ETH zürich
Institute for Dynamic Systems and Control

Pedro Reyero Santiago

# Sensor rejection for reliable state estimation of an autonomous last-mile delivery vehicle

**Master's Thesis**

Institute for Dynamic Systems and Control
Swiss Federal Institute of Technology Zurich

**Supervision**

Dr. Erik Wilhelm
Prof. Dr. Christopher Onder

July 2021

# Abstract

State estimation is a critical component of an autonomous vehicle. Safe, accurate and reliable localization can be a challenging task, especially in worst-case scenarios where sensors may suffer from poor performance or failure (for instance, a GPS system inside a building). In these cases, the sensors provide corrupt measurements. Previous works have studied the fusion of valid information from different sensors, however the sensor rejection problem has barely been explored in a general, non algorithm-specific formulation. In this thesis, the online cross-validation and rejection of pose estimates coming from sensors that may fail is studied. This is the applied to the Autonomous Plus2, an autonomous vehicle prototype for last-mile delivery developed by Kyburz Switzerland AG. First, a general overview of autonomous vehicle sensors and their typical failure modes is presented. The specific localization algorithms present on the Autonomous Plus2 are also described. Then, existing sensor rejection approaches are explored, based on parity-based Fault Detection and Isolation (FDI) techniques. New approaches are studied, based on Machine Learning (ML) techniques (decision trees, k-NN, feedforward neural networks and RNNs). A solution combining the aforementioned techniques is proposed, to improve reliability. Finally, an experimental setup is presented and implemented to test and tune each algorithm on the Autonomous Plus2. Various challenging scenarios are considered, to assess performance, trade-offs and weaknesses. The final proposal is a combined decision algorithm, capable of a 70% rejection rate of failed measurements while only losing 5% of the valid measurements.

**Keywords:** Pose estimation, Autonomous vehicles, Reliability, Sensor rejection, Fault Detection and Isolation, Machine Learning.

# Acknowledgment

Throughout the writing of this thesis, I have received a great deal of support and assistance, which have been key to reaching the final result I hereby present.

I would first like to thank my supervisor at Kyburz, Dr. Erik Wilhelm, whose constant and active feedback was invaluable in formulating the thesis problem and coming up with solutions to tackle it successfully, as well as refine and improve results and aim the project in the right direction in difficult times.

I would like to thank my supervisor at ETH Zürich, Prof. Dr. Christopher Onder, who put his trust in an exchange student and granted me the great and unique opportunity to develop this thesis in collaboration between university and industry. His support and kind feedback was very useful to keep the project progressing on a sound and fruitful track.

I would also like to thank the rest of the team in Kyburz, and in particular the people in the research team, who always offered me their help and time to solve issues with the vehicle and overcome obstacles on the way. Their assistance allowed me to get used to working with the Autonomous Plus2 very fast and to collect data and test efficiently and extensively.

In addition, I would like to thank my family and friends for their support and care, which was indispensable for reaching where I am right now. I owe them my personal and academic development over these seven years at university, as well as some of my dearest memories during that time.

# Contents


**1 Introduction** **1**
1.1 Problem statement . . . 1
1.2 Previous and related research . . . 2
1.3 Thesis outline . . . 3

**2 Sensors and sensor failure in autonomous vehicles** **5**
2.1 GPS . . . 5
2.2 IMU . . . 6
2.3 INS . . . 7
2.4 LiDAR . . . 8
2.5 Camera . . . 8
2.6 Wheel odometry . . . 11
2.7 Others . . . 11

**3 Pose estimation on the Autonomous Plus2** **13**
3.1 Google Cartographer . . . 13
3.2 GPS/INS . . . 15
3.3 Dragonfly . . . 16
3.4 Dead reckoning . . . 17
3.5 ROS architecture and pipeline . . . 18

**4 Sensor rejection and trust estimation** **21**
4.1 Manual heuristics (rule-based) . . . 21
4.2 Fault Detection and Isolation methods . . . 22
4.2.1 Extended NIS sensor validation . . . 22
4.3 Learning-based approaches . . . 25
4.3.1 Classical ML classification methods . . . 28
4.3.2 Fully-connected feedforward neural networks . . . 31
4.3.3 Sequence models . . . 32
4.4 Voting systems . . . 34

**5 Experimental procedure** **37**
5.1 Performance evaluation . . . 37
5.2 Test scenarios description . . . 38
5.3 Data acquisition and pipeline . . . 38
5.4 Training and parameter tuning procedures . . . 42

**6 Results and Discussion** **45**
6.1 Model training and tuning results . . . 45
6.1.1 ExNIS cross-validation . . . 45
6.1.2 Decision trees . . . 48
6.1.3 k-Nearest Neighbors . . . 50

6.1.4 FC feedforward NN . . . 51
6.1.5 Recurrent NN . . . 53
6.1.6 Note on performance metrics . . . 55
6.2 Final performance in test scenarios . . . 55

**7 Conclusion** **59**

**A Code** **61**
A.1 Data pre-processing . . . 61
A.2 Algorithm nodes . . . 70
A.3 Other ROS files . . . 77

**Bibliography** **79**

# Chapter 1

# Introduction

## 1.1 Problem statement

This project's objective is to develop and deploy strategies for improving reliability of state estimation for Kyburz's autonomous vehicle *Autonomous Plus2* (see Figure 1.1), which is designed for last-mile mobility.

In particular, sensor rejection schemes are developed to deal with worst-case scenarios, where one or more of the different pose estimation sources fail to provide a reliable estimate, but still provide an estimate (and potentially also a covariance matrix associated to it) that has to be rejected. Common sensor fusion techniques rely on covariance matrices for filtering new incoming information and producing a new state estimate, and thus are biased towards precision. In normal situations, when all sensors/sources are providing reliable information, these approaches provide accurate and precise results with relatively low computational cost. In "sensor failure" scenarios, however, some of the sources may provide apparently very precise estimates (according to their statistical information), which are actually not accurate. Those estimates have to be rejected before filtering, in order to avoid corrupting the vehicle's state estimate by fusing inaccurate information (which filtering techniques may trust due to their high precision).

The *Autonomous Plus2* has many on-board sensors which provide information that can be used for localization purposes. Seizing the available setup (both hardware and software) at the project start date, in this project it is considered that the *Autonomous Plus2* can receive 2D pose estimates (i.e. 2 position variables and 1 orientation variable) from 4 different sources, which are described in detail in Chapter 3. Those estimates come as pose vectors of the form $[x, y, \theta]$, obtained by processing and filtering data from several sensors and sometimes also using localization algorithms (e.g.: SLAM). Those 4 estimates are available to decide which sources can be trusted and which should be rejected. In this project, schemes/algorithms are developed to make this decision autonomously and on-the-fly (before, this decision was taken manually by a skilled member of the Kyburz team before starting navigation, and it couldn't be changed afterwards). This decision comes in the form of a binary decision vector $[s_1, s_2, s_3, s_4]$ (with 0 meaning source rejection) and can be complemented with trust/belief information, to express the degree of reliability of each source. Such information can also be additionally used by the vehicle's planning and decision systems in very challenging worst-case scenarios (where several sensors aren't reliable) in order to re-plan motion to ensure safe navigation (e.g.: by lowering path speeds). Figure 1.2 summarizes the system developed in this project.

Figure 1.1: **Autonomous Plus2**, the autonomous vehicle prototype used in this thesis.

## 1.2 Previous and related research

In order to improve reliability on autonomous vehicle state estimation, many techniques have been developed and tested across literature. There is quite a broad consensus in the community that in order to avoid the perceptual limitations and uncertainties of a single sensor, to build a richer comprehension about the vehicle's environment and to improve its overall external perception abilities, it is essential to integrate information obtained by different sensors [1].

Multi-source and heterogeneous information fusion (MSHIF) can be performed at different levels of abstraction from the original data in the fusion phase, which motivates the following classification from [1]:

- *Fusion strategies based on discernible units:* Direct fusion of the data coming from each sensor, and further processing of the data after the fusion.
- *Fusion strategies based on complementary features:* For each target, combination or fusion of features obtained with each sensor (to build a richer representation of each target) and subsequent classification or recognition of each target using the fused multi-sensor features.
- *Fusion strategies based on target attributes:* Different sensors are used to generate different target lists, which are then fused to acquire a reliable target list with reliable information, avoiding false alarms and missed detections.

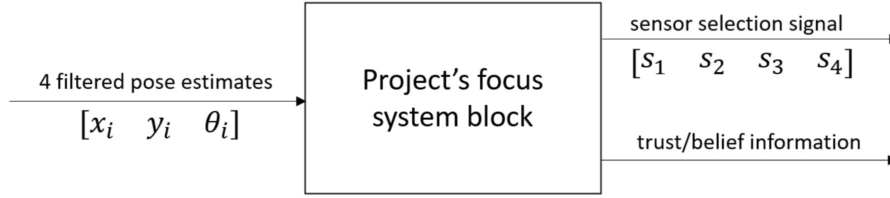


Figure 1.2: Schematic representation of this project's inputs, outputs and developed system.

- *Fusion strategies based on multi-source decision:* For each sensor, a decision is made regarding location, attributes or categories of the target, and then fusion strategies are applied to combine the decisions from individual sensors.

The problem proposed in this thesis can be classified as a part of a fusion strategy that lies in the last of the former categories. The Autonomous Plus2 has 4 independent sources of 2D pose estimates which are generated using different sensors (although some may share, for instance, the IMU data). Those 4 preliminary estimates have to be used to produce a final pose estimate, which may or may not use the full information from each preliminary estimate. In this thesis, this fusion problem is broken into 2 distinctive steps, and the focus is set on only the first of them:

1. The 4 pose estimates are used to cross-validate their information and detect which sources are reliable and which may be falling into a "failure mode", a failure that can be just temporary or need a reset routine. The output of this step are a sensor selection signal for the next step and trust/belief information on each sensor's estimates.
2. Using the sensor selection signal from the previous step, and potentially also its trust/belief information, the non-rejected estimates are fused into a final estimate that should ideally be more accurate and reliable.

The latter step is a well-studied and active topic across the literature, for which many algorithms have been successfully applied. Two of the most used algorithms for it are Kalman Filters and Particle Filters, with countless variants proposed, depending on the characteristics of the sensors used on each vehicle and the algorithms used to process raw sensor data. Some previous works on this task for autonomous last-mile delivery vehicles are [2], [3] and [4].

The former step, however, is not a very explored topic across literature, especially with this formulation, that is, independent of which algorithms were used to generate the pose estimates (algorithm-specific sensor selection methods have been presented, for instance, in [5], where the method is restricted to sensors/algorithms that provide a 2D discrete grid-map). A few related previous works on general sensor rejection and/or cross-validation using residuals and fault detection techniques can be found in [6], [7] and [8]. More modern techniques (e.g.: Machine Learning approaches) haven't been applied to this particular problem, to the best of the writer's knowledge.

## 1.3 Thesis outline

The remainder of this thesis is organized as follows. Chapter 2 describes the main sensors used in autonomous vehicles, their characteristics and the different scenarios/situations in which they can "fail" and provide non-accurate information. Chapter 3 presents the 4 sources of 2D pose estimates used on the Autonomous Plus2 for this project, together with the way their output information is preprocessed/treated before feeding it to the decision schemes. Chapter 4 presents the different approaches to sensor rejection that are explored in this thesis. Chapter 5 describes how each scheme's performance is assessed (metrics, test scenarios, etc.) and how each scheme's parameters are tuned/learned. Chapter 6 contains the final experimental results for each scheme and scenario, together with an assessment of their overall performance and reliability. Finally, Chapter 7 summarizes the main conclusions of this thesis.

# Chapter 2

# Sensors and sensor failure in autonomous vehicles

In order to localize an autonomous vehicle, information about ego motion or the environment has to be gathered. This can be done using many sensors, which can provide various types of raw information that can be later processed by localization algorithms (e.g.: SLAM) to generate an estimate of "where the vehicle is". Some sensors can provide local/relative information (e.g.: position of obstacles with respect to the ego vehicle, relative ego movement...), while others provide global information (e.g.: position of the ego vehicle with respect to a fixed terrestrial reference frame).

Although there are a lot of different sensors that can be used nowadays in autonomous vehicles, here the focus is put on the sensors that are currently being used on the Autonomous Plus2 for localization. For those sensors, their main characteristics, working principles, advantages and disadvantages are presented, together with a discussion on sensor "failure modes" that may arise during typical operation of the vehicle (either due to the sensors' intrinsic characteristics or due to typical associated algorithms' limitations) and that can be triggered in a controlled experimental setup. These failure modes condition and motivate some of the different test scenarios presented in Chapter 5, which focus on triggering sensor failure to assess algorithm performance in worst-case scenarios. Finally, this chapter also includes a brief discussion of other common sensors present in the Autonomous Plus2 that could be used for localization in the future.

## 2.1 GPS

GPS is a constellation of satellites that provides a user with an accurate position on the surface of the earth [9]. A GPS receiver can provide precise global position information to an autonomous vehicle anywhere in the world, by locking on to any visible satellites. This information is provided in the form of latitude and longitude, which can be transformed into northing and easting.

The GPS system has a constellation of 24 satellites in fixed orbits around the earth (see Figure 2.1), each of which continuously transmits GPS signals to the earth. These signals consist of 2 carrier frequencies, digital codes and a navigation message. The distance between satellite and receiver can be determined using the carrier frequencies and digital codes, while the navigation message is used for clock compensation, knowing the satellite's location, etc. GPS receivers use trilateration [10], which requires at least 4 visible satellites at a given time in order to provide the position estimate (see Figure 2.1), but can greatly benefit from having more than 4 satellites visible.

In Table 2.1, 3 failure modes are considered for GPS systems [11], which are also graphically depicted in Figure 2.2. Satellite visibility is key for a good GPS performance, so any situation that may affect it is likely to trigger a reduction in accuracy (e.g.: when navigating between trees) or

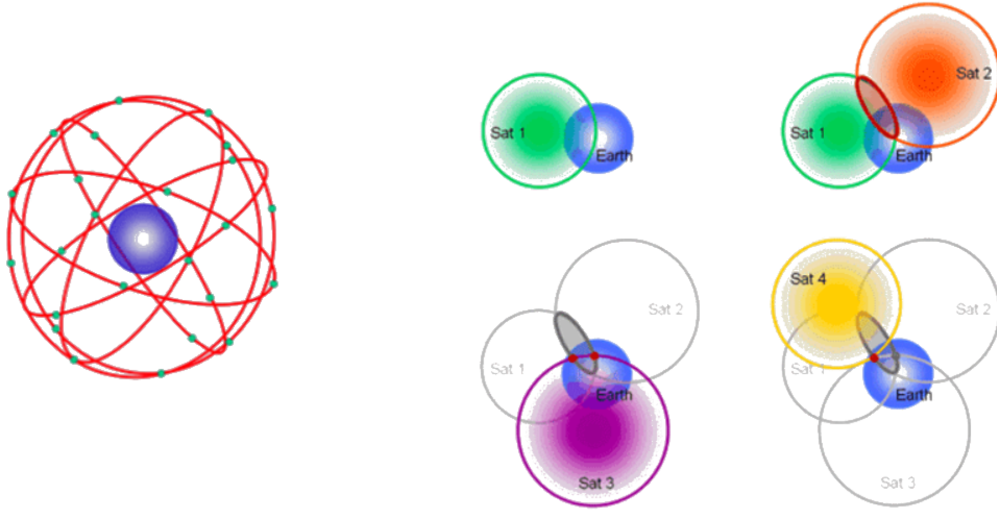


Figure 2.1: The GPS network has 24 satellites that orbit in 6 planes around the Earth. GPS trilateration needs at least 4 satellites visible at a given time [10].

Table 2.1: Summary of considered GPS failure modes.

| Failure mode | Code | Description |
|---|---|---|
| Satellite blockage | GPS_1 | The number of visible satellites decreases due to blockage by elements like buildings, trees, bridges, etc. |
| Multi-path effect | GPS_2 | Time-of-flight calculations are corrupted by GPS signal getting reflected off of buildings, walls, etc. |
| Complete ceiling above vehicle | GPS_3 | Indoor or underground navigation impose a physical ceiling above the vehicle, blocking proper satellite visibility. |

even produce a corrupted estimate (e.g.: when entering a building, where a ceiling blocks satellite visibility). Another common and easy-to-trigger failure mode for GPS is the so-called "multi-path effect" [12], where some GPS signals are reflected off of high buildings or walls and don't reach the vehicle directly. As GPS positioning relies on computing the distance from the satellites to the GPS receiver using the propagation time taken for the signal to travel from satellite to receiver, reflected multi-path signals (which take longer paths than direct signals) can cause significant errors.

## 2.2 IMU

Initial measurement units (IMUs) are sensors that can directly measure the vehicle's 6 DOF acceleration, through the combination of an accelerometer (three linear acceleration components) and a gyroscope (three rotational acceleration components). Some IMU units also include a magnetometer, although they are not very useful for autonomous vehicles due to the presence of magnetic fields from the ego vehicle and other nearby vehicles [13].

