph
        plt.clf()
        nx.draw_networkx(G, pos)
        plt.show()
        print('\n')

        # save edges
        for n, data in G.edges.items():
            if data['dotted']:
                bool_val = 1
                line.append(bool_val)
            else:
                bool_val = 0
                line.append(bool_val)

        # save nodes
        for node in G.nodes(data=True):
            source_node.append(node[0])

        # find the shortest path
        self.node_distance = nx.dijkstra_path_length(G, '112',
            '18')
        self.Path = nx.dijkstra_path(G, '112', '18')
```

```
        return x, y, line, source_node
```

- The *_understandPosition()* function, as in trajectory section, receives the coordinates of the car and compares them with the coordinates of the nodes present in the graph. To determine the nearest node the calculation of the euclidean distance from each of them is evaluated and the smallest one is selected. Nodes has no a precise numerical order in the graph, thus team decided to iterate, each time, to all the node present in the graph to be sure to select the right one. In the competition, the car had no problem to understand its position and it was relatively fast to fulfill its goal.

- In the *_behaviourPath()* function vectors with source node and target nodes that describe the possible behaviour are stored. Knowing the map, it is possible to know the feasible manoeuvres from a specific node. Being aware of the path, if the car is in the position of an "input" node, for example "RIGHT_INPUT" node, and the destination node after the intersection is on the right and it is stored in "RIGHT_OUTPUT", the behaviour of the car is to turn right and it is sent to *MovCarProcess*. The *for* statement is used to iterate over the path vector in order to understand which behaviour the car must follow.

```
    def _behaviourPath(self):
        RIGHT_INPUT = [77, 79, 43, 41, 52, 374, 50, 68, 70, 2, 4,
            6, 34, 32, 59, 61, 15, 16, 25, 23]
        RIGHT_OUTPUT = [78, 80, 44, 42, 53, 51, 51, 69, 71, 3, 5,
            7, 35, 33, 60, 62, 17, 17, 26, 25]
        LEFT_INPUT = [79, 81, 45, 43, 54, 52, 70, 72, 2, 4, 6, 34,
            36, 61, 63, 25, 27]
        LEFT_OUTPUT = [76, 78, 42, 40, 51, 49, 67, 69, 7, 1, 3,
            31, 33, 58, 60, 22, 24]
        STRAIGHT_INPUT = [77, 81, 45, 41, 54, 50, 68, 72, 2, 6, 4,
            36, 32, 59, 63, 14, 18, 27, 23]
        STRAIGHT_OUTPUT = [80, 76, 40, 44, 49, 53, 71, 67, 5, 1,
            7, 31, 35, 62, 54, 17, 13, 22, 26]

        self.current_node = int(self.current_node)

        for D in self.node_distance:
            if D < (self.node_distance-2):
                if self.current_node==self.Path[D]:
                    for i in RIGHT_INPUT:
                        if self.Path[D]==RIGHT_INPUT(i) and
                            self.Path[D+2]==RIGHT_OUTPUT(i):
                            #turn right
                            self.behaviour = 4
                        elif self.Path[D] == LEFT_INPUT(i) and
                            self.Path[D + 2] == LEFT_OUTPUT(i):
                            #turn left
                            self.behaviour = 3
                        elif self.Path[D] == STRAIGHT_INPUT(i) and
                            self.Path[D + 2] == STRAIGHT_OUTPUT(i):
                            #go straight
                            if ((313 < self.current_node and
                                self.current_node < 336) or (375 <
```

```
                        self.current_node and
                        self.current_node < 398) or (256 <
                        self.current_node and
                        self.current_node < 263) or (201 <
                        self.current_node and
                        self.current_node < 208)):
                        #dotted lane
                        self.behaviour = 1
                    else:
                        #continuous lane
                        self.behaviour = 2
            else:
                #lane keeping
                self.behaviour = 2