IMUs provide a way to estimate ego motion without interacting with the environment or having any knowledge from it, which is very useful for localization algorithms in terms of safety and reliability. This environment-independent nature allows IMUs to track position even in complicated scenarios, like slipping and skidding, where tires lose traction and measurements from wheel odometry become inaccurate.

In Table 2.2, 1 failure mode is presented for IMUs [11], which is also graphically depicted in Figure 2.3. When an autonomous vehicle enters an area where it suffers significant vibration (e.g.: uneven

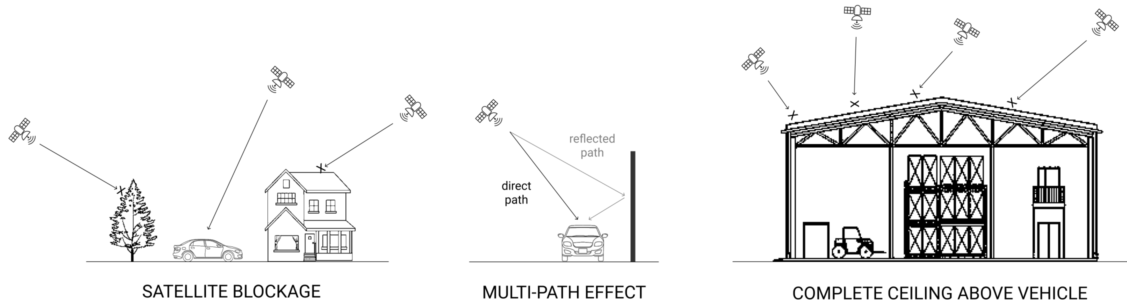


Figure 2.2: Schematic representation of situations that can trigger the considered GPS failure modes in Table 2.1.

Table 2.2: Summary of considered IMU failure modes.

| Failure mode | Code | Description |
| --- | --- | --- |
| Vibration | IMU_1 | Heavy vibration (e.g.: due to terrain conditions) can affect inertial measurements. |

or very rough floors), IMUs ingest these vibrations as a part of their acceleration measurements, altering the resulting signal.

## 2.3 INS

An inertial navigation system (INS) takes the readings from GPS, gyroscope, accelerometer and magnetometer and fuses them to achieve a more accurate and reliable output. Inertial navigation systems must contain some sort of central processing unit (CPU) in order to perform the sensor fusion of GPS and IMU. While GPS provides global positioning information, IMUs provide information relative to their last known position. The fusion of these two sources (for instance, running a Kalman Filter on its CPU) also allows an INS to reduce problems like drift or magnetic interference in heading estimation, allowing for proper compassing and ultimately providing a better estimate of the ego vehicle's pose. This fused inertial and global positioning information can be used, for instance, to confirm and update position with respect to high precision maps in applications combining INS with other sensors like LiDAR [1].

As INS systems fuse GPS and IMU readings, they may enter failure whenever one of those two main components does (and thus no separated failure modes have been defined for INS systems). It may also happen that the other component (and the algorithm on the CPU) is enough to compensate the problem and the INS system does not provide corrupt or unreliable estimates even in situations that would be challenging for one of the components individually.

Figure 2.3: Schematic representation of a situation that can trigger the considered IMU failure mode in Table 2.2.

## 2.4 LiDAR

A LiDAR (Light Detection and Ranging) sensor allows to determine the distance between a laser transmitter and an object or surface using a beam of pulsed laser. The distance to the object is determined by measuring the time difference between the pulse emission and the detection of the reflected signal (see Figure 2.4). These reflections conform a point cloud that represents the environment around the vehicle (both static and dynamic elements), which can be used by algorithms for positioning, obstacle detection, environmental reconstruction, etc. The most common wavelength for LiDARs nowadays is in the 900 nm range (the Autonomous Plus2, in particular, uses a LiDAR with a 850 nm wavelength), although longer wavelengths may be used to obtain better performance under challenging weather conditions (e.g.: rain, fog...) [14].

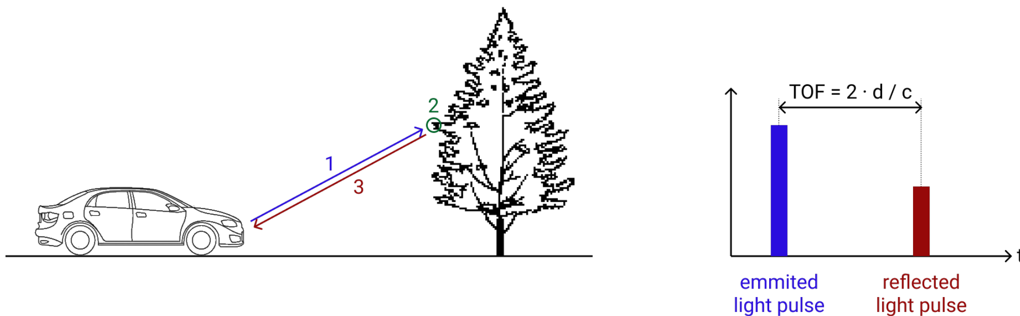


Figure 2.4: Working principle of LiDAR sensors: time-of-flight (TOF) [15]. Distance $d$ to the detected obstacle is computed with the TOF and the speed of light $c$.

There are two main types of LiDAR systems [16]: mechanical LiDARs and solid-state LiDARs. The Autonomous Plus2 has a mechanical LiDAR, which uses high-grade optics and a rotating assembly to create a wide Field Of View (usually 360º). The resulting implementation is bulky, but offers a high signal-to-noise ratio over a wide FOV. They are the most commonly used LiDAR sensors, although their price tag make them quite restrictive in some cases.

LiDAR systems present much higher spatial resolution than Radar, because of the more focused laser beam, the larger number of vertical scan layers and the high density of LiDAR points in each layer [14], but most LiDAR implementations can't measure velocity of objects directly, unlike Radar systems. They are also much more affected by weather conditions or dirt on the sensor, as these elements can interfere in the emission and reflection of the laser pulses.

In Table 2.3, 4 failure modes are presented for LiDAR [17], [18], some of which are also exemplified in Figure 2.5. Ground clutters are reflection points that appear, for instance, when the road has abrupt inclination changes. When using SLAM algorithms, saliency qualifies if a landmark or a set of landmarks as having enough information to compute a unique localization solution that is coherent with the real displacement [18]. Motion distortion occurs in mechanical LiDARs at non-slow speeds, where the ego motion of the vehicle during a full turn of the LiDAR shaft is significant and needs to be compensated (usually handled by LiDAR data processing pipelines [19] or localization algorithms using LiDAR raw data).

## 2.5 Camera

Cameras are one of the first sensors that were used in autonomous vehicles and are nowadays present in almost any such system [14]. They allow the vehicle to visualize its environment in a similar way a human driver would do (and even much beyond human sensing capabilities, as usually many cameras are mounted in different parts of a vehicle, even achieving 360º global FOV, as can be seen in Figure 2.6). In order to capture the scene, cameras use an imaging sensor [20],

Table 2.3: Summary of considered LiDAR failure modes.

| Failure mode | Code | Description |
|---|---|---|
| Weather conditions | LiDAR_1 | Rain, fog or snow can heavily affect LiDAR's performance. |
| Non-salient configurations | LiDAR_2 | Available landmarks do not give enough information for a unique solution (e.g.: "tunnel passage" scenarios). |
| Ground clutters | LiDAR_3 | Undulations on the ground or going up/downhill can generate an accumulation of reflection points. |
| Motion distortion | LiDAR_4 | At higher speeds, car movement during a full rotation of mechanical LiDARs can generate distortions in the point cloud. |

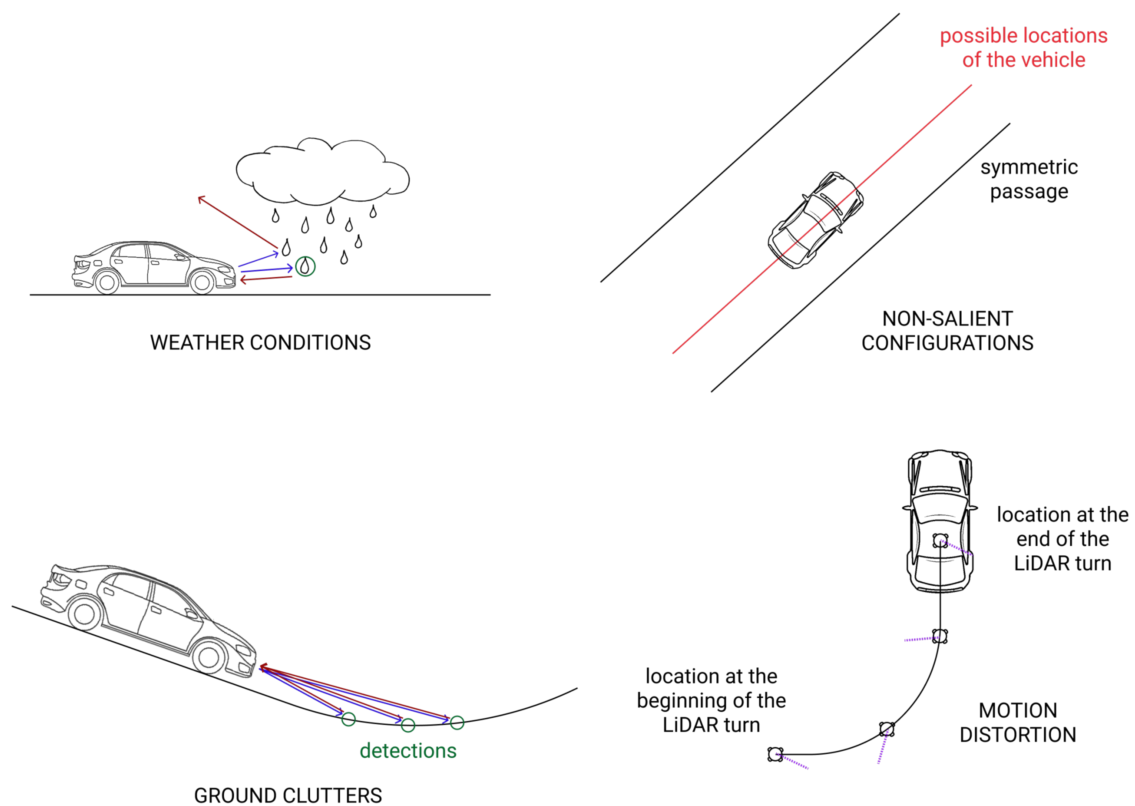


Figure 2.5: Schematic representation of situations that can trigger the considered LiDAR failure modes in Table 2.3.

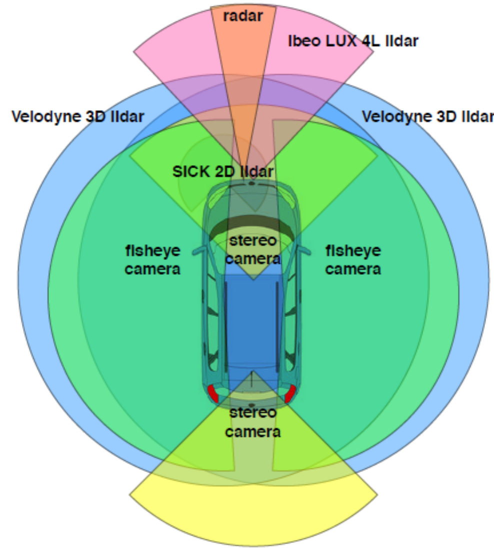


Figure 2.6: Example of sensors mounted on an autonomous vehicle [21]. Several cameras allow for a full 360º FOV, and may be used by different vision algorithms for different tasks.

for which two main technologies are used: charge-coupled device (CCD) and complementary metal oxide semiconductor (CMOS).

Cameras are a mature and very affordable technology (specially when compared to other sensors like LiDAR) that is very efficient at capturing scene texture. Latest cameras allow for high-definition image streams at high frame rates (usually between 30 and 60 fps), which need to be processed in real-time, which can also pose a computational power problem. Optical imaging allows capturing contour, texture and color distribution information [1], which can be used by algorithms for target recognition and tracking, object identification, local path planning, etc. If binocular (stereo) cameras are used, instead of monocular (mono) cameras, depth information is also obtained, similar to human vision, which can be further exploited by algorithms to improve their performance and extend their capabilities and reliability [22].

In Table 2.4, 7 failure modes are presented for camera [23], [24], some of which are also exemplified in Figure 2.7. Some camera-based localization algorithms (like the one present on the Autonomous Plus2, described in Chapter 3) have much worse performance in outdoors environments [25], due to lower feature density, usually higher distances to detected features, high proportions of the image occupied by the sky, etc. Some vehicle movements can also degrade camera-based algorithms' performance, like non-planar movements, higher speed movements, navigation on rough floors that generate significant vibration, etc.

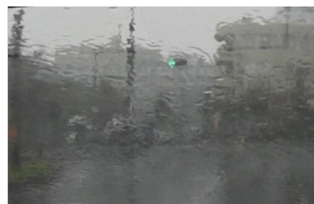

Figure 2.7: Some of the situations that can trigger considered camera failure modes in Table 2.4.

Table 2.4: Summary of considered camera failure modes.

| Failure mode | Code | Description |
| --- | --- | --- |
| Poor lighting | Camera_1 | Too dark/bright light conditions can heavily degrade camera's performance. |
| Weather conditions | Camera_2 | Rain, fog or snow can heavily affect camera's performance. |
| Outdoor navigation | Camera_3 | Some camera-based algorithms suffer significant performance degradation when operating outdoors. |
| Medium speed navigation | Camera_4 | Some camera-based algorithms suffer significant performance degradation when not operating at low speed. |
| Low-texture scenery | Camera_5 | Low texture scenes (e.g.: off-road open empty areas) don't have many visual features to use for localization. |
| Vibration | Camera_6 | Motion jitter and vibration can degrade some camera-based algorithms' performance. |
| 3D movement | Camera_7 | Non-planar movement (e.g.: going up/downhill) can mislead some camera-based algorithms. |

## 2.6 Wheel odometry

Odometry is the use of motion sensors (e.g.: wheel encoders) to determine the vehicle's ego motion relative to a previous known position [26]. Vehicles usually have shaft encoders attached to their drives' wheels that measure wheel turning and allow determining how far the vehicle has travelled by using a certain model for its motion, usually based on its kinematic configuration.

Wheel odometry is commonly used in autonomous vehicles, as its information can be very useful in scenarios where the environment doesn't provide enough feature density for LiDAR or camera systems and GPS signal is poor, but odometry is heavily affected by drift due to error accumulation in wheel sensor measurements. Some solutions have been proposed to estimate this error and compensate it [27], but such algorithms usually can only reduce the consequences of these errors. Some sources of odometry error can be inaccurate wheel diameter measurements, non-homogenous wheel sizes, counting errors in shaft encoders, slow processing of odometry data, etc.

In Table 2.5, 3 failure modes are presented for wheel odometry [28], [29]. Motion models' assumptions condition their performance, and thus bringing them to the limit (e.g.: heavy maneuvering when motion model assumes constant heading) can trigger very inaccurate pose estimates. Also, situations where the contact between floor and wheels is not ideal can progressively or abruptly degrade wheel odometry's performance.

Table 2.5: Summary of considered wheel odometry failure modes.

| Failure mode | Code | Description |
| --- | --- | --- |
| Heavy maneuver changes | Odom_1 | Abrupt changes in direction (aggressive maneuvering) can cause drift, depending on sampling time and motion model. |
| Uneven floors | Odom_2 | Travelling on uneven floors or floors with unexpected objects on them can cause wheel odometry errors. |
| Wheel slippage | Odom_3 | Slippery floors, over-acceleration and skidding in fast turns can cause wheel odometry errors. |

## 2.7 Others

The previously presented automotive sensors are currently being used on the Autonomous Plus2 for localization purposes, either alone or through some algorithm that uses more than one sensor

at a time. Nevertheless, the vehicle also has other sensors that may be used in the future for localization purposes, the most remarkable of which are Radar and Sonar systems.

Radar systems have some very interesting advantages, like being able to better operate in hard weather conditions (e.g.: fog, rain, etc.), providing a direct and independent measurement of obstacle speed, or needing less computational power to process their output data [14]. Their limited resolution (especially in the vertical direction), complex return signals or smaller angular accuracy are some of their main drawbacks. Several algorithms have been recently developed to allow localization using Radar data (for instance, [30] and [31]).

Sonar systems are usually used in autonomous ground vehicles for collision avoidance or in combination with Radar systems, but some algorithms have also been recently developed for standalone localization in the field of autonomous underwater vehicles [32], [33], where these sensors can perform reliably. Some of their advantages for autonomous ground vehicles are robustness against weather conditions, reduced cost and compact sensor size, which support this technology as a good complement to some of the other sensors already presented.

# Chapter 3

# Pose estimation on the Autonomous Plus2

The sensors presented in Chapter 2 don't directly output pose estimates, but rather provide very different information (e.g.: point-cloud data, latitude and longitude estimations, image data, etc.) that usually has to be pre-processed, handled by dedicated localization algorithms that can turn that data into actual pose estimates, and finally post-processed, in order to obtain directly comparable data that can be fed into the decision schemes of this thesis (see Figure 1.2) and later also to the fusion schemes that may be used with the non-rejected sensor data.

The Autonomous Plus2 has 4 different sources of pose estimates, each of them using different sensors (or combinations of them). Here, each of those sources are discussed, including sensors used, underlying algorithms, data pre-/post-processing and other relevant considerations. The part of the ROS pipeline present on the Autonomous Plus2 that is relevant for this thesis is also described.

## 3.1 Google Cartographer

Google Cartographer is an open-source system that provides real-time simultaneous localization and mapping (SLAM) in 2D and 3D across multiple platforms and sensor configurations [34]. In the case of the Autonomous Plus2, in particular, its ROS integration is used [35], ingesting the available information from the vehicle's LiDAR and IMU sensors. Cartographer developers claim that it can achieve real-time mapping and loop closure up to a 5 cm resolution. It has significantly lower computational requirements than other LiDAR SLAM solutions (especially when executing real-time loop closure [36]) and can apply inertial corrections with IMU data. Figure 3.1 shows an overview of the Cartographer system blocks and possible inputs that it can use. It comprises two main separated but related subsystems: "local SLAM" (for constructing submaps) and "global SLAM" (for finding loop-closure constraints).

Upon other information, Cartographer's ROS integration provides the Autonomous Plus2 with a pose estimate including 2D position and orientation information (that is, 3 degrees of freedom). Alongside that information, an estimation of the covariance matrix associated to that estimate is provided by Kyburz's proprietary code on the vehicle. This format already fits the input defined in Figure 1.2, but before being directly comparable to the rest of estimates, a translation + rotation operation (change of reference frame) has to be performed. Upon initialization of the sensor rejection system, the initial values of $x_{CG}$, $y_{CG}$ and $\theta_{CG}$ are recorded to be used for the fixed transformation between Cartographer's reference frame and a common zero-origin reference frame (where all pose variables have an initial value of 0). Any incoming pose estimate is transformed into the common reference frame using Equations 3.1 and 3.2, which consist of a translation and a rotation. As Cartographer outputs the $\theta_{CG}$ estimate wrapped between $-\pi$ and $+\pi$, the *unwrap*

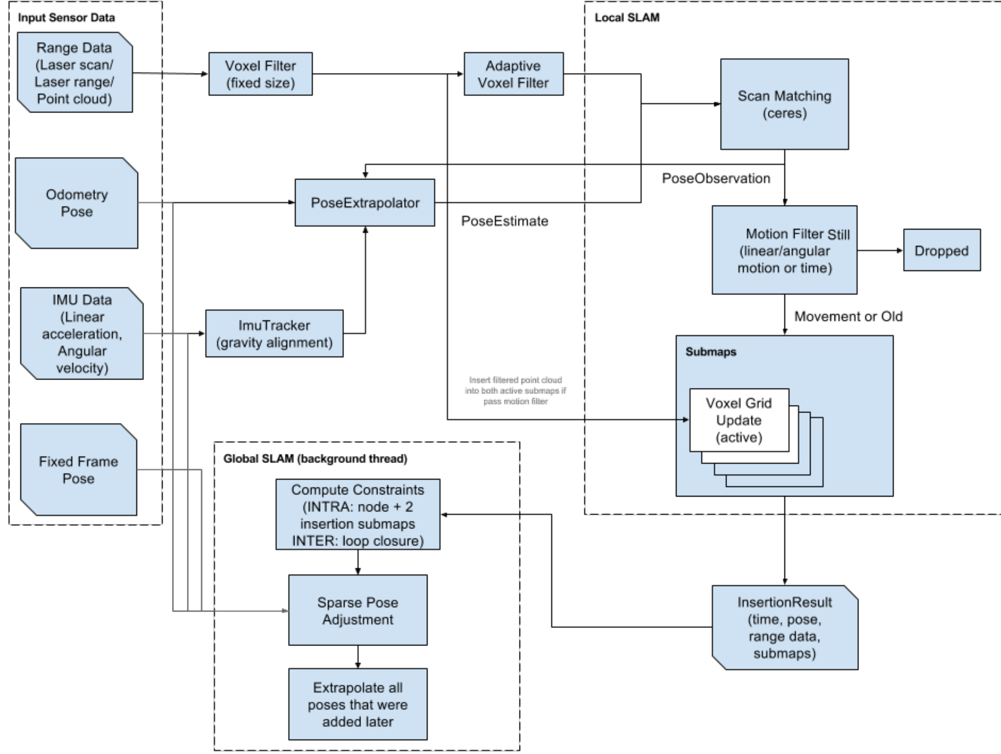


Figure 3.1: Overview of the Cartographer system [35].

function is applied before subtracting the initial $\theta_{CG}$ value to eliminate jumps and properly shift the angle. Wrapping back after shifting is not necessary and can add complexity to the algorithms (e.g.: comparing angles by direct subtraction doesn't work around the wrapping limits), so it is not performed. Here, variables with a prime symbol (e.g.: $x'_{CG}$) refer to those pose variables expressed in the common zero-origin reference frame, after applying the corresponding transformation. Thus, $x'_{CG,0}$, $y'_{CG,0}$ and $\theta'_{CG,0}$ all have a value of 0 (this convention also holds for the other sources of pose estimates).

$$\begin{bmatrix} x'_{CG} \\ y'_{CG} \end{bmatrix} = \begin{bmatrix} cos(-\theta_{CG,0}) & -sin(-\theta_{CG,0}) \\ sin(-\theta_{CG,0}) & cos(-\theta_{CG,0}) \end{bmatrix} \cdot \begin{bmatrix} x_{CG} - x_{CG,0} \\ y_{CG} - y_{CG,0} \end{bmatrix} \tag{3.1}$$

$$\theta'_{CG} = unwrap(\theta_{CG}) - \theta_{CG,0} \tag{3.2}$$

Although a covariance matrix estimate for Cartographer is produced by Kyburz's proprietary code, its values have been detected to be unreliable upon analyzing data recorded on vehicle from sensor nominal conditions. In particular, although the provided values are on average in a reasonable range (cm precision), very low values are often received (around $\mu$m precision), which alter the results from some algorithms that use these covariance matrices values and worsen their performance (although the effect isn't deal-breaking, it does influence negatively). For this reason, instead of using the provided covariance matrix values, the fixed matrix in Equation 3.3 has been defined for Cartographer, rounding the average values from gathered data, which are in physically reasonable ranges.

$$Q_{CG,fixed} = \begin{bmatrix} 0.01 & 0 & 0 \\ 0 & 0.01 & 0 \\ 0 & 0 & 0.001 \end{bmatrix} \tag{3.3}$$

## 3.2 GPS/INS

The Autonomous Plus2 has an INS that fuses GPS and IMU readings to provide an estimate in the form of latitude and longitude coordinates. In the vehicle, the GPS system also fuses vehicle speed information coming from wheel odometry.

The latitude and longitude estimates produced by the INS have to be projected into a 2D Cartesian space [37] in order to fit the input format shown in Figure 1.2. To achieve this, the *Universal Transverse Mercator* (UTM) projection is used [38], which consists on a cylindrical projection that transforms the latitude and longitude estimates into a zone number, an hemisphere (North or South), an easting and a northing. For the purposes of this thesis, the easting and northing values are used as position estimates. The value of the vehicle's orientation is obtained directly from IMU readings in the form of a heading. The implementation of the UTM projection has been translated from the one provided in [37]. The details of the transformation equations can be found in the open code [39].

Once the easting, northing and heading values are obtained, a similar transformation to the one in Equations 3.1 and 3.2 has to be used to obtain comparable pose estimates. Nevertheless, there is one additional consideration in the GPS case. Easting and northing values can be directly used as $x_{GPS}$ and $y_{GPS}$ respectively, but the heading values coming from IMU can't be directly used as $\theta_{GPS}$. While $\theta_i$ values are defined from the $x_i$ axis increasing in a counterclockwise sense, the heading values from IMU are defined from the northing axis (corresponding to the $y_i$ axis) increasing in a clockwise sense. For this reason, Equation 3.4 has to be applied to the heading readings before applying the transformation to the common reference frame. This equation shifts the angle by 90º to define it from the $x_i$ axis, and then inverts its sense to make it increase counterclockwise. Figure 3.2 portrays the conflict between the two orientation values. Once the value of $\theta_{GPS}$ is obtained, a transformation completely analogous to Equations 3.1 and 3.2 is applied to obtain the pose estimate in the common reference frame. Due to the subtraction of the initial $\theta_{GPS}$ value in the transformation, omitting the 90º shift would yield the same final results, but here the shift is kept for clarity of the transformation between heading and $\theta_{GPS}$.

$$\theta_{GPS} = -(Heading - \pi/2) \tag{3.4}$$

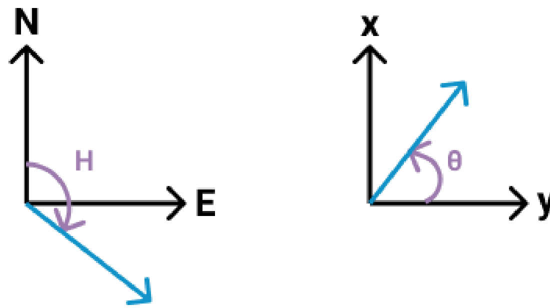


Figure 3.2: Reference axes and positive senses for heading and $\theta_i$ (also $\theta'_i$).

Note that no covariance matrix for INS is produced by Kyburz's proprietary code. For this reason, and taking into account orders of magnitude of the errors and comparative errors between the different sources, the fixed matrix in Equation 3.3 has been defined for GPS/INS.

$$Q_{GPS,fixed} = \begin{bmatrix} 0.1 & 0 & 0 \\ 0 & 0.1 & 0 \\ 0 & 0 & 0.001 \end{bmatrix} \tag{3.5}$$

## 3.3 Dragonfly

Dragonfly is a commercial system that provides real-time simultaneous localization and mapping (SLAM) in 3D (6 degrees of freedom) using a standard camera [40]. It offers support for mono and stereo cameras, and claims to achieve centimeter-level accuracy in indoor scenarios (usage in outdoor scenarios is discouraged by the developers). In the Autonomous Plus2, its available ROS integration is used, which is configured to work with the vehicle's front stereo camera, using the on-board local computation architecture described in Figure 3.3, which relies on the vehicle's computational power in exchange for lower latency and thus higher performance of the underlying Visual SLAM algorithm.

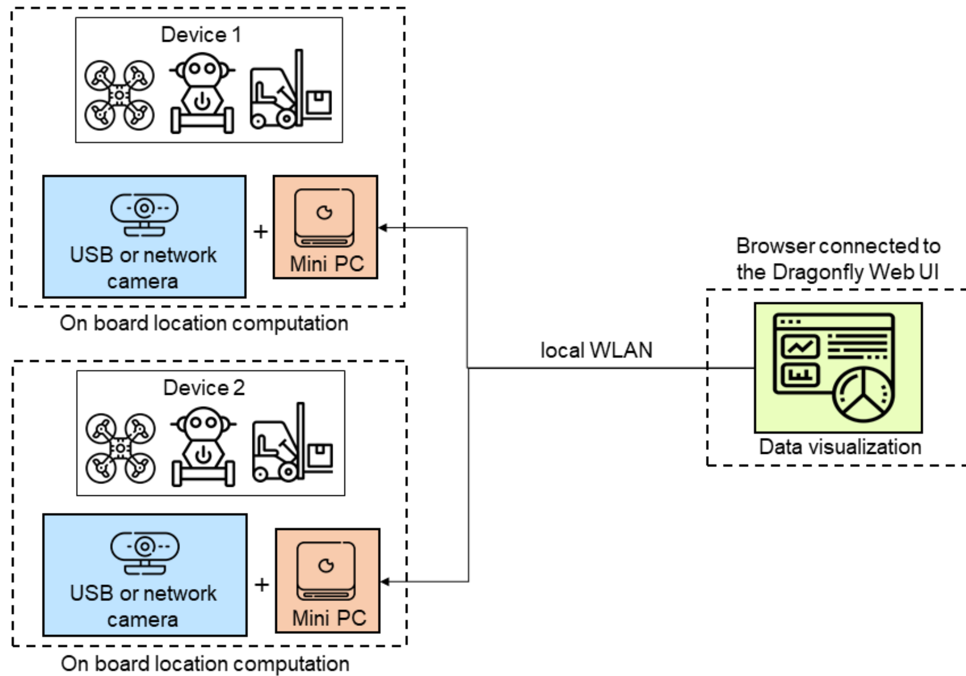


Figure 3.3: Dragonfly's on-board local computation architecture scheme [40].

Dragonfly's ROS integration is configured to provide the Autonomous Plus2 with a pose estimate including 2D position and orientation information (that is, 3 degrees of freedom). Alongside that information, an estimation of the covariance matrix associated to that estimate is provided by Kyburz's proprietary code on the vehicle. Being the incoming data already in such format, the only necessary step to process Dragonfly's data for the sensor rejection algorithms is to perform a transformation to the common reference frame, analogous to the one defined by Equations 3.1 and 3.2, but this time using $x_{DF}$, $y_{DF}$ and $\theta_{DF}$. As with Google Cartographer and unlike with GPS, the orientation data does not need a shift or change of sense to fit the common frame's $\theta$ definition (referenced to the $x$ axis and positive counterclockwise).

Analogously to the Cartographer case, the covariance matrix produced by Kyburz's proprietary code has been found to be unreliable, often providing unreasonably low values that can reduce the performance of some algorithms using covariance matrix information, so the fixed matrix in Equation 3.6 has been defined for Dragonfly, rounding the average values from gathered data, which are in physically reasonable ranges.

$$Q_{DF,fixed} = \begin{bmatrix} 0.02 & 0 & 0 \\ 0 & 0.02 & 0 \\ 0 & 0 & 0.001 \end{bmatrix} \tag{3.6}$$

## 3.4 Dead reckoning

Dead reckoning consists in calculating the current position of the vehicle by using a previously determined position to which estimations of speed and heading direction are incorporated. For this work, the Autonomous Plus2 provides speed information obtained from its wheel odometry and heading information from its IMU.

Given the inner workings of Kyburz's proprietary code on the vehicle, wheel odometry and heading information are published on the ROS pipeline in the form of a combined message with both information. This message contains a heading value $H_t$ and a displacement value $d_t$, the latter corresponding to the total advanced distance in the vehicle's forward direction. Internally, those two values are computed according to Equations 3.7 and 3.8, where the superscript *meas* indicates measured values and $b$ is the length of the wheelbase. The Autonomous Plus2's INS provides the value of $H^{meas}$ at each time instant, while its wheel odometry provides the value of $v^{meas}$. The value of the steering angle $\alpha$ is also measured.

$$H_t = H_{t-1}^{meas} + \hat{\omega}_{t-1} \cdot \Delta t, \quad \hat{\omega}_{t-1} = \frac{v_{t-1}^{meas} \cdot tan(\alpha_{t-1}^{meas})}{b} \tag{3.7}$$

$$d_t = d_{t-1} + v_t^{meas} \cdot \Delta t \tag{3.8}$$

In order to convert the values $H_t$ and $d_t$ into the format defined in Figure 1.2, first the heading $H$ needs to be transformed into the right angle definition of $\theta_i$, analogously to the transformation in Equation 3.4. Once the value of $\theta_{DR}$ is computed, the last remaining processing step is to perform a non-static transformation to the common zero-origin frame, following Equations 3.9 and 3.10. Notice that the latter transformation differs from the one in Equation 3.1, due to the nature of dead reckoning (i.e., incrementally adding to $x$ and $y$ the corresponding part of the forward displacement of the vehicle). As a consequence of this, as already commented in Chapter 2 when discussing wheel odometry, dead reckoning is subject to position error accumulation and progressive drift. The presented formulation allows, for instance, to reset the dead reckoning position (e.g.: by setting $x'_{DR,t-1}$ and $y'_{DR,t-1}$ to the last fused position estimate) when the sensor rejection scheme has labeled dead reckoning as non-reliable for a certain amount of time, and thus recover dead reckoning from an accumulated error that would otherwise not be undone (unlike in SLAM approaches or GPS, where recovery can happen automatically once the factors causing failure stop, as they are not based in numerical integration).

$$\theta'_{DR} = unwrap(\theta_{DR}) - \theta_{DR,0} \tag{3.9}$$

$$\begin{bmatrix} x'_{DR,t} \\ y'_{DR,t} \end{bmatrix} = \begin{bmatrix} x'_{DR,t-1} \\ y'_{DR,t-1} \end{bmatrix} + \begin{bmatrix} cos(\theta'_{DR,t}) & -sin(\theta'_{DR,t}) \\ sin(\theta'_{DR,t}) & cos(\theta'_{DR,t}) \end{bmatrix} \cdot \begin{bmatrix} \Delta d \\ 0 \end{bmatrix}, \quad \Delta d = d_t - d_{t-1} \tag{3.10}$$

Analogously to the GPS/INS case, as no covariance matrix is produced by Kyburz's proprietary code, fixed values are chosen for dead reckoning's covariance matrix (taking into account orders of magnitude of the errors and comparative errors between the different sources), which can be seen in Equation 3.11.

$$Q_{DR,fixed} = \begin{bmatrix} 0.2 & 0 & 0 \\ 0 & 0.2 & 0 \\ 0 & 0 & 0.001 \end{bmatrix} \tag{3.11}$$

## 3.5 ROS architecture and pipeline

Many of the vehicle's systems, especially sensors and localization algorithms, are connected through a ROS infrastructure. The Robot Operating System (ROS) is an open-source, meta-operating system for robotic systems [41], which provides hardware abstraction, low-level device control, message-passing between processes, etc. It is not a real-time framework, although it is possible to integrate ROS with real-time code. ROS is based on a publisher-subscriber paradigm, and is one of (if not the most) widely-spread frameworks in the field of robotics (especially in academia).