        elif D==self.node_distance-1:
            continue
        elif D==self.node_distance:
            if self.current_node == self.Path[D]:
                #stop
                self.behaviour = 5
            else:
                #lane keeping
                self.behaviour = 2
```

5.2.2 A* Algorithm

A* algorithm is an extension of Dijkstra algorithm and, in static network, it is the most efficient approach finding the briefest path. A* algorithm has numerous benefits, for example it has small search space or fast convergence. A* algorithm is based on graph search method where a grid map represents the environment. Traditional A* algorithm finds a path that avoids collisions respecting car limits and constraints when the car structure and the position of obstacles are known. The path in autonomous vehicle must be comfortable, real-time and reliable, which means the path planning must consider a trade-off between constraints in real time and model completeness [28]. A* algorithm is a search problem therefore it is important to set the start node, the end node and all the state that the car can bump into; another significant actions are to check the code, take into consideration all possible manoeuvres and set movement costs (edges in the graph). It uses a function that indicates the directions to follow and it finishes when the end point is reached. According to the function, if a step is reasonable the algorithm performs it. Heuristic methods are the base of the A* algorithm that finds the best possible solution. A* is a kind of algorithm that is guaranteed to find a solution if the solution exist thanks to the property of completeness. At each node, A* evaluates the cost *f(n)*, where n being the neighboring node, to move to all adjacent nodes and it travels to the one with the smallest value of *f(n)*. The *f(n)* can be calculated in the following way:

$$f(n) = g(n) + h(n) \tag{2}$$

where *g(n)* is the value of the briefest path from the initial node to node n and *h(n)* is the heuristic approximation of the node value. Each node is marked with its relative optimal *f(n)* value in order to reconstruct any path. To find the optimal solution, it is strictly needed a correct heuristic value *h(n)*; this determines the efficiency of A*.

Heuristic function have two properties [29]:

- Admissibility: the property to not overestimate the real distance between n and the final node in a given heuristic function *h(n)*. For each node, the following formula is employed:

$$h(n) \leq h^*(n) \tag{3}$$

  where $h^*(n)$ is the real distance between n and the target node. Nevertheless, if the heuristic function overvalues the real distance by a value smaller than d, it can be defined as a solution with accuracy equal to d.

- Consistency: the property to evaluate an estimate smaller or equal to the predicted distance between the final n and any given adjoint added to the estimated cost to reach that specific neighbour:

$$c(n, m) + h(m) \geq h(n) \tag{4}$$

  where *c(n,m)* is the distance between nodes n and m. The path for every node is optimal if *h(n)* is consistent and this denotes that the function is optimal.

The author created a process called *PathPlanningAStar* to implement A* algorithm. It is composed by the main function *_pathplanning()* that calls the function *_saveGraph()* to save the nodes of the graph of the map and it saves the edges in different dictionaries. Dictionary in Python is similar to a map and it consists of key-value pairs; dictionaries are joined in *graph_create()* function and the complete graph is stored in a fixed variable in *Graph()* function. After that, the function *a_star_algorithm()* computes the A* algorithm. Next, it receives the position of the car from localisation system and calls the function *_understandPosition()* to acknowledge the position of the car in the track. At the end, it uses the function *_behaviourPath()* to figure out the right behaviour of the car and sends it to *MovCarProcess*.

- *_pathplanning()* script is presented below:

```
def _pathplanning(self, inPs, outPs):
    try:
        self.x, self.y = self._saveGraph()

        #join the lists
        adjacency_list_u = self.graph_create(self.adjacency_list1,
            self.adjacency_list_supp1, self.adjacency_list_supp2,
            self.adjacency_list_supp3, self.adjacency_list_supp4,
            self.adjacency_list_85)

        #create the graph
        adjacency_list_u=self.Graph(adjacency_list_u)
        print(adjacency_list_u)
        #a* algorithm
        self.Path = self.a_star_algorithm('77','127')
        while True:
            coora = q1.get() # car's coordinate
            x_gps = coora['coor'][0].real
            y_gps = coora['coor'][0].imag
            self._understandPosition(x_gps, y_gps)
```

```
            self._behaviourPath()
            outPs[0].send(self.behaviour) # send behaviour to
                MovCarProcess
```

- The *_saveGraph()* function reads the known graph and saves all the nodes. Edges are saved in different dictionaries in order to avoid overwriting problems: for example, if node '2' has two destination nodes as '5' and '7', the code subscribes the first with the second destination node, therefore only node '2' with destination '7' is saved.