The ROS infrastructure available in the Autonomous Plus2 is very wide and offers many functionalities, but here the focus is put on the part of the overall pipeline that is relevant for this thesis. Figure 3.4 summarizes the main topics and nodes involved in this work. For the purpose of solving the problem stated in Chapter 1, one ROS node has been developed for each of the different proposed solutions described in Chapter 4, together with a ROS node that does all data pre-processing needed by the algorithms, to avoid repeating those steps in each solution node. Each of the ROS nodes are developed in Python and ROS Melodic. The data processing node subscribes to available topics with data steams from the different sensors/algorithms on the vehicle, and publishes a topic with a custom message containing the processed data and a time stamp. The solution nodes subscribe to this topic, and publish a topic with a custom message containing the sensor selection signal (see Figure 1.2) and, depending on the algorithm, also the sensor trust signal, time-stamped again.

Finally, Table 3.1 summarizes relevant inputs and outputs of the algorithms presented in this work, at a data level, each of which may be sent through different messages (see Figure 3.4), all of them time-stamped by ROS by default. Algorithm 3.1 summarizes all the data processing pipeline from data streams already available on the vehicle's ROS architecture to $(x'_i, y'_i, \theta'_i)$ pose estimates in the common reference frame, format that suits the input definition for our problem in Figure 1.2.

Table 3.1: Summary of inputs from the different sources considered in this work and outputs from the presented algorithms, all at a data level.

| I/O | Source | Data | Format(s) |
|---|---|---|---|
| Input | Cartographer | $x_{CG}$, $y_{CG}$, $\theta_{CG}$, $Q_{CG}$ | float64, float64[9] |
| | GPS/INS | $Lat$, $Long$ | float64 (all) |
| | | $Heading$ | float64 |
| | Dragonfly | $x_{DF}$, $y_{DF}$, $\theta_{DF}$, $Q_{DF}$ | float64, float64[9] |
| | Dead reckoning | $\Delta d$, $H_t$ | float64 (all) |
| Output | | $selection$ | bool[4] |
| | (only some algorithms) | $trust$ | float64[4] |
| | | $x'_{CG}$, $y'_{CG}$, $\theta'_{CG}$, $Q_{CG}$ | float64, float64[9] |
| | | $x'_{GPS}$, $y'_{GPS}$, $\theta'_{GPS}$, $Q_{GPS}$ | float64, float64[9] |
| | | $x'_{DF}$, $y'_{DF}$, $\theta'_{DF}$, $Q_{DF}$ | float64, float64[9] |
| | | $x'_{DR}$, $y'_{DR}$, $\theta'_{DR}$, $Q_{DR}$ | float64, float64[9] |

**Algorithm 3.1** Processing of available ROS data streams

**function** DATASTREAMPROCESSING()
  Retrieve $x_{CG}$, $y_{CG}$, $\theta_{CG}$ from *cartographer/tracked_pose* topic
  $x'_{CG}$, $y'_{CG}$, $\theta'_{CG} \leftarrow$ TRANSFORM_FRAMES($x_{CG}$, $y_{CG}$, $\theta_{CG}$) ▷ Equations 3.1, 3.2
  $Q_{CG} \leftarrow Q_{CG,fixed}$ ▷ Equation 3.3

  Retrieve $Lat$, $Long$ from *gps* topic
  Retrieve $Heading$ from *ins_raw/state* topic
  $x_{GPS}$, $y_{GPS} \leftarrow$ UTM_PROJECTION($Lat$, $Long$) ▷ Transformation details in [39]
  $\theta_{GPS} \leftarrow$ HEADING_TO_THETA($Heading$) ▷ Equation 3.4
  $x'_{GPS}$, $y'_{GPS}$, $\theta'_{GPS} \leftarrow$ TRANSFORM_FRAMES($x_{GPS}$, $y_{GPS}$, $\theta_{GPS}$) ▷ Equations 3.1, 3.2
  $Q_{GPS} \leftarrow Q_{GPS,fixed}$ ▷ Equation 3.5

  Retrieve $x_{DF}$, $y_{DF}$, $\theta_{DF}$ from *dragonfly_manager/tracked_pose* topic
  $x'_{DF}$, $y'_{DF}$, $\theta'_{DF} \leftarrow$ TRANSFORM_FRAMES($x_{DF}$, $y_{DF}$, $\theta_{DF}$) ▷ Equations 3.1, 3.2
  $Q_{DF} \leftarrow Q_{DF,fixed}$ ▷ Equation 3.6

  $d_{prev}, x'_{prev}, y'_{prev} \leftarrow 0$ ▷ Initialization
  Retrieve $H_t$, $d_t$ from *odom* topic
  $\theta_{DR} \leftarrow$ HEADING_TO_THETA($H_t$) ▷ Equation 3.4
  $\Delta d \leftarrow d_t - d_{prev}$
  $d_{prev} \leftarrow d_t$
  $x'_{DR}, y'_{DR}, \theta'_{DR} \leftarrow$ TRANSFORM_FRAMES2($\theta_{DR}$, $\Delta d$, $x'_{prev}$, $y'_{prev}$) ▷ Equations 3.9, 3.10
  $x'_{prev} \leftarrow x'_{DR}$
  $y'_{prev} \leftarrow y'_{DR}$
  $Q_{DR} \leftarrow Q_{DR,fixed}$ ▷ Equation 3.11

  **return** $x'_i$, $y'_i$, $\theta'_i$, $Q_i \quad \forall i \in \{CG, GPS, DF, DR\}$
**end function**

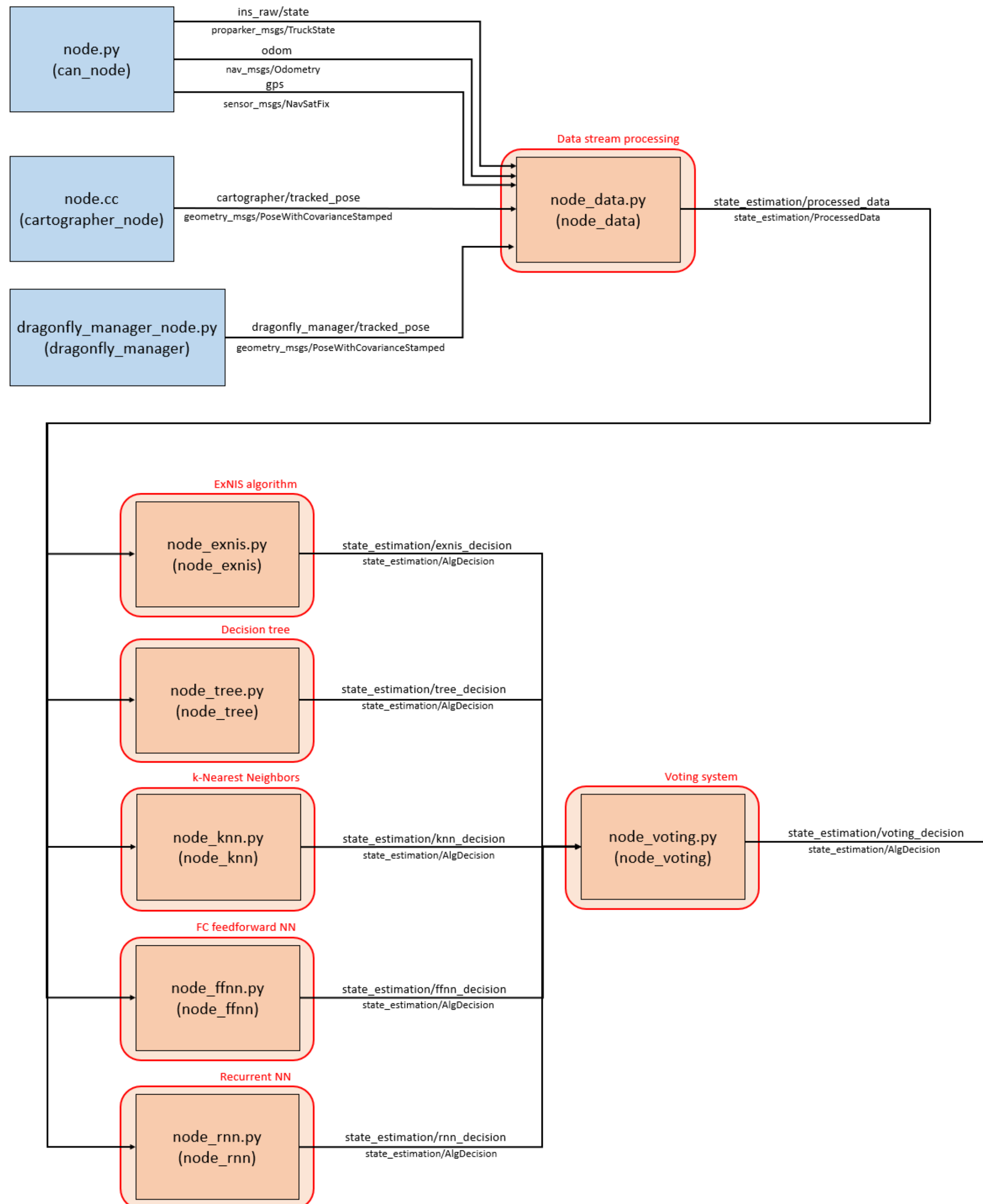


Figure 3.4: Part of the Autonomous Plus2's ROS pipeline that is relevant for this project's solutions. In blue, nodes already available in the vehicle; in orange, nodes developed in this work (one main node for each different approach, plus another one for data pre-processing); in black, different topics used to communicate between nodes. For each node, its associated file name is provided, together with its ROS node name in parentheses.

# Chapter 4

# Sensor rejection and trust estimation

The main object of this thesis is sensor rejection of 4 different pose estimate sources with flexible algorithms that do not restrict their applicability to a particular set of sensors/sources. Here, the techniques explored in this thesis are presented, some of them coming from related literature, and others being novel proposals or application of existing algorithms to a different problem. Techniques for solving this problem can be split into three big categories/paradigms:

- Manual heuristics (rule-based)
- Fault Detection and Isolation (FDI) methods
- Learning-based approaches

## 4.1 Manual heuristics (rule-based)

Manually designing a set of rules is one of the simplest approaches for sensor rejection. The crafting of such rules can be done through the use of theoretical knowledge of the system's elements, practical knowledge from experienced technicians, and/or analysis of recorded data from the vehicle and insight extraction from it. Such rules can come in many forms (thresholding the differences between each source's $x_i$, rejecting sensors that indicate heavy position variations when wheel speed readings are close to 0, etc.) and imply hand-crafting conditional checks that can validate or reject a certain hypothesis about a sensor's reliability.

The process of building such manual heuristics is usually not standardized or systematic, and an "only hand-crafted" rule set may miss important scenarios or fail to generalize to new scenarios (and can also be hard to migrate to a different platform, as the rules can end up being very vehicle-specific). Also, as discussed later in Section 4.3, there are learning based methods like decision trees that can automatically build simple and/or complex rule-based models from representative training data, which can directly look for good heuristics based on input data, and which can be manually aided by feature engineering techniques. Nevertheless, manual rules can be useful to aid other more systematic approaches, adding additional conditions and last-resort checks to make them more reliable in very specific situations where a particular algorithm may fail to perform well due to its inherent structure and properties.

For these reasons, in this work no "only manual" rule-based system is used (although rule-based systems are discussed through learning approaches), but instead additional hand-crafted rules (e.g.: last-resort sensor) and other hand-crafted algorithms (e.g.: multi-level multi-signal thresholding for

more flexibility) are proposed in the rest of sections from this chapter to be used along with other systematic approaches.

## 4.2 Fault Detection and Isolation methods

Fault Detection and Isolation (FDI) is a subfield of control engineering that concerns itself with monitoring a system, identifying when a fault has occurred, and pinpointing the type of fault and its location [42]. This thesis' problem (see Figure 1.2) can be seen as an FDI problem, where information coming from sensors is monitored to detect when some sensor becomes "faulty" (not reliable for localization purposes) and to be able to reject the information coming from faulty sensors (and only from faulty sensors), before it corrupts the running fused pose estimate.

There have been a vast amount of FDI methods proposed and applied successfully across literature [43]. Due to the nature of this thesis' problem and the desire to focus on generalist methods (which can, for instance, be used with completely different sets of sensors/algorithms, or even in a completely different vehicle), the parity-relation based FDI method proposed in [7] is explored. In parity-based FDI methods, a set of residual signals or "parity relations" are derived, whose evolution is affected by system faults. By monitoring those parity relations over time, faults may be detected and isolated. Some model or set of assumptions is needed regarding how faults affect those residuals.

Usually, residuals are designed such that each fault triggers a different response on residuals, so that faults can not only be detected, but also isolated (i.e., recognize which fault has occurred). For this thesis' problem, although several failure modes have been defined for each sensor in Chapter 2, they have all been considered the same fault in the application of the FDI method in [7] (i.e., all failure modes associated to Dragonfly trigger the same "Dragonfly fault"). This formulation suits the output definition in Figure 1.2, where the selection signal is the binary complement of the "fault signal", which has one binary value for each sensor that is set to 1 when a fault on that sensor occurs (regardless of which particular failure mode the sensor may have fallen into).

### 4.2.1 Extended NIS sensor validation

The approach proposed in [7] generalizes the NIS test for multiple measurements validation. In the NIS test [44], when a sensor measurement is available, as the true state is not known, the innovation between the observation data $s_t$ and the predicted state $\hat{s}_t$ is assumed to obey a Gaussian distribution with covariance $I_v^t$ and is used to verify the coherence of the measurement with the prediction. In particular, Equation 4.1 defines the difference $vs_t$, Equation 4.2 computes the covariance matrix of $vs_t$ (where $\hat{P}_{(t|t-1)}$ is the covariance of the prediction, $Q_t$ is the covariance of measurement and $H_t$ is the measurement model), and Equation 4.3 computes the Mahalanobis distance $d_M$ between observed measurement and predicted state. Under the uncorrelated Gaussian process assumption, $d_M$ follows a Chi-squared distribution of $m$ degrees of freedom, with $m$ being the dimension of the measurement vector. The NIS test rule rejects a measurement if $d_M$ doesn't lie within the confidence region defined by the corresponding $\chi^2(m)$ distribution.

$$vs_t = s_t - \hat{s}_t \tag{4.1}$$

$$I_v^t = H_t \cdot \hat{P}_{(t|t-1)}^{-1} \cdot H_t^{-1} + Q_t \tag{4.2}$$

$$d_M = vs_t^T \cdot (I_v^t)^{-1} \cdot vs_t \tag{4.3}$$

The extended NIS (ExNIS) sensor validation method proposed in [7] allows to check the coherence between the measurements coming from multiple sensors (also including the process model, if

used, whose prediction is considered as a virtual sensor observation and might be rejected if not coherent with the other sensors' observations). In particular, a parity relation is calculated for every sensor pair, yielding a set of parity relations associated to each sensor. For this thesis' problem, for instance, Cartographer has 3 associated parity relations, originating from separately comparing its measurements to the measurements from GPS/INS, Dragonfly and Dead reckoning, respectively. Whenever two sensors are not coherent with each other, their associated parity relation rises. The authors of [7] apply this approach only to single-sensor fault scenarios, as in those cases there is a single set of parity relations that rise at the same time (i.e., all the parity relations associated to the faulty sensor rise, while the rest remain stable). Nonetheless, the ExNIS approach can also be applied to multi-sensor fault scenarios, as long as at least 2 sensors remain non-faulty. This further extension hereby proposed implies, however, that in scenarios where no pair of sensors are coherent with each other, a last-resort decision rule has to be provided, which can't rely on sensor cross-validation, in order to still select one source for pose estimation. This last situation, however, is unlikely to happen if the autonomous vehicle has enough independent and uncorrelated pose estimate sources (like in the case of the Autonomous Plus2, where 4 are available), as even in very challenging scenarios usually a couple of sensors still can function reliably (e.g.: if the vehicle is inside a building and the lights go off, GPS/INS and Dragonfly won't be reliable, but Cartographer and Dead reckoning will still work to survive such worst-case scenario).