```
def _saveGraph(self):
# read graph
G = nx.read_graphml('./Competition_track.graphml')
pos = nx.circular_layout(G)
for (node, node_pos) in pos.items():
    node_pos[0] = G.nodes[node]['x']
    x.append(node_pos[0])
    node_pos[1] = G.nodes[node]['y']
    y.append(node_pos[1])
plt.clf()
nx.draw_networkx(G, pos)
plt.show()
print('\n')
j = 0
for n, data in G.edges.items():
    j = n[0]
    if j == '9':
        self.adjacency_list_supp1[j] = [('3', '1')]
        self.adjacency_list_supp2[j] = [('5', '1')]
        self.adjacency_list_supp3[j] = [('7', '1')]
        self.adjacency_list_supp4[j] = [('8', '1')]
    elif j == '10':
        self.adjacency_list_supp1[j] = [('7', '1')]
        self.adjacency_list_supp2[j] = [('5', '1')]
        self.adjacency_list_supp3[j] = [('8', '1')]
        self.adjacency_list_supp4[j] = [('1', '1')]
    elif j == '11':
        self.adjacency_list_supp1[j] = [('7', '1')]
        self.adjacency_list_supp2[j] = [('8', '1')]
        self.adjacency_list_supp3[j] = [('1', '1')]
        self.adjacency_list_supp4[j] = [('3', '1')]
    elif j == '12':
        self.adjacency_list_supp1[j] = [('1', '1')]
        self.adjacency_list_supp2[j] = [('3', '1')]
        self.adjacency_list_supp3[j] = [('5', '1')]
    #...
    elif j == '304':
        self.adjacency_list_supp1[j] = [('305', '1')]
        self.adjacency_list_supp2[j] = [('343', '1')]
    elif j == '306':
        self.adjacency_list_supp1[j] = [('231', '1')]
        self.adjacency_list_supp2[j] = [('307', '1')]
    elif j == '468':
        self.adjacency_list_supp1[j] = [('243', '1')]
```

```
                self.adjacency_list_supp2[j] = [('468', '1')]
            elif j == '315':
                self.adjacency_list_supp1[j] = [('316', '1')]
                self.adjacency_list_supp2[j] = [('426', '1')]
            else:
                self.adjacency_list1[j] = [(n[1], '1')]
        self.adjacency_list_85['85'] = [(None, '1')]

        return x, y
```

- The *graph_create()* function joins the dictionaries created in *_saveGraph()* using the 'ChainMap' container, member of module 'collection', which is composed by different containers. A container is an object that is employed to store and provides the access to distinct objects. The 'ChainMap' container encapsulates several dictionaries into one.

```
    def graph_create(self):
        adjacency_list_t = ChainMap(self.adjacency_list1,
            self.adjacency_list_supp1, self.adjacency_list_supp2,
            self.adjacency_list_supp3, self.adjacency_list_supp4)
        adjacency_list = {**adjacency_list_t, **self.adjacency_list_85}
        return adjacency_list
```

- The *Graph()* function sets the dictionary fixed in the class.

- The *a_star_algorithm()* function is presented below. Two lists are set in order to understand if the neighbors have been inspected or not: the list *g* contains all distances among nodes and the list *parents* contains an adjacency map of all nodes. A *while* statement is used to find the lowest value of *f(n)* and, to find the path, it takes into consideration the first edges of each node; if it does not find a path, it tries again with different edges. The *if* statement is used to understand if the path is ended. In each node, *open_list* and *closed_list* are updated.

```
    def a_star_algorithm(self, start_node, stop_node):
        # open_list is a list of nodes which have been visited, but
            who's neighbors
        # haven't all been inspected, starts off with the start node
        # closed_list is a list of nodes which have been visited
        # and who's neighbors have been inspected

        open_list = set([start_node])
        closed_list = set([])

        # g contains current distances from start_node to all other
            nodes
        # the default value (if it's not found in the map) is +infinity
        g = {}

        g[start_node] = 0

        # parents contains an adjacency map of all nodes
        parents = {}
```

```
parents[start_node] = start_node

while len(open_list) > 0:
    n = None

    # find a node with the lowest value of f() - evaluation
        function

    for v in open_list:
        if n == None or g[v] + self.h(v) < g[n] + self.h(n):
            n = v

    if n == None:
        print('Path does not exist! I will try again')

        adjacency_list =
            self.self.graph_create(adjacency_list1,
            adjacency_list_supp2, adjacency_list_supp1,
            adjacency_list_supp3, adjacency_list_supp4,
            adjacency_list_85)

        path = self.Graph(adjacency_list)
        self.Path = self.a_star_algorithm(start_node, stop_node)
        return

    # if the current node is the stop_node
    # then we begin reconstructin the path from it to the
        start_node
    if n == stop_node:
        reconst_path = []
        self.Distanza_nodi=0

        while parents[n] != n:
            reconst_path.append(n)
            self.Distanza_nodi=self.Distanza_nodi+1
            n = parents[n]

        reconst_path.append(start_node)

        reconst_path.reverse()
        self.Path=reconst_path
        print('Path found: {}'.format(reconst_path))
        return reconst_path

    # for all neighbors of the current node do
    for (m, weight) in self.get_neighbors(n):
        # if the current node isn't in both open_list and
            closed_list
        # add it to open_list and note n as it's parent
        if m not in open_list and m not in closed_list:
            open_list.add(m)
            parents[m] = n
            g[m] = g[n] + int(weight)

        # otherwise, check if it's quicker to first visit n,
            then m
```

```
                # and if it is, update parent data and g data
                # and if the node was in the closed_list, move it to
                    open_list
                else:
                    if g[m] > g[n] + weight:
                        g[m] = g[n] + weight
                        parents[m] = n

                        if m in closed_list:
                            closed_list.remove(m)
                            open_list.add(m)

            # remove n from the open_list, and add it to closed_list
            # because all of his neighbors were inspected
            open_list.remove(n)
            closed_list.add(n)

        print('Path does not exist!')
        return None
```

- In the *_behaviourPath()* function, as in Dijkstra algorithm, vectors with source node and target nodes that describe the possible behaviour are stored. Being aware of the map, it is possible to know the feasible manoeuvres from a specific node. Taking into account the path, if the car is in the position of an "input" node, for example "RIGHT_INPUT" node, and the destination node after the intersection is on the right and it is stored in "RIGHT_OUTPUT", the behaviour of the car is to turn right and it is sent to *MovCarProcess*. The *for* statement is used to iterate over the path vector in order to understand which behaviour the car must follow.

6 Parking

One of the most difficult tasks, in particular for amateur drivers, is parking (Fig.30). Finding a parking spot, enough large for the drivers' car, is one of the most common issues in cities or popular places. Once a suitable parking spot is found, the second problem is to start a manoeuvre avoiding collisions and trying to be fast in order to not disturb the surrounding traffic. Autonomous Parking system helps drivers to shirk these problems and autonomous vehicle to fulfill the manoeuvre. Nowadays, parking systems use cameras and sensors to analyze the surrounding environment and to compute the manoeuvre. Automatic parking has several different control approaches available [30]: some approaches use specific servers that mark occupied parking spot, others use camera to analyze the spot and understand if there is a vehicle inside, others use LiDar sensors or Ultrasonic sensors. Due to high customer value, car companies are developing parking systems and functions related to parking assistance [31].

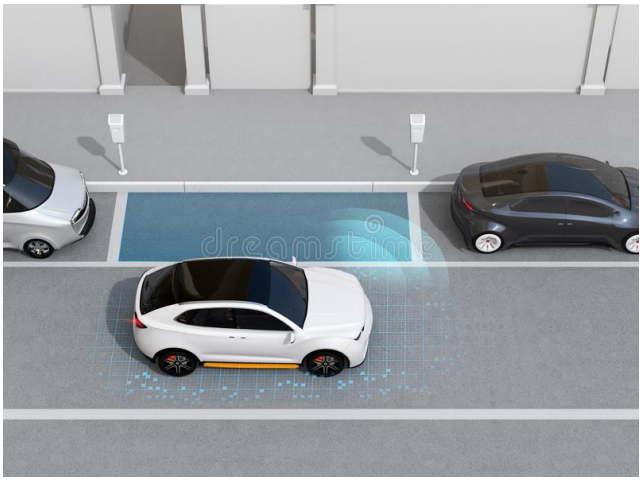

Figure 30: Autonomous parking.

6.1 Parking at competition

Parking is an important task in autonomous vehicle and it is mandatory for the competition. In the track is known that parking station is composed by two parking spots. The car is provided by a camera, a pointing LiDar and Ultrasonic sensor and the team exploits them to accomplish the parking manoeuvre.

Team Politron develops Parking manoeuvre (Fig.31) in *MovCarProcess* making use of *SignDetectionProcess*. *SignDetectionProcess* exploits the camera to recognize traffic sign and sends a flag that corresponds to the detected sign. Parking manoeuvre starts when *SignDetectionProcess* detects the parking sign: when the parking sign is detected, the HC-SR04 is used to evaluate if the first parking spot is empty or is busy. Next, it scans the right-hand side of the car and saves the obstacle position, then it starts the manoeuvre depending on it. At the same time, the car uses IMU sensor to go straight. If the first lot is empty, counters are used to mark the different steps of the parking manoeuvre:

1. The car places its middle lateral axis perpendicular to the last line delimiting the parking lot.

2. The car steers the wheels and starts the backward motion.
3. The car steers again the wheels to have them straight and to complete the parking manoeuvre.
4. If the car is not too near to another car, it goes straight in order to reach the center of the parking spot. If it is too near, LiDar detects it and the car stops.
5. The car starts the backward motion.
6. The car steers the wheels and starts the forward motion.
7. The car steers again the wheels in the other side to go back into the lane.

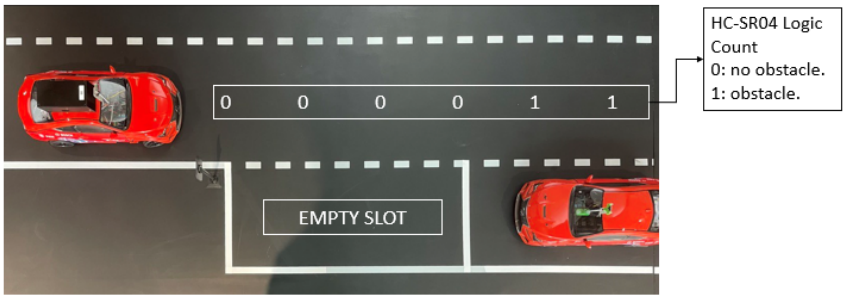

Figure 31: Autonomous parking in the competition.

If the first parking spot is busy, the car goes straight until HC-SR04 sensor detects an empty parking spot and immediately after the seven steps written above are executed.