Equation 4.4 computes $vs_t^{ij}$, which denotes the difference between the measurements of sensor $i$ and sensor $j$ at time instant $t$. It is important to note that, for this work, all sources provide a full state vector $(x_i, y_i, \theta_i)$, so $vs_t^{ij}$ always is a 3-dimensional vector with all state components. As proposed in [7], in case some of the sources/sensors only provide a part of the state vector, the same equations can be used, using only the state dimensions that both sensors in a sensor pair share. If two sensors don't share a single state dimension (i.e., they do not provide redundant information), no parity relation can be associated to their pair. Equation 4.5 computes the covariance matrix $I_v^t$ associated to each $vs_t^{ij}$ (using the corresponding measurement covariances, $Q_k^t$), and Equation 4.6 computes the parity relation associated to the sensor pair $i, j$. Note that, by definition, all parity relations are non-negative quantities, and that $d_{ij} = d_{ji}$.

$$vs_t^{ij} = s_t^i - s_t^j \tag{4.4}$$

$$I_v^t = Q_i^t + Q_j^t \tag{4.5}$$

$$d_{ij} = (vs_t^{ij})^T \cdot (I_v^t)^{-1} \cdot (vs_t^{ij}) \tag{4.6}$$

Once the parity relations have been computed, parity checking methods can be applied to detect sensor faults. The main assumption of this approach is that whenever a sensor enters failure mode, all parity relations associated to that sensor shall rise. In single-sensor fault scenarios, this assumption has been proven to hold well [7]. In multi-sensor fault scenarios, this assumption may not hold if, for instance, 2 faulty sensors start drifting in the same way, and thus their associated parity relation does not rise. Such scenarios are highly unlikely if the input data used by different sensors/sources is uncorrelated (e.g.: camera and LiDAR data).

In order to check the evolution of parity relations for FDI, many change detection algorithms can be used, which can usually be recast into the problem of deciding between the following two hypotheses [45]:

$$H_0 : E(s_t) = 0,\ H_1 : E(s_t) > 0 \tag{4.7}$$

In order to solve this problem, we need a tool to decide whether a result is significant or not, which is sometimes called a "stopping rule". A stopping rule can be achieved by:

1. Low-pass filtering $s_t$ (to smooth instantaneous fluctuations and make the decisions more stable and less sensitive to noise).

2. Comparing the filtered value to a threshold.

Two well-known computationally cheap and simple low-pass filtering techniques are Cumulative Sum (CUSUM) and Exponentially Weighted Averages (EWA). Equation 4.8 computes the CUSUM filtered signal $g_t$ from the original signal $s_t$ (the parity relation in this case). The *drift parameter* $\nu \in \mathbb{R}$ allows to tune the low-pass effect (which has a significant impact on each algorithm's performance in terms of fault detection speed). Equation 4.9 computes the EWA filtered signal instead, where the *forgetting factor* $\beta \in [0, 1]$ allows to tune the low-pass effect.

$$g_t = max(g_{t-1} + s_t - \nu, 0) \tag{4.8}$$

$$g_t = \beta \cdot g_{t-1} + (1 - \beta) \cdot s_t \tag{4.9}$$

Equation 4.10 gives an approximate relation between parameter $\beta$ and the number of time steps taken into account for the weighted average. Note that setting a value of $\beta = 0$ is equivalent to not filtering. Upon initialization, the EWA filtered signal takes some steps to reach the signal's average value, yielding a biased result in the first steps. Equation 4.11 applies a bias correction, which solves this problem and allows for better performance in the first steps. Note that the effect of this correction vanishes significantly with time, as $0 \leqslant \beta \leqslant 1$. Details on the tuning procedure for all parameters appearing in this chapter are discussed in Chapter 5.

$$time\ steps \approx \frac{1}{1 - \beta} \tag{4.10}$$

$$g_t^{corr} = \frac{g_t}{1 - \beta^t} \tag{4.11}$$

Regarding thresholding techniques, once each parity relation has been low-pass filtered, each sensor has 3 signals to threshold and reach a final decision on whether "all the parity relations associated to that sensor have risen" or not. In [7] the way this decision is taken is not specifically described, so here two approaches are proposed:

- Independent 1-level thresholding on all 3 signals separately.
- Combined 3-level thresholding on all 3 signals together.

The first approach consists in applying the same single threshold to all 3 signals associated to a sensor, separately. Once all 3 signals exceed the threshold, the sensor associated to those signals is considered faulty. In other words, a sensor is considered non-faulty as long as there is at least 1 signal associated to it that lies inside the range defined by the threshold. This simple approach only has 1 parameter to tune, the threshold $th$, whose candidate values can be taken from a $\chi^2(3)$ distribution (the measurement vector has dimension 3). This way, for instance, applying this approach with $th = 7.815$ would be roughly equivalent to considering that a sensor is faulty if its "distances" to the rest of the sensors are all less than 5% likely to occur under non-faulty circumstances.

The second approach (novel for this problem, to the best of the writer's knowledge) consists in applying 3 different thresholds to all 3 signals associated to a sensor, at the same time. A sensor is considered non-faulty if at least 1 of the following 3 conditions are met:

- 1 or more signals lie inside the range defined by $th_1$.
- 2 or more signals lie inside the range defined by $th_2$.
- All 3 signals lie inside the range defined by $th_3$.

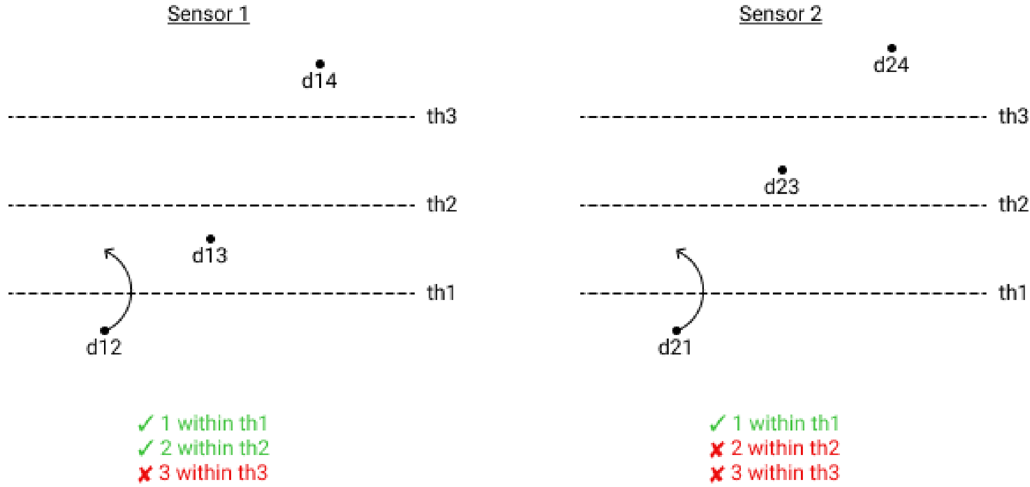


Figure 4.1: Particular scenario where the combined 3-level thresholding approach allows for additional flexibility, not possible with the first approach. In green and red, respectively, fulfilled and non-fulfilled conditions before the change described by the arrow.

Here, normally $th_1 < th_2 < th_3$, although some degenerate cases can be explored by setting the values of $th_i$ so that they don't follow this inequality. In particular, if $th_2$ is chosen such that $th_2 \leqslant th_1$, its associated condition becomes redundant and the 3-level thresholding collapses into 2-level thresholding. Same occurs when setting $th_3$ is chosen such that $th_3 \leqslant th_2$ and/or $th_3 \leqslant th_1$. If both $th_2 \leqslant th_1$ and $th_3 \leqslant th_1$, the approach collapses into the first approach, with $th = th_1$. For this reason, this second approach can be considered a generalization of the first approach, with more flexibility (it allows for 1-level, 2-level and 3-level thresholding) and several rejection conditions at the same time.

While the first approach only has 1 parameter, $th$, producing fairly simple rules (e.g.: reject when all signals have values less than 5% likely to occur), this second approach has 3 parameters, $th_1$, $th_2$, and $th_3$, whose values can be chosen to reach more refined rules (e.g.: if one of the signals is very unlikely to occur, reject unless either the 2 remaining are in a low level, or one of them is in a very low level). Another worth noting consequence of this flexibility is the possibility of situations like the one in Figure 4.1 when one sensor in a pair is supported by 2 validation conditions (1st condition on $d_{12}$, and 2nd condition on $d_{12}$ and $d_{13}$), while the other sensor only by 1 (1st condition on $d_{21} = d_{12}$). If their associated parity relation rises above $th_1$, sensor 1 would still be considered non-faulty, while sensor 2 wouldn't. With the first approach, both sensor would have to be rejected in this situation, not allowing sensor 1 to have a second validation condition.

Finally, in order to summarize all approaches presented in this section, pseudocode for each of them are presented in Algorithms 4.1, 4.2 and 4.3. Note that at the end of Algorithm 4.1 an additional last-resort step is proposed. Due to ExNIS being based on cross-validation of sources, in unlikely cases where 3 out of 4 sources fail, the selection signal becomes a vector of zeros, even if the remaining source is very reliable. Adding a last-resort step allows enforcing individual trust on the sensor that is most likely to be reliable in such "3/4 situations". In the case of the Autonomous Plus2, Cartographer is chosen as the last-resort source, as it has shown to be the least prone-to-failing source by far.

## 4.3 Learning-based approaches

Statistical learning theory is a framework for machine learning (ML) arising from statistics and functional analysis that deals with the problem of finding a predictive function (sometime also

**Algorithm 4.1** Extended NIS-based sensor validation [7]

**function** ExNIS($x_k$, $y_k$, $\theta_k$, $Q_k$)
  $s_k := \left[x_k, y_k, \theta_k\right]^T$ ▷ Measurement vectors
  **for** $i \leftarrow 1, \ldots, n_{sources}$ **do**
    **for** $j \leftarrow 1, \ldots, n_{sources},\ j \neq i$ **do**
      $vs_{ij} \leftarrow s_i - s_j$
      $I_v \leftarrow Q_i + Q_j$
      $d_{ij} \leftarrow (vs_{ij})^T \cdot (I_v)^{-1} \cdot (vs_{ij})$ ▷ Parity relation associated to the $i, j$ pair
      $g_{ij} \leftarrow$ LowPassFilter($d_{ij}$) ▷ Either CUSUM or EWA
    **end for**
  **end for**
  $selection \leftarrow$ Thresholding($g_{ij}$) ▷ Either 1-level or 3-level approaches
  **if** $selection = \left[0, \ldots, 0\right]$ **then** ▷ Last-resort mechanism if 3/4 sources fail
    $selection_{CG} \leftarrow 1$
  **end if**
  **return** $selection$
**end function**

**Algorithm 4.2** Low pass filtering approaches

**function** CUSUM($\vec{s}$, $\nu$) ▷ Applied one step at a time
  $\vec{g} \leftarrow \left[0, \ldots, 0\right]$ ▷ Initialization
  $g_{prev} \leftarrow 0$
  **for** $t \leftarrow 0, \ldots, t_f$ **do**
    $g_t \leftarrow max(g_{prev} + s_t - \nu, 0)$ ▷ Filtering rule
    $g_{prev} \leftarrow g_t$
  **end for**
  **return** $\vec{g}$
**end function**

**function** EWA($\vec{s}$, $\beta$) ▷ Applied one step at a time
  $\vec{g}^{corr} \leftarrow \left[0, \ldots, 0\right]$ ▷ Initialization
  $g_{prev} \leftarrow 0$
  **for** $t \leftarrow 0, \ldots, t_f$ **do**
    $g_{new} \leftarrow \beta \cdot g_{prev} + (1 - \beta) \cdot s_t$ ▷ Filtering rule
    $g_{prev} \leftarrow g_{new}$
    $g_t^{corr} \leftarrow g_{new} / (1 - \beta^t)$ ▷ Bias correction for first steps
  **end for**
  **return** $\vec{g}^{corr}$
**end function**

**Algorithm 4.3** Multi-signal thresholding approaches

```
function IndependentOneLevelTh(g_t, th)
    selection ← [0, ..., 0]
    for i ← 1, ..., n_sources do
        for j ← 1, ..., n_sources, j ≠ i do
            if g_ij ⩽ th then        ▷ Parity relations for sensor i are thresholded independently
                selection_i ← 1
                break
            end if
        end for
    end for
    return selection
end function

function CombinedThreeLevelTh(g_t, th_1, th_2, th_3)
    selection ← [0, ..., 0]
    for i ← 1, ..., n_sources do
        if count(g_ij ⩽ th_1, ∀j ≠ i) ⩾ 1    ▷ Parity relations for sensor i are thresholded together
        or count(g_ij ⩽ th_2, ∀j ≠ i) ⩾ 2
        or count(g_ij ⩽ th_3, ∀j ≠ i) ⩾ 3 then
            selection_i ← 1
        end if
    end for
    return selection
end function
```

called an input-output "mapping") based on data. It is commonly used for many tasks in the field of autonomous vehicles [46], like perception, motion planning, localization, etc. This thesis' problem can be approached with learning techniques by either considering it an FDI problem (monitoring sensor measurement to label sensors as faulty or non-faulty) and using learning-based FDI techniques, which output binary information about the sensors, or taking advantage of the probabilistic nature of some learning-based algorithms to not only produce binary sensor state information (corresponding to the selection signal in Figure 1.2), but also generate some probabilistic non-binary "trust" information about each sensor (e.g.: a reliability score ranging from 0 to 1, where a value of 0.7 indicates that a sensor is 70% likely to be reliable). This additional information can be valuable both for later sensor fusion and/or for other decision tasks the autonomous vehicle may need to perform. Therefore, learning-based approaches capable of producing such probabilistic information are considered of great interest for this thesis and explored in greater detail.

There is a wide variety of statistical/machine learning algorithms that have been proposed and applied across the literature, both for autonomous driving-related and non-related tasks. These algorithms are usually classified into 3 big paradigms:

- **Unsupervised learning:** The algorithm learns patterns directly from unlabeled data.
- **Supervised learning:** The algorithm learns an input-output mapping through labeled data (input-output examples).
- **Reinforcement learning:** The algorithm (usually called an "agent") learns how to take actions in a certain environment to maximize a "cumulative reward".

This thesis' problem fits well into the second paradigm, supervised learning, as there is an input-output mapping to be learned (where pose estimates from different sources are the inputs and the sensor selection and trust signals are the outputs) and input-output examples (labeled data) can be obtained for the problem (as discussed in Chapter 5). Supervised learning approaches have proven

very successful in learning highly nonlinear and complex input-output mappings, especially with the recent works in the field of Deep Learning. However, this high flexibility comes with a cost: there is a considerable need for (labeled) data, which increases with model's and problem's complexity, in order to learn a map that can both fit the input-output examples and properly generalize to new, never-seen-before examples (which is the ultimate goal of the practical application of these techniques).

This ulterior need for labeled data is one of the main differences between the methods described in this section and methods like ExNIS. Labeled data is very useful for assessing any algorithm's performance with quantitative metrics, and can also be used in ExNIS for systematically tuning its parameters (e.g.: thresholds), as described in Chapter 5, but approaches like ExNIS can be deployed without using any labeled data, manually tuning parameters by either trial-and-error or manual inspection of unlabeled data and intuition from an experienced technician. In the case of supervised learning algorithms, this is almost never a possibility, as model complexity, size (i.e. number of parameters) and "close-to-black-box" structure makes it impractical to manually tune them. Therefore, relevant labeled data has been collected for this thesis following the approach described in Chapter 5.

In order to use supervised learning approaches, the inputs chosen for the algorithms can have a huge impact on their performance, as they are the only information used to predict the outputs. There are two main approaches to define inputs for supervised learning approaches applied to this thesis' problem: directly using the $(x_i', y_i', \theta_i')$ pose estimates, or manually extracting "features" (alternative representations, usually of different dimension, of the input pose estimates obtained by a fixed "hand-crafted" transformation) from those pose estimates and using them as inputs to the supervised learning algorithms. There are many techniques to hand-craft features to allow for better algorithm performance (e.g.: subtracting $x_i$ from different sensors to create "$x$ distance features", computing the square of signals, using numerical derivatives or integrals, etc.). This process is usually called "feature engineering" and it requires practical experience, as well as some trial-and-error (each problem and algorithm may have completely different optimal input features). Because optimal feature engineering is usually non-intuitive and non-systematic to tackle, and due to the recent advancements in Deep Learning, this feature hand-crafting process is being progressively substituted by the use of big and complex models that can learn those feature transformations internally when trained with enough data, in exchange for a less explainable, more "black-box" model. For this reason, in this work the focus is mostly put on models that can directly use pose estimates as inputs, avoiding manual feature engineering (which can also cause information simplification if the resulting feature vector is of lower dimension). Nevertheless, in order to still analyze the possibilities of manual feature engineering for this problem, and in order to help some smaller and less complex ML models, motivated by the previously presented ExNIS approach, the parity relations $d_{ij}$ defined in Equation 4.6 are considered as a manual feature alternative to direct pose estimate input for the supervised learning algorithms.

### 4.3.1 Classical ML classification methods

Supervised Machine Learning algorithms can tackle a wide variety of problems, which are usually classified into two big categories: prediction and classification. This thesis' problem falls into the classification category (in particular, binary classification). For such task, before the recent rise of Deep Learning, some of the most widely used ML algorithms were logistic regression, k-nearest neighbors, decision trees, support vector machines and naive Bayes. Among these options, in this work decision trees (DT) and k-nearest neighbors (k-NN) are explored, as they gave the best preliminary results for the particular problem in hand in a vanilla model training session where all the aforementioned algorithms were trained in different hyperparameter configurations and in slightly different versions of each algorithm, in a coarse search to discard the less promising options.

### Decision trees

A decision tree is a tree-structure classification model, where each internal node (also known as "decision node") represents a "test" on an attribute, each branch represents one of the possible outcomes of a test, and each end node (also known as "leaf node") represents a final label into which the sample is classified. A path from the root of the tree to a certain leaf represents a classification rule. Figure 4.2 presents an example of a very simple decision tree for classifying a person's gender.

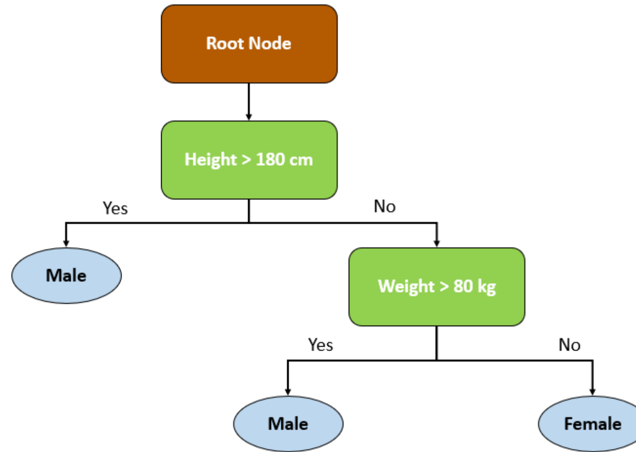


Figure 4.2: Example of a very simple decision tree for gender classification. In brown, the root node, where the inference starts; in green, decision nodes, where a certain input variable is used for branching; in blue, leaf nodes, corresponding to a male/female classification label.

Decision trees are usually easy to understand, visualize and interpret (even by non-expert users) and can be efficiently induced from data [47]. They are related to manual rule-based systems, but learn their decision rules (and their hierarchy) automatically from the training data, instead of being manually programmed. They also implicitly perform feature selection (feature importance is natively modelled by the hierarchy of decision nodes, and non-relevant variables end up not being used in them), and have fast run time. Nevertheless, decision trees are prone to overfitting (creating over-complex trees that do not generalize to new data), can be unstable (small variations in the training data can create very different trees, which can be reduced with techniques like *boosting*) and can create biased trees if some class dominates in unbalanced training data.

In order to make the prediction for each of the sources, two main different approaches can be followed:

- Build 4 trees, each predicting the class label (0/1) of one of the sources.
- Build 1 tree, which outputs a single class label representing the combination of labels for all sources. In this case, 16 different labels could be assigned, from 0 corresponding to all sources being rejected up to 15 corresponding to all sources being valid.

After preliminary testing of each option, here one tree is built for each of the 4 sources. The single tree approach is more difficult to train (it needs significant data from all 16 possible combinations) and is prone to mistaking cases with only 1 source labelled differently. These two factors combined end up causing poor performance, and also make it more difficult to generalize to other vehicles, especially if the number of sources is higher than on the Autonomous Plus2 (the number of labels grows as $2^{n_{sources}}$).

The main parameter to tune when using decision trees is the maximum number of splits, which controls how many decision nodes a tree can have. This has a direct impact on the tree's complexity

and its proneness to overfitting. Trees with too few splits usually can't fit the training data, while trees with too many splits may not generalize well to new samples. Details on the tuning of this parameter are found in Chapter 5.

Once the model is trained, classifying a new sample with a decision tree only requires to evaluate decision nodes sequentially, following the branches associated to the outcomes of the tests for that particular sample.

#### k-Nearest Neighbors

K-Nearest Neighbors (k-NN) is a classification algorithm that relies on the assumption that similar things exist in proximity in the space defined by the input variables. A new example is assigned the most common class label between its $k$ nearest neighbors (see Figure 4.3), according to a distance function (e.g.: Euclidean distance) defined in the input space. It approximates the input-output mapping locally, and it defers all computations until evaluation of a new sample. Instead of explicitly learning a model, k-NN stores the training examples (also known as "instance-based learning" [48]), which are then used for inference with new samples.

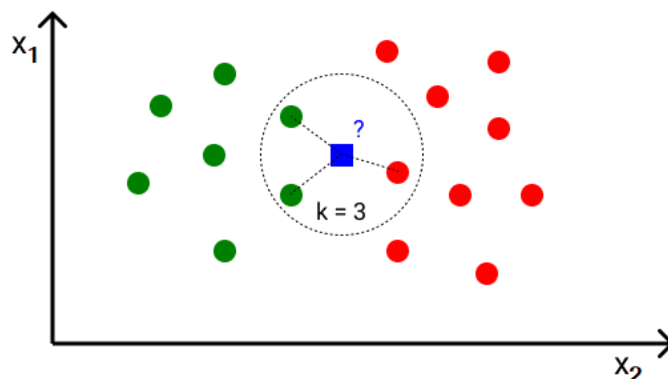


Figure 4.3: Example of k-nearest neighbors binary classification with $k = 3$. The new sample (in blue) is assigned the most common class between its 3 nearest neighbors (here, the green class).

K-NN is simple and easy to interpret, and doesn't require building and tune a model (although the value of $k$ can be tuned for optimal performance) or make additional assumptions. Nevertheless, k-NN requires higher computational times and uses more memory the bigger the training dataset is (and the bigger the dimensionality of the data), which poses a trade-off between performance and resource requirements.

Analogously to decision trees, there is the option to choose between building 4 different k-NN classifiers, each dedicated to 1 source, or a single k-NN classifier with non-binary labels. After preliminary testing of each option and for the same reasons discussed for decision trees, here one classifier is built for each of the 4 sources.

The main parameter to tune when using k-NN is the value of $k$, which controls how many neighbors are taken into account for each decision. Usually, $k$ is chosen to be an odd number to avoid tie situations in binary classification. Values of $k$ close to 1 yield less stable results, more prone to overfitting, as less training examples are considered to make the final decision. Higher value of $k$ yield more stable results, reducing overall noise in the training data, but too high values can cause underfitting and an increase in the number of classification errors.

In order to classify a new sample with k-NN, Algorithm 4.4 is used, where it can be seen that all the computational effort is concentrated in the inference step for new samples, as no model is previously learned.

**Algorithm 4.4** k-Nearest Neighbors

**function** k-NN($\vec{x}_{new}$) ▷ Query sample with $N$ features
  $\vec{x}_{train,i},\ y_i \leftarrow$ LoadTrainingData( ) ▷ Training data with $M$ samples, loaded once
  $\vec{d} \leftarrow [0, \ldots, 0]$
  **for** $i \leftarrow 0, \ldots, M$ **do**
    $d_i \leftarrow \sqrt{\sum_{k=1}^{N}(x_{new}^k - x_{train,i}^k)^2}$ ▷ Distance between query and $i$-th training sample
  **end for**
  $I \leftarrow \underset{(k\ best)}{arg\ min}\ d_i$
  $y_{new} \leftarrow$ MajorityLabel($y_{i \in I}$)
  **return** $y_{new}$
**end function**

### 4.3.2 Fully-connected feedforward neural networks

Artificial neural networks, sometimes simply called neural networks (NN), are a widely used supervised ML technique, based on a collection of connected units (often called neurons, in an analogy with animal brains) that can process input signals to generate an output signal, which is then sent to other units. These units are often aggregated into layers, which can be classified into the input layer (to which input data is fed), the output layer (which yields the final signal for the ML task) and intermediate layers (which have a certain number of units each and process signals from their previous layer to produce signals to be fed to their next layer). Neural networks with only 1 intermediate layer are usually called *shallow*, while networks with more than 1 intermediate layers are usually called *deep*, whose study is the object of the field of *Deep Learning* (DL).

Connections between neurons in different layers can be established in several ways. One of the most common is *fully-connected* (FC) *feed-forward* (FF) neural networks, an architecture such that all the units in one layer are (only) connected to all the units in the next layer (see Figure 4.4). Such architecture can give rise to very flexible and versatile networks that can learn highly non-linear mappings.

Fully-connected feedforward neural networks, however, are usually computationally intense and prone to overfitting (due to the high number of parameters associated to the high number of connections, especially in deeper networks), and have shown some limitations when used alone for dealing with image or time-series data. Nevertheless, they have been successfully applied in many research areas, significantly improving the results from classical supervised ML methods, especially as the amount of labeled training data has become substantially bigger in many application areas. In Figure 4.5, the dependence between the amount of available data and the maximum performance of learning algorithms is depicted, which explains the reason for the increasing interest in Deep Learning. Unlike most "traditional" supervised learning approaches, whose performance plateaus at a certain amount of available training data, DL approaches usually keep increasing their performance when trained with even more data. The potential performance of DL algorithms is also increased with the number of layers ("how deep the network is"), in exchange for a higher proneness to overfitting (and an associated bigger need for training data).

Feedforward neural networks can automatically deduce and optimally tune features for the desired task (i.e. they perform feature engineering without prior manual extraction), and they can learn inherent variability in data coming from that task. When using certain activation functions (e.g.: sigmoid function), their output provides probabilistic information, richer than the binary information from other classical ML algorithms (it can, for instance, be interpreted as a trust metric, which can both be used for sensor rejection and other tasks in the autonomous vehicle). However, they can require large amounts of data to perform significantly better than other ML techniques and avoid overfitting, there is no standard theory to guide the process of designing networks' architectures, and the resulting models are much less explainable (closer to the "black-box" modelling paradigm).

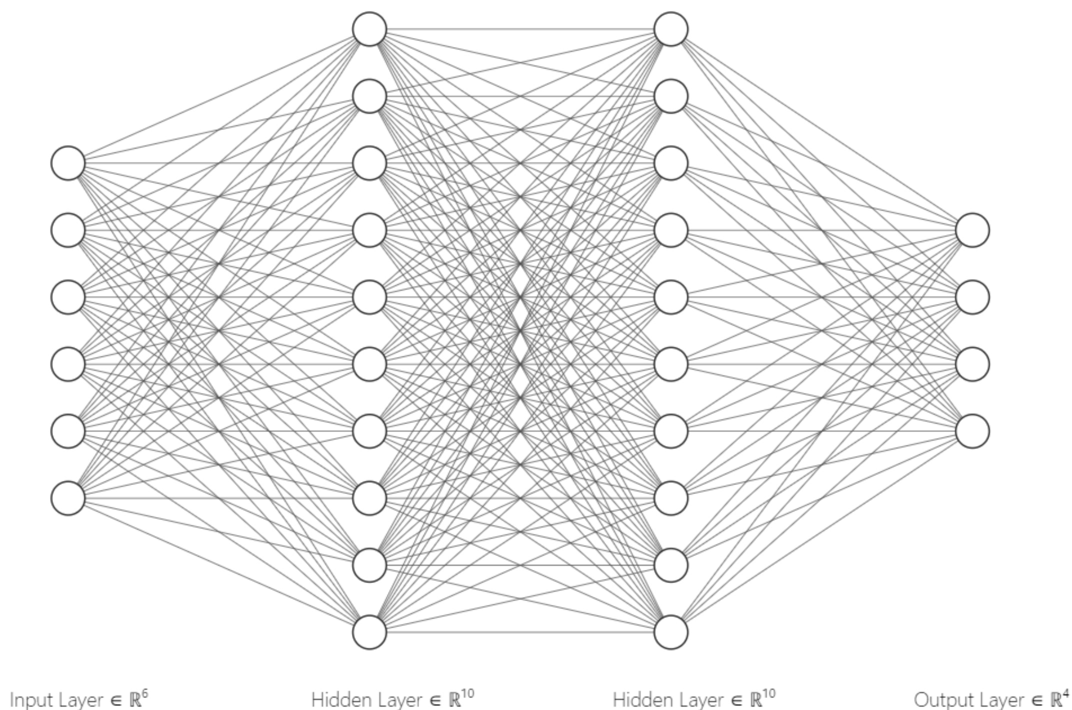


Figure 4.4: Example of a FC feedforward neural network with 6 inputs, 4 outputs and 2 intermediate layers, each with 10 units. Every unit of a certain layer is connected to all units in the next layer.

Feedforward neural networks have many architecture parameters that can be tuned, like the number of intermediate layers, the number of units per layer, the activation function used in each layer, the use of regularization (e.g.: L2, dropout, ...) and normalization techniques, all of which can have a big impact in the potential performance of the resulting NN. Many other hyperparameters are also linked to the training of the network, like learning rate (the most relevant in general), optimization algorithm, batch size, etc. Details on parameter tuning are found in Chapter 5.

Once the model is trained, classifying a new sample with a fully-connected neural network (either shallow or deep) simply requires to perform a forward propagation pass, feeding the inputs to the input layer, computing linear combinations and non-linear activation functions, and feeding the output signals to the next layers, until the output layer's activation is computed, which yields the output signals for the given input sample. For the particular problem of this thesis, both shallow and deep FF neural networks are discussed. A more detailed discussion on the output format of the network can be found in Chapter 6.

### 4.3.3 Sequence models

Standard feedforward neural networks, like the neural networks discussed before, do not share features across different positions of the network, assuming that all inputs and outputs are independent of each other. These models can have a hard time when working with sequence data (e.g.: time-series data), as with this kind of data previous inputs (and even outputs) are key for predicting the next output, capturing trends and time-associated information. Although some approaches exist to try to handle sequence data with FC neural networks (e.g.: storing the last $n$ samples of $x_t$ and including them all as part of the input vector), in order to directly deal with this limitation, sequence models can be used, whose structure is explicitly designed for supervised ML tasks that have sequence input data.

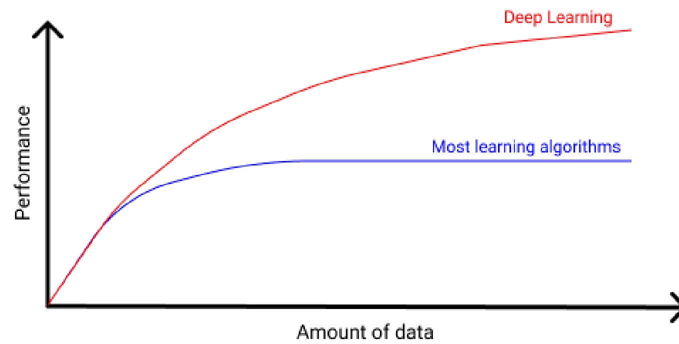


Figure 4.5: Maximum performance of traditional supervised learning and Deep Learning algorithms depending on the amount of available labeled training data.

For the particular case of this thesis' problem, the sensor rejection and trust estimation has so far mostly been tackled as a time-isolated decision problem, where only the $(x_i, y_i, \theta_i)$ estimates from the current time instant are taken into account (except when signal filtering techniques have been applied). However, this same problem can also be tackled by considering information from previous time instants too (i.e.: treating pose estimate streams as sequence data, instead of isolated instantaneous data), by using sequence models, which allow for a state-dependent decision scheme. In particular, in this work, the use of Recurrent Neural Networks (RNN) is explored.

#### Recurrent Neural Networks

Recurrent neural networks (RNN) are a type of artificial neural networks that exhibit temporal dynamic behavior, as the connections between units form a directed graph along a temporal sequence. They have an internal state (usually interpreted as a form of memory), which can be used to process variable length sequences of inputs. Figure 4.6 depicts a vanilla RNN architecture that has 1 recurrent fully-connected layer between the input and output layers.

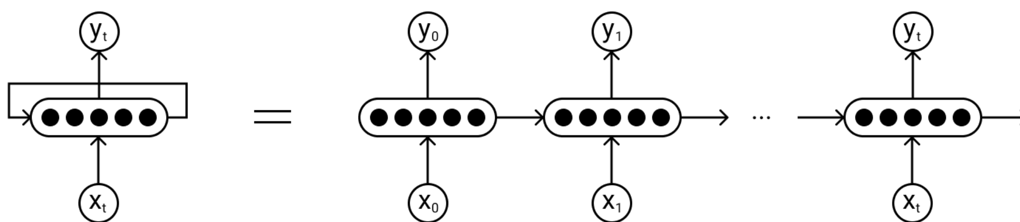

Figure 4.6: Example of two ways of schematically representing a vanilla RNN architecture.

Recurrent neural networks can process inputs of any length, their model size does not grow with input size (as the same weights are shared for any time step) and they can learn time dependencies through their internal state. However, recurrent layers can be computationally more demanding, training RNN models can be more difficult, and they are more prone to suffering from gradient exploding and/or vanishing. When long-term time dependencies need to be captured, vanilla RNN models may not perform well and modifications like long short-term memory (LSTM) models are needed.

Similarly to FC feed-forward neural networks, there are many parameters that can be tuned for RNN models, both regarding network architecture and training procedure, most of which are analogous to the ones presented for feedforward neural networks. Again, details on parameter tuning are found in Chapter 5.