```
# PARKING MANOUVRE
elif PARKING_MANOEUVRE == True:
    cnt = cnt + 1
    if cnt == 1:
        print("Parking manoeuvre initiating...")
        self.CarDetected = 2
        actual_yaw = self.Yaw
    if cnt <= 10:
        #go straight
        valore = self._straight_correction(actual_yaw)
    elif cnt > 17 and cnt < 27:
        # first slot is empty, the car is in correspondence of the second
            slot, it stops
        valore = 999
    elif cnt >= 27 and cnt < 60: #diminuito di 10
        # turn the wheels and it starts backward motion
        valore = 1000
    elif cnt >= 60 and cnt < 93: #diminuito di 20
        # turn the wheel to the other side and it starts backward motion
        valore = 1001
    elif cnt >= 93 and cnt < 130 and not SECOND_PARKING:
        #if there is a car in front and the team car is near it must stops
        if self.Lidar == 99:
            valore = 999
        else:
```

```
        #otherwise go straight in order to reach the center of the slot
            valore = self._straight_correction(actual_yaw)
    elif cnt >= 93 and cnt < 130 and SECOND_PARKING:
        if cnt < 100:
        #if the first slot in busy, go straight
            valore = self._straight_correction(actual_yaw)
        else:
            valore = 999
    elif cnt >= 130 and cnt < 155:
        # Stop in the parking slot
        valore = 999
    elif cnt >= 155 and cnt < 170:
        # Start the backward motion
        valore = 1000
    elif cnt >= 170 and cnt < 180:
        # Turn wheel and start forward motion
        valore = 2001
    elif cnt >= 180:
        cnt = 0
        #end PARKING manoeuvre
        PARKING_MANOEUVRE = False
        NORMAL = True
        print("Parking manoeuvre completed")
    else:
        pass
```

The parking algorithm used in the competition is not general; manoeuvres are set for the parking spot of the competition track, they are preset knowing the size of the parking spot. Furthermore, parallel parking is the only one that the car is able to fulfill.

7 Conclusion and future development

This work has dealt with the participation to the Bosch Future Mobility challenge. It is an international autonomous driving and connectivity competition for bachelor and master students created by Bosch Romania. Nowadays, self driving is one of the most discussed topics and it develops faster and faster. Students join the challenge to learn more about autonomous driving and to develop the concerning algorithms. The author was member of team Politron and together with the other colleagues of the team decided to face the challenge. They win the Best New participating Team award.

This thesis deepens the deployment of a trajectory planning and path planning system for the autonomous 1:10 scaled vehicle of the competition. To do that, TF-Luna LiDar sensor and HC-SR04 Ultrasonic Sensor Module were used. In qualifications, the former was able to detect obstacle but it was positioned in the wrong way, it was pointing to the ground, so the car continuously stops because it detects a wall that did not exist; instead in finals, the car stops because LiDar laser points on the road sign post concluding the overtake manoeuvre. HC-SR04 Ultrasonic Sensor Module, both in qualifications and in finals, had no problem to detect right-side obstacle.

At competition, the team chose a predefined path that the car must follow to fulfill all the tasks required. Trajectory planning algorithms were used in the competition and the car followed the chosen path for both the technical and the speed race as the team expected. Firstly, algorithms saves the graph that corresponds to the competition track and connecting to the GPS, they were able to understand the position of the car on the track. Next, the behaviour of the car is set: if the nodes are in the intersection, the car must accomplish the behavior selected for that certain node in order to follow the chosen path, otherwise the car must follow the lane.