Once the model is trained, a single pass through the net is done with every new sample, similarly to the case of FF NN models. The main difference, though, lies on the fact that RNN recurrent layers' activations from last time step (which conform the internal state or memory) are fed back as input to the same recurrent layers for the next time step (see Figure 4.6). For the particular problem of this thesis, vanilla RNN models are discussed, as LSTM models didn't show significant performance differences upon basic testing of different RNN model options (which may indicate the absence of hard-to-capture long-term dependencies that justify switching to more complex and heavier models).

## 4.4 Voting systems

Across this chapter, the following final approaches have been presented (whose tuning and results are discussed in later chapters):

- Extended NIS cross-validation
- Decision trees
- k-Nearest Neighbors
- Fully-connected feedforward neural networks
- Recurrent neural networks

For each of these approaches, a ROS node has been built to perform independent sensor rejection, yielding each a binary selection signal (and in some cases also a continuous trust signal). The training and tuning procedures for each algorithm to obtain the final weights and parameters is discussed in Chapter 5, and the individual results obtained by their final versions in several relevant test scenarios are discussed in Chapter 6. Nonetheless, in order to take a final decision on which sources to trust, two main approaches can be used:

- Only use the decision coming from the rejection algorithm that displayed the best performance in test scenarios according to some of the presented metrics.
- Build a voting system that takes into account the decision of each proposed method to come up with a final decision.

The first option would render all nodes but one useless, only needing the one associated to the "best-performing" algorithm, which can reduce computational load, in exchange for potentially worse overall performance (trade-offs usually exist, as some algorithms may perform better than others in different scenarios, capturing different failure modes).

Voting is also considered an ensemble ML algorithm. For classification, hard voting systems consider the final label assigned by each individual algorithm to the sample, while soft voting consider probabilistic information provided by each algorithm for the sample. Examples of the former are choosing the most voted label (most common simple approach), considering a source reliable only when at least a certain fraction of the algorithms label it as reliable (usually more conservative), etc. An example of the latter can be adding up the probabilities or trust scores from each algorithm and taking the label with the highest total score.

These approaches are very computationally cheap and can slightly increase the final system's reliability. In this work, as only some algorithms offer a continuous trust estimate, while all of them provide a binary selection signal, hard voting systems are the most natural choice. In particular, a simple majority voting system is explored, which provides a final sensor selection signal for each of the 4 pose estimate sources. No tie-breaking rule is needed, as the number of implemented algorithms is odd. In Chapter 6, results from individual algorithms and the voting system are compared.

Finally, Algorithm 4.5 summarizes the overall system proposed as the solution to this thesis' problem, from data streams available at system input, to individual selection signals (and trust information in some algorithms) and final selection signal available at system output. Note that an additional last-resort step is used at the end of the overall process, for cases where the voting system happens to label all sources as unreliable. Due to the nature of autonomous vehicles, the Autonomous Plus2 needs to rely on at least one source at every time to keep updated information on its current location, but in some situations the voting system may end up labelling all sources as unreliable. In those cases, a default sensor has to be chosen to keep the localization (as well as potentially sending a warning to the vehicle's logic systems about this situation), which for the Autonomous Plus2 is chosen to be Cartographer, as discussed before.

**Algorithm 4.5** Proposed overall system for sensor rejection and trust estimation

**procedure** SENSOR REJECTION AND TRUST ESTIMATION
  $x'_i$, $y'_i$, $\theta'_i$, $Q_i \leftarrow$ DATASTREAMPROCESSING() ▷ Algorithm 3.1
  $selection_I \leftarrow$ EXNIS($x'_i$, $y'_i$, $\theta'_i$, $Q_i$) ▷ Algorithm 4.1
  $selection_{II} \leftarrow$ DECISIONTREE($x'_i$, $y'_i$, $\theta'_i$)
  $selection_{III} \leftarrow$ K-NN($x'_i$, $y'_i$, $\theta'_i$) ▷ Algorithm 4.4
  $selection_{IV}, trust_{IV} \leftarrow$ FEEDFORWARDFCNN($x'_i$, $y'_i$, $\theta'_i$)
  $selection_V, trust_V \leftarrow$ RECURRENTNN($x'_i$, $y'_i$, $\theta'_i$)
  $selection_f \leftarrow$ VOTINGSYSTEM($selection_k$) ▷ Final selection signal
  **if** $selection_f = \left[0, \ldots, 0\right]$ **then** ▷ Last-resort mechanism if all sources are voted unreliable
    $selection_{f,CG} \leftarrow 1$
  **end if**
**end procedure**

# Chapter 5

# Experimental procedure

In Chapter 4, different approaches for sensor rejection have been presented, which need to be trained and/or tuned to ideally obtain their maximum potential performance. Also, once an optimal version of each approach is obtained, they can be compared and contrasted to determine which approach yields the best performance on the Autonomous Plus2, or to assess the existing trade-offs in case none of the options is a clear "winner".

Here, metrics for quantitatively assessing algorithm performance are presented, different real and relevant test scenarios are proposed, training data acquisition and processing are discussed, and algorithm training and/or tuning procedures are described. All of these aspects determine how the results in Chapter 6 are obtained.

## 5.1 Performance evaluation

### Performance metrics

There are many metrics that can be defined for classification tasks, and their relevance depends on the particular problem in hand. For this thesis, two metrics are the basis of algorithm performance evaluation:

- **True negative rate (TNR):** It is defined as the fraction of ground-truth negative samples that are labeled by the algorithm as negative (label 0).
- **True positive rate (TPR):** It is defined as the fraction of ground-truth positive samples that are labeled by the algorithm as positive (label 1).

In particular, TNR represents the fraction of times when our algorithm manages to reject faulty samples, while TPR represents the fraction of times when our algorithm doesn't reject a reliable sample. Due to this thesis' focus on reliability in worst-case scenarios, TNR is the most important metric, as the obtained algorithms aim at rejecting faulty measurements to avoid corrupting the fused estimates. However, TPR is also relevant, as rejecting too many reliable measurements is not desirable (it may affect localization performance), although it is much less critical than accepting a faulty measurement.

Parameter optimization approaches like grid search, discussed later in this chapter, benefit from having a single metric to compare, which summarizes the preferences regarding the existing trade-offs into a number that can be computed for each considered combination of parameters. Here, two ways of combining TNR and TPR are proposed:

- Computing a **harmonic mean** (Equation 5.1), which gives the same importance to both metrics, but only has a large value when both metrics are high, penalizing imbalances (unlike a plain average).

- Computing a **normalized weighted average** (Equation 5.2), which first normalizes TNR and TPR to have comparable metrics (especially useful for metrics with different orders of magnitude in their ranges), and then imposes the desired importance for these metrics through weights $\alpha_{TNR}$ and $\alpha_{TPR}$. In particular, respective values of 2/3 and 1/3 have been chosen, to give double importance to TNR over TPR. Note that these weights must add up to 1.

$$\phi_1 = \frac{2 \cdot TNR \cdot TPR}{TNR + TPR} \tag{5.1}$$

$$\phi_2 = \alpha_{TNR} \cdot \frac{TNR}{TNR_{max}} + \alpha_{TPR} \cdot \frac{TPR}{TPR_{max}} \tag{5.2}$$

All the aforementioned metrics allow analyzing performance for the algorithms on each individual source. However, in order to have a single global metric for each grid point, the metric value for each source can be combined with the rest of sources. In order to do so, different statistical measures can be used. Due to this thesis' focus on reliability in worst-case scenarios, a low metric value for one of the sources is not desirable, even if the rest of the sources have a high value, so a statistical measure that is sensitive to outliers must be chosen. For this reason, the harmonic mean of the values for each source is chosen (Equation 5.3), as it heavily penalizes single low values, and other choices like the median are discarded, which are robust to outliers and could hide poor performance on a single source.

$$\hat{\phi_i} = \frac{n_{sources}}{\sum_{k=1}^{n_{sources}} \frac{1}{\phi_{i,k}}} \tag{5.3}$$

**Loss function**

As this work deals with a binary classification problem for each source, when training neural networks the binary cross-entropy loss function is used (Equation 5.4), which highly penalizes probabilities that are very far from the actual ground-truth label. With this loss function, training isn't biased in favoring TNR or TPR, which is instead handled by parameter tuning. The same "balanced training and biased tuning" idea has also been applied with classical ML techniques, which use different mechanisms to select error relevance in training.

$$\mathcal{L} = -\frac{1}{M} \sum_{i=1}^{M} \left[ y_i \cdot log(p(y_i)) + (1 - y_i) \cdot log(1 - p(y_i)) \right] \tag{5.4}$$

## 5.2 Test scenarios description

Once proper performance metrics have been defined, data has to be gathered directly from the vehicle. In particular, real sensor data is needed to train and/or tune the proposed algorithms, and afterwards real output data is needed to verify the performance of those algorithms running on the Autonomous Plus2. As the focus of this work lies on reliability and worst-case situations, relevant test scenarios have to be defined, such that they aim to consistently trigger failure modes from Tables 2.1, 2.2, 2.3, 2.4 and 2.5. In order to do so, 16 test scenarios are proposed in Table 5.1, from which independent training, validation and test data are obtained.

## 5.3 Data acquisition and pipeline

In order to train learning approaches (e.g.: weights in neural networks) and/or tune algorithm parameters (e.g.: $k$ in k-NN), real data has to be gathered from the tests scenarios in Table 5.1 and processed to convert it into appropriate labeled data.

Table 5.1: Test scenarios considered for the Autonomous Plus2 in this work

| # | Test name | Description | Target failure modes |
|---|---|---|---|
| 1 | Hangar 1 | Several laps of starting at hangar entrance, traversing the corridor, turning around, and getting back to the entrance. | GPS_3 |
| 2 | Hangar 2 | Several laps of starting at hangar entrance, traversing the corridor, and getting back to the entrance without turning around. | GPS_3 |
| 3 | Hangar 3 | Several laps of starting at hangar entrance, traversing the corridor in a straight line, and getting back to the entrance wihout turning around, all at medium speed. | Camera_4<br>GPS_3 |
| 4 | Around hangar (morning) | Several laps of driving around the hangar, outside, in a closed trajectory. | Camera_3 |
| 5 | Around hangar (afternoon) | Several laps of driving around the hangar, outside, in a closed trajectory. | Camera_3 |
| 6 | Tall building flat 1 | Several laps of driving in a straight line, back and forth, in a flat area next to tall buildings. | GPS_1<br>GPS_2 |
| 7 | Tall building flat 2 | Several laps of driving in a circles, back and forth, in a flat area next to tall buildings. | GPS_1<br>GPS_2 |
| 8 | Tall building sloped 1 | Several laps of driving in a straight line, back and forth, in a sloped area next to tall buildings. | GPS_1<br>GPS_2<br>LiDAR_3 |
| 9 | Tall building sloped 2 | Several laps of driving in circles, back and forth, in a sloped area next to tall buildings. | GPS_1<br>GPS_2<br>LiDAR_3 |
| 10 | Vibration 1 | Several laps of driving in a straight line, back and forth, in a non-smooth terrain made of dihedral-shaped tiles. | IMU_1<br>Camera_6<br>Odom_2 |
| 11 | Vibration 2 | Several laps of driving in circles, back and forth, in a non-smooth terrain made of dihedral-shaped tiles. | IMU_1<br>Camera_6<br>Odom_2 |
| 12 | Tunnel passage | Several laps of going back and forth in a "tunnel passage" area. | LiDAR_2 |
| 13 | Open field | Several laps of driving in a straight line, back and forth, in an open field area. | LiDAR_2<br>Camera_5 |
| 14 | Main ramp 1 | Several laps of going up and down a steep ramp, without turning around. | LiDAR_3<br>Camera_7 |
| 15 | Main ramp 2 | Several laps of going up and down a steep ramp, turning around to always face forward. | LiDAR_3<br>Camera_7 |
| 16 | Heavy maneuvering | Several laps of navigating in an open parking area while doing intense direction changing ("snake movements"). | Odom_1 |

### Data acquisition

Regarding data acquisition, the process can be handled easily thanks to the available ROS architecture. Using capabilities from the rosbag package, the topics shown in Figure 3.4 that are already available from existing nodes in the Autonomous Plus2 are recorded into `.bag` files. Those files can be either replayed on an external computer with ROS installed, or opened in Matlab using functions from the ROS Toolbox. The latter allows for offline quick visualization, scripting and simulation, which has been very useful for developing and testing under COVID restrictions, reducing the required hands-on time on the vehicle. Each recorded `.bag` file includes all topics needed for the proposed solution (see Algorithms 4.5 and 3.1) in a certain scenario. Once algorithms had been developed and deployed in ROS nodes, the output information for the proposed solution also started to be recorded in the `.bag` files (by recording the corresponding topics, also displayed in Figure 3.4), in order to assess performance when running in the real vehicle (although the developed Matlab scripts could already simulate the expected output with very high fidelity).

### Data labeling

Before the different algorithms can be trained with the relevant data recorded in a set of `.bag` files, this data needs to be labeled in order to be able to compute the proposed performance metrics and/or define loss functions for the learning approaches. Before this thesis, choosing which sensor to trust according to the expected environment and issues the Autonomous Plus2 was likely to encounter had to be manually done by an experienced human technician before navigation. In consequence, it is reasonable to assume that labeled data can be obtained by having a person manually label the recorded data for each of the different scenarios.

In particular, by observing temporal plots of $x'_i$, $y'_i$ and $\theta'_i$ from all sources at the same time, together with X-Y trajectory plots, it is possible to generate 0/1 binary labels for this problem, where a 0 label is assigned to parts of the trajectory where one sensor is clearly presenting some failure mode. This decision is subjective and not completely systematic, but it can be aided by the column for target failure modes in Table 5.1, by qualitative knowledge of the actual trajectory done by the human driving during the test, by qualitative and quantitative analysis of the superposed plots for each source, etc. The resulting human-labeled data gives reasonable results, as it can be seen in Figure 5.1, where recorded and labeled data for Scenario #3 is shown.

This labeling approach has proven to be good enough to allow the algorithms to learn the patterns and interactions that occur when failure modes arise, even though the exact moment of the transition from 0 to 1 labelling can be slightly arbitrary and thus some labels near the transition zone can actually be ambiguous/wrong. This problem arises only for a few time steps for each test scenario, which doesn't have a significant effect in the algorithms' qualitative performance and reliability. Nevertheless, the somewhat arbitrary transition time and the fact that there is not an absolute objective "ground-truth", but a partially subjective decision instead, does affect the quantitative metrics' values, which are actually approximate/indicative values (unlike, for instance, in pure localization accuracy problems, where there actually exists an objective ground-truth position where the vehicle ultimately is, regardless of whether it is known precisely or not).

Note that choosing a fixed and systematic labeling decision rule instead that can automatically label any training data with no human subjectivity is not an option, as it would be equivalent to this thesis' problem itself, so if any such rule were to be used to label the training data instead of human labelling, the solution to this thesis problem with 100% accuracy would be to just have an algorithm that mimics that rule and would bias any solution towards learning that particular (arbitrarily) chosen rule.

### Train/validation/test split and data augmentation

Once labeled data has been produced, a train/validation/test split is performed. Training data is used to train learning-based methods, validation data is used to tune algorithm parameters, and test data is used to evaluate the performance of the best obtained version of each algorithm.

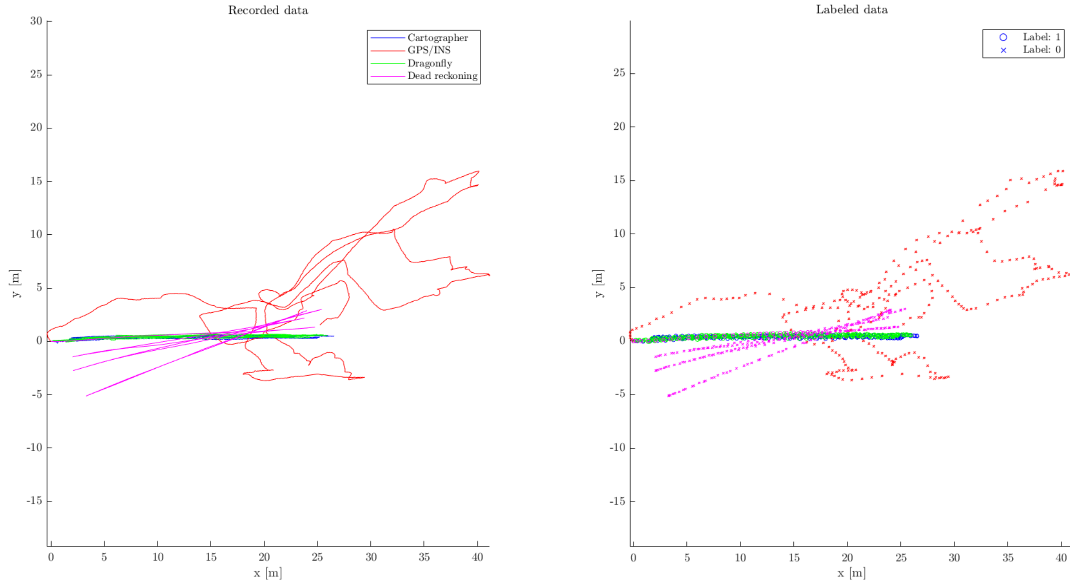


Figure 5.1: Example of recorded (left) and labeled (right) data from Scenario #3. The human labeling produces a qualitatively good result, where data from sources under failure is rejected (only after the failure).