Path planning algorithms finds the shortest path using the data from the saved graph of the competition track. Dijkstra algorithm processes the shortest path from source to adjacent nodes, the latter is called intermediate node. Next, it repeats the action between intermediate node to its adjacent nodes. A* algorithm finds a path that avoids collisions respecting car limits and constraints when the car structure and the position of obstacles are known. It uses a function that indicates the directions to follow and it finishes when the end point is reached. According to the function, if a step is reasonable the algorithm performs it. Heuristic methods are the base of the A* algorithm that finds the best possible solution. Both Dijkstra and A* algorithm save the path and connecting to the GPS, the car knows its position on the track and it is able to fulfill the briefest path.

All things considered, trajectory planning algorithms have demonstrated good performance during the competition; in spite of this, further improvements can be made.

Trajectory planning algorithms used in the competition and the path planning subsequently implemented can be used in the challenge thanks to the server connection given by Bosch Romania however, they must be adapted to a real GPS to be applicable in real world with hardware upgrade. Both trajectory and path planning are based on graphs composed by nodes fundamental to find the path. Manoeuvres in the intersection are set for the dimension of the track, a very good implementation is an algorithm that provides a correct behaviour for different kind of intersections.

The brain of the car is on the Raspberry Pi, a single board computer whose performance is not optimal for the required tasks. Moreover, the car is provided by a fixed PiCamera, a HC-SR04 Ultrasonic Sensor Module and a TF-Luna LiDar Module. To improve the project, a good idea is to implement a servomotor to move the camera and have a better visual of the environment and to use a better performing camera in order to increase accuracy in lane, traffic sign, traffic light and object detection. Another HC-SR04 Ultrasonic Sensor Module put in front of the camera can improve the functionality of the pointing LiDar because it has a bigger radius to sense if an obstacle is approaching, like a pedestrian that wants to cross the road. Instead of TF-Luna LiDar Module, it can be useful to employ a 360° LiDar that can improve in a particular way parking manoeuvre.

References

[1] "Bosch invented for life," Documentation. [Online]. Available: https://boschfuturemobility.com/documentation-main/

[2] N. Deo and M. M. Trivedi, "Multi-modal trajectory prediction of surrounding vehicles with maneuver based lstms," in *2018 IEEE Intelligent Vehicles Symposium (IV)*, 2018, pp. 1179–1184.

[3] S. Abdallaoui, E.-H. Aglzim, A. Chaibet, and A. Kribèche, "Thorough review analysis of safe control of autonomous vehicles: Path planning and navigation techniques," *Energies*, vol. 15, no. 4, 2022. [Online]. Available: https://www.mdpi.com/1996-1073/15/4/1358

[4] J. M. Scanlon, K. D. Kusano, T. Daniel, C. Alderson, A. Ogle, and T. Victor, "Waymo simulated driving behavior in reconstructed fatal crashes within an autonomous vehicle operating domain." [Online]. Available: https://storage.googleapis.com/waymo-uploads/files/documents/Waymo-Simulated-Driving-Behavior-in-Reconstructed-Collisions.pdf

[5] B. Gringer, "History of the autonomous car." [Online]. Available: https://www.titlemax.com/resources/history-of-the-autonomous-car/

[6] J. Fayyad, M. A. Jaradat, D. Gruyer, and H. Najjaran, "Deep Learning Sensor Fusion for Autonomous Vehicle Perception and Localization: A Review," *Sensors 20*, 2020, no. 15: 4220. [Online]. Available: https://doi.org/10.3390/s20154220

[7] S. Kuutti, R. Bowden, Y. Jin, P. Barber, and S. Fallah, "A Survey of Deep Learning Applications to Autonomous Vehicle Control," *IEEE Transactions on Intelligent Transportation Systems*, vol. 22, 07 January 2020, doi: 10.1109/TITS.2019.2962338.

[8] J. M. Anderson, N. Kalra, K. D. Stanley, P. Sorensen, C. Samaras, and O. A. Oluwatola, "Autonomous vehicle technology," *RAND Corporation*, isbn: 978-0-8330-8398-2.

[9] H. A. Ignatious, Hesham-El-Sayed, and M. Khan, "An overview of sensors in Autonomous Vehicles," *Elsevier B.V*, 2021. [Online]. Available: www.sciencedirect.com

[10] Vargas, Jorge, S. Alsweiss, O. Toker, R. Razdan, , and J. Santos, "An Overview of Autonomous Vehicles Sensors and Their Vulnerability to Weather Conditions," *Sensors 21*, 2021, no. 16: 5397. [Online]. Available: https://doi.org/10.3390/s21165397

[11] U. Wandinger, "Introduction to Lidar. In: Weitkamp, C. (eds) Lidar. Springer Series in Optical Sciences," *Springer, New York, NY*, vol. 102, 2005, isbn: 978-0-387-40075-4. [Online]. Available: https://doi.org/10.1007/0-387-25101-4_1

[12] J. Liu, Q. Sun, Z. Fan, and Y. Jia, "Tof lidar development in autonomous vehicle," in *2018 IEEE 3rd Optoelectronics Global Conference (OGC)*, 2018, pp. 185–190, doi: 10.1109/OGC.2018.8529992.

[13] B. S. Lim, S. L. Keoh, and V. L. L. Thing, "Autonomous vehicle ultrasonic sensor vulnerability and impact assessment," in *2018 IEEE 4th World Forum on Internet of Things (WF-IoT)*, 2018, pp. 231–236, doi: 10.1109/WF-IoT.2018.8355132.

[14] "Complete Guide for Ultrasonic Sensor HC-SR04 with Arduino." [Online]. Available: https://randomnerdtutorials.com/complete-guide-for-ultrasonic-sensor-hc-sr04/

[15] N. Ahmad, R. A. R. Ghazilla, , and N. M. Khairi, "Reviews on Various Inertial Measurement Unit (IMU) Sensor Applications," *International Journal of Signal Processing Systems*, vol. 1, no. 2, 2013.

[16] K. Jo, J. Kim, D. Kim, C. Jang, and M. Sunwoo, "Development of autonomous car—part i: Distributed system architecture and development process," *IEEE Transactions on Industrial Electronics*, vol. 61, no. 12, pp. 7131–7140, 2014, doi: 10.1109/TIE.2014.2321342.

[17] R. C. Shit and S. Sharma, "Localization for autonomous vehicle: Analysis of importance of iot network localization for autonomous vehicle applications," in *2018 International Conference on Applied Electromagnetics, Signal Processing and Communication (AESPC)*, vol. 1, 2018, pp. 1–6.

[18] Fujii, Sae, A. Fujita, Umedu, Takaaki, Kaneda, Shigeru, Yamaguchi, Hirozumi, Higashino, Teruo, Takai, and Mineo, "Cooperative vehicle positioning via v2v communications and onboard sensors," in *2011 IEEE Vehicular Technology Conference (VTC Fall)*, 2011, pp. 1–5.

[19] "NetworkX, Network Analysis in Python." [Online]. Available: https://networkx.org

[20] "Bosch future mobility challenge," GITHUB, Copyright (c) 2019, Bosch Engineering Center Cluj and BFMC organizers. All rights reserved. [Online]. Available: https://github.com/ECC-BFMC

[21] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics, Modelling, Planning and control*. Springer-Verlag London Limited, 2009, isbn: 978-1-84628-641-4, doi: 10.1007/978-1-84628-642-1.

[22] A. Trotta, "Trajectory planner for autonomous vehicle," 2016, master Degree's Thesis at Politecnico di Torino.

[23] A. E. Mahdawy and A. E. Mougy, "Path planning for autonomous vehicles with dynamic lane mapping and obstacle avoidance." in *In Proceedings of the 13th International Conference on Agents and Artificial Intelligence (ICAART 2021)*, vol. 1, 2021, pp. 431–438, isbn: 978-989-758-484-8.

[24] P. Mishrs, B. S. Sujith, and K. Mall, "A novel path planning algorithm for autonomous robot navigation," in *2010 International Conference on Computer Applications and Industrial Electronics*, 2010, pp. 180–185.

[25] P. Ren, S. Chen, and H. Fu, "Intelligent path planning and obstacle avoidance algorithms for autonomous vehicles based on enhanced rrt algorithm," in *2021 6th International Conference on Communication and Electronics Systems (ICCES)*, 2021, pp. 1868–1871.

[26] P. Bautista-Camino, A. I. Barranco-Gutiérrez, I. Cervantes, M. Rodríguez-Licea, J. Prado-Olivarez, and F. J. Pérez-Pinal, "Local path planning for autonomous vehicles based on the natural behavior of the biological action perception motion." *MDPI, Wiseman Yair and Antonio Cano-Ortega*, 27 February 2022, energies 2022, 15,1769. [Online]. Available: https://doi.org/10.3390/en15051769

[27] Qing, Guo, Zheng, Zhang, Yue, and Xu, "Path-planning of automated guided vehicle based on improved dijkstra algorithm," in *2017 29th Chinese Control And Decision Conference (CCDC)*, 2017, pp. 7138–7143, doi: 10.1109/CCDC.2017.7978471.

[28] W. Yijing, L. Zhengxuan, Z. Zhiqiang, and L. Zheng, "Local path planning of autonomous vehicles based on a algorithm with equal-step sampling," in *2018 37th Chinese Control Conference (CCC)*, 2018, pp. 7828–7833.

[29] "A* search algorithm," © 2013-2022 Stack Abuse. [Online]. Available: https://stackabuse.com/courses/graphs-in-python-theory-and-implementation/lessons/a-star-search-algorithm/

[30] D. Pérez-Morales, S. Domínguez-Quijada, O. Kermorgant, and P. Martinet., "8th workshop on planning, perception and navigation for intelligent vehicles at ieee int. conf. on intelligent transportation systems," in *Autonomous parking using a sensor based approach.*, Nov 2016, Rio de Janeiro, Brazil.

[31] P. Strömberg, "Device-less remote parking based on ultrasonic sensor detection," in *Virtual Leash*, 2019.

Ringraziamenti

Vorrei ringraziare la mia famiglia per avermi sempre sostenuto, per essermi stata vicina nei momenti di crisi e per avermi dato la possibilità di crescere facendomi intraprendere il mio percorso universitario dall'altra parte dell'Italia. Tra loro, vorrei citare Ginevra che nell'ultimo anno, nonchè il suo primo anno di vita, è riuscita a portare dentro di me una felicità immensa.

Inoltre vorrei ringraziare tutti i ragazzi del Team Politron e il professore Stefano Malan per l'esperienza che mi hanno permesso di vivere. La BFMC 2022 rimarrà sempre un ricordo bellissimo. Vorrei anche ringraziare il Politecnico di Torino e tutte le persone che ci lavorano per avermi fatto maturare e comunque per essere stati parte di un bel po' di anni della mia vita.

Infine vorrei ringraziare, Daniel per avermi fatto rialzare tutte le volte che sono caduta, per essermi stato vicino, per avermi sopportato e per avermi cucinato ogni volta che non l'avrei fatto. Le Trigone che nonostante la distanza ci sono sempre, e dico sempre, state. Le mie amiche e i miei amici "la famiglia che ti scegli" che hanno creduto in me tra cui Pipino per essere stato il miglior Cotino di sempre. Rinomino Daniel e Cuchina per ringraziarli particolarmente per la pazienza che hanno avuto durante la scrittura di questa tesi, per avermi aiutata e sopportata.

E come non potrei ringraziare Hardin, il nostro piccolo e peloso coniglietto che si è subito le spiegazioni della parte teorica di molte materie.