In order for performance evaluation to be unbiased, it is important not to mix samples of the same `.bag` file in training, validation and test datasets, as they could be significantly correlated, especially samples of the same `.bag` file that are close in time. Choosing different `.bag` files for training, validation and testing allows the algorithm to fit and generalize well, and to offer reliable quantitative performance metric estimates, as long as both datasets have examples of various failures modes for each source. Note that this restriction doesn't prevent from training and testing on the same scenario from Table 5.1, as long as at least 2 different `.bag` files have been recorded for it.

Due to the limited number of different and relevant areas for data collection and testing available during this thesis' development and a bounded number of considered failure scenarios of interest that can be triggered repetitively, the amount of "different" data that can be gathered is limited. Moreover, test scenarios usually have a certain fixed orientation due to the position of tracks, buildings, and other elements of interest, which is a form of bias in the available data. The combination of this two effects can lead to some algorithms tending to overfit the training data, either learning its biases or not managing to learn the relevant underlying patterns that allow for proper classification in any new scenario.

A resource-cheap way to overcome this limitation is data augmentation, which consists on increasing the amount of training data available by creating synthetic (but realistic and representative) data from existing (real) data. In particular, for this work, the original training data is augmented by adding random displacements (of a maximum size of the same order of magnitude as the overall trajectories in the test scenarios, i.e. around $\pm 20$ m) and random rotations (of any magnitude between $-2\pi$ and $+2\pi$). Such technique only requires a bit of offline computation power and can potentially increase the available data to any desired amount, although the relevance of new synthesized data starts to decrease at some point. Data augmentation is not performed on validation or test data, as it is good common practice to try to always obtain these datasets directly from the real system with no modification, so that its data follows the desired target "distribution".

In particular, in this thesis, data from the proposed scenarios has been collected twice. The first data is split for train and validation datasets, while the second data is used for the test dataset, in

order to ensure unbiased performance evaluation. Before data augmentation, the first labeled data follows approximately a 60%/40% train/validation split (measured as the respective fraction of the total number of samples), with 11 different `.bag` files for the training dataset, comprising a total of 40:14 min of recordings, and 6 different `.bag` files for the validation dataset, for a total of 25:31 min of recordings. Data augmentation has been used to make the training dataset around 100 times bigger than originally, also eliminating the potential position origin and orientation biases in it. Such data augmentation has displayed a regularizing effect, significantly driving performance and qualitative generalization capabilities up. For the test dataset, 16 different `.bag` files have been used, with a total of 29:27 min of recordings.

## 5.4 Training and parameter tuning procedures

Once training and test data are available, the algorithms can be trained and tuned to achieve their best potential performance for this thesis' problem. In this section, the applied parameter tuning techniques and optimization algorithms are presented. The particular results of these procedures are described and discussed in Chapter 6.

#### Grid search

In this work, for tuning each algorithm's parameters, a grid search approach is used, which consists on exhaustively searching through a manually chosen set of values of each parameter, which defines a grid in the parameter space. Once every possible point in the grid is tested, a sampled approximation to the function of each performance metric with respect to the algorithm's parameters is obtained, which allows to choose the final values of each parameter, taking into account usually existing trade-offs. Grid search suffers from high computational (offline) demand, which increases significantly (and not proportionally) with the number of parameters to explore. However, sometimes the different parameters can be explored independently, as long as they do not interact significantly, reducing the dimensionality of each parameter grid and thus the overall computational load.

#### ExNIS tuning

For the presented version of the ExNIS sensor validation approach, summarized in Algorithm 4.1, there are two parts of the algorithm that can be tuned:

- Thresholding: either parameter $th$ (1-level) or parameters $th_1$, $th_2$ and $th_3$ (3-level).
- Low-pass filtering: either parameter $\beta$ (EWA) or $\nu$ (CUSUM).

Unlike in some learning-based approaches, all these parameters can relatively easily be tuned manually (e.g.: by trial-and-error and using some sample data as guidance). However, systematic parameter optimization methods can also be used, which also provide quantitative insight on the effect of parameters on the final values of the performance metrics.

In order to avoid dimensionality problems, first two grid searches are performed, one with the EWA filtering and the other with the CUSUM filtering, both using 1-level thresholding (which only has 1 parameter), to determine which of the two filtering approaches has a better potential in terms of performance. Then, once the best low-pass filtering method has been chosen, a grid search is performed with 3-level thresholding together with the best filtering, to determine the best thresholding approach and decide the final chosen parameters for the ExNIS approach.

Doing a grid search on 3-level thresholding already includes the 1-level case (it is a degenerate case of it), as discussed in Chapter 4, but this separation in 2 phases avoids a single grid search on 5 different parameters, which would require very high computational times (of the order of days), and also allows for an isolated analysis on low-pass filtering techniques with a simpler thresholding method.

**Decision tree training and tuning**

As mentioned in Chapter 4, a decision tree is trained for each of the 4 sources of this thesis. For this model, one parameter is tuned: the maximum number of splits. A simple grid search on this parameter is used. A quantitative discussion on whether to use ($x'_i$, $y'_i$, $\theta'_i$) directly or to perform manual feature engineering first is also developed using this simple grid search.

Regarding the training procedure, which is repeated for each point in the grid, the augmented training dataset is used, as it drastically reduces overfitting and dataset size does not make the final model heavier. To train the decision trees, the Classification Learner from Matlab's Statistics and Machine Learning Toolbox is used, as well as the Python library scikit-learn.

**k-NN tuning**

Analogously to the case of decision trees, one k-NN classifier is built for each of the 4 sources in this thesis. The only parameter to tune in this case is $k$ (the number of nearest neighbors to take into account). A simple grid search on this parameter is used again, but in this case no training occurs. All computational load is differed until new sample prediction, and the training data only needs to be stored. However, due to these distinctive aspects of k-NN, final model size and inference time grow with training dataset size. Thus, instead of the full x100 data augmentation, after $k$ has been selected, a smaller augmentation is proposed, after a quantitative discussion to see which is the smaller augmentation that achieves similar levels of performance to the x100 case.

**FC feedforward NN training and tuning**

For fully-connected feedforward neural networks, one unique network is trained for all the 4 sources of this thesis, making the intermediate layers focus on features that can help classify several sources at the same time. For this algorithm, there are many parameters (usually also called hyperparameters) than can be tuned. Here, three hyperparameters are tuned, one for the optimizer and two for the model, as they tend to have the biggest impact on final performance: the learning rate, the amount of intermediate layers, and the number of units per layer. A grid search is performed on these 3 parameters.

Regarding the training procedure, repeated for each parameter combination in the grid, the full augmented training dataset is used, as it doesn't make the final model heavier and has a desirable regularizing effect. To train the neural networks, the Adam optimizer is used [49], together with the Keras library on Python.

**RNN training and tuning**

The training and tuning procedures are analogous to those for fully-connected feedforward neural networks, but having recurrent FC layers instead of plain dense intermediate layers.

# Chapter 6

# Results and Discussion

After describing this thesis' system and problem, proposing solutions to it, and defining the experimental procedure to use, here, the actual obtained results are presented and discussed. In particular, model training and tuning results for each approach guide towards the selection of the best version of each of them, which have been deployed on the Autonomous Plus2 as described in Figure 3.4. Once a final version of each algorithm is ready, they are compared with each other in the different scenarios described in Table 5.1. Finally, overall performance and reliability of the approaches is discussed, reaching a final decision and insights on the existing trade-offs, weak points and limitations.

## 6.1 Model training and tuning results

### 6.1.1 ExNIS cross-validation

**Choosing a low-pass filter**

Following the split strategy described in Chapter 5, the first two grid searches are performed in order to compare the low-pass filtering techniques while keeping the simplest thresholding approach.

For the thresholding parameter $th$, the following values are considered, extracted from the $\chi^2(3)$ distribution (as the measurement vector is 3-dimensional): 0.072, 0.115, 0.352, 0.584, 1.212, 2.366, 4.11, 6.25, 7.82, 11.35, 12.84, 16.3. For the $\beta$ parameter, the values in Table 6.1 are considered, which are presented together with their equivalence in number of time steps taken for averaging, according to Equation 4.10. For the $\nu$ parameter, equally spaced values between 0.0 and 12.0 are considered, with a step of 0.5. Figure 6.1 shows the results for the grid search using EWA for low-pass filtering. Figure 6.2 shows the results for the grid search using CUSUM.

In Figure 6.1, values for $T\hat{N}R$, $T\hat{P}R$, $\hat{\phi}_1$ (Equation 5.1) and $\hat{\phi}_2$ (Equation 5.2) are plotted as a function of $th$ and $\beta$. The resulting sampled surface allows to visualize the relations between performance metrics and algorithm parameters, and to assess the existing trade-offs.

First, it is clear that the parameter with the biggest influence, by far, is $th$, as it directly defines the decision rule. The effect of $\beta$ is secondary, as it only controls the low-pass filtering profile, and it only starts to be significant with values of $\beta$ around 0.8 (averaging approximately the last 5

Table 6.1: Values considered for the $\beta$ parameter for EWA, together with an approximate value for the number of steps used for the exponentially weighted average.

| **Beta** | 0 | 0.5 | 0.8 | 0.9 | 0.95 | 0.98 | 0.99 |
|---|---|---|---|---|---|---|---|
| **Time steps** | 1 | 2 | 5 | 10 | 20 | 50 | 100 |

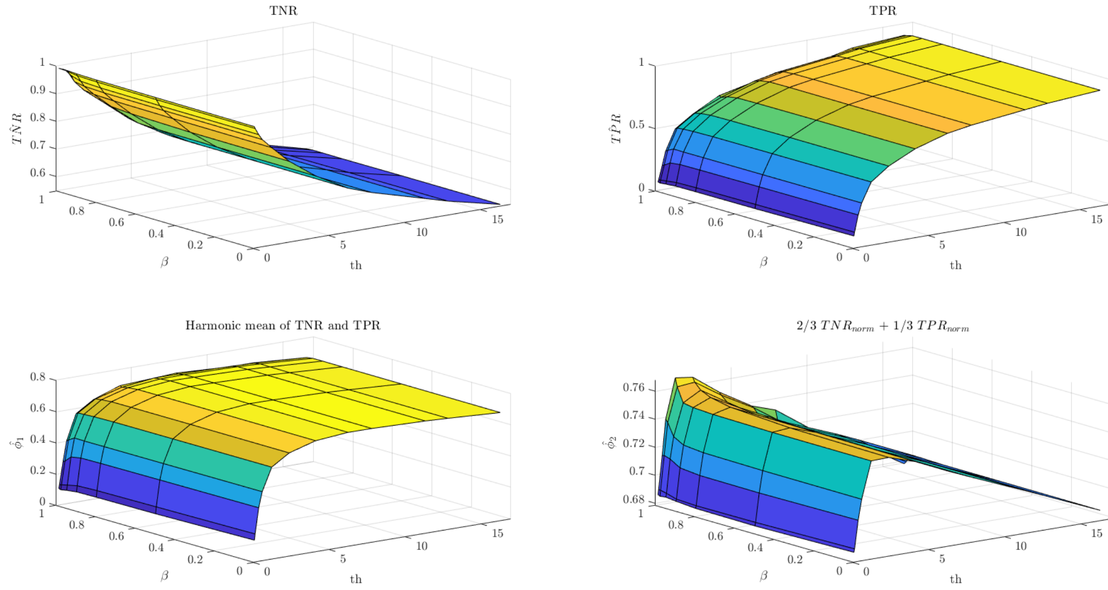


Figure 6.1: Grid search on the ExNIS approach using EWA and 1-level thresholding. Two parameters define the grid: $th$ and $\beta$.

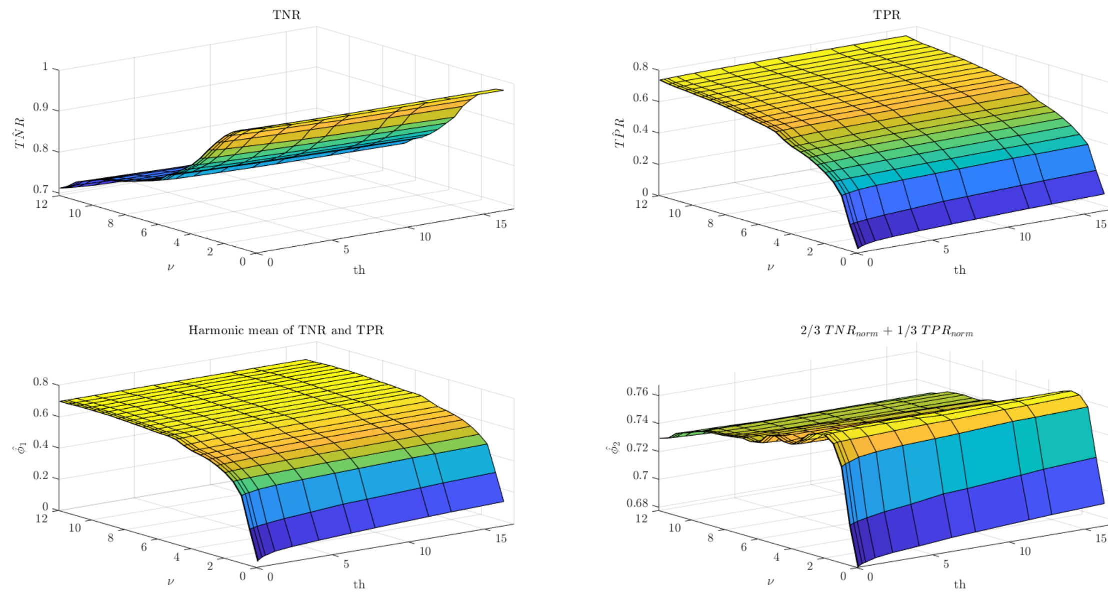


Figure 6.2: Grid search on the ExNIS approach using CUSUM and 1-level thresholding. Two parameters define the grid: $th$ and $\nu$.

time steps). Values of $\beta$ closer to 1 increase the time-averaging effect, generating "smoother" parity relations, which favors TNR and harms TPR. The $th$ parameter has opposite effects on TNR and TPR, with higher values of $th$ yielding a more "permissive" algorithm, which results in lower TNR and higher TPR.

Observing the resulting combined metrics $\hat{\phi}_1$ and $\hat{\phi}_2$, a stronger trade-off effect is observed in $\hat{\phi}_2$. In the higher $th$ ranges, while the harmonic mean is more permissive with lower TNR in exchange for higher TPR, the weighted average clearly favors TNR and makes this range for $th$ much less desirable. The respective maxima for each metric is reached in the following points of the parameter grid: $\hat{\phi}_{1,max} = 0.731$ for $th = 6.25$, $\beta = 0.8$, and $\hat{\phi}_{2,max} = 0.768$ for $th = 1.212$, $\beta = 0.99$.

In Figure 6.2, the same plots are shown for the grid search with CUSUM filtering, which has the parameter $\nu$, instead of $\beta$. Here, it is the parameter $\nu$ that has the strongest influence, conditioning both the low-pass filtering and the final performance, and leaving $th$ as a secondary parameter. Greater values of $\nu$ favor TPR and harm TNR, because a bigger subtracted drift creates a more "permissive" algorithm. There is also an optimal value of drift, after which overall performance decreases according to $\hat{\phi}_1$. The values where each metric reach their respective maxima are: $\hat{\phi}_{1,max} = 0.702$ for $th = 16.3$, $\nu = 9.5$, and $\hat{\phi}_{2,max} = 0.767$ for $th = 16.3$, $\nu = 2$.

Comparing the maximum performance obtained, EWA reaches higher maxima than CUSUM in both metrics. For this reason, EWA has been chosen as the low-pass filtering approach for the final ExNIS approach on the vehicle. Note that performance without filtering has also been considered with the grid points corresponding to $\beta = 0$.

**Tuning the general 3-level thresholding approach**

With the EWA filtering approach fixed, a grid search is performed on the proposed 3-level ExNIS approach. This way, the parameters that define the grid are $th_1$, $th_2$, $th_3$ and $\beta$. For the thresholding parameters $th_i$, the same set of values is chosen as in the last grid searches for $th$, while for $\beta$ the previous values of this parameter are also copied.

In this case, a 4-dimensional grid is built, which can't be directly visualized in all its dimensions. In order to observe trade-offs and parameter effects, two parameters can be fixed each time (for instance, to their optimal values) and the two others can be used as variables for plotting the local 3D surface corresponding to each performance metric. The values where each metric reach their respective maxima are: $\hat{\phi}_{1,max} = 0.731$ for $th_1 = 6.25$, $th_2 = 7.82$, $th_3 = 0.072$, $\beta = 0.5$, and $\hat{\phi}_{2,max} = 0.768$ for $th_1 = 1.212$, $th_2 = 0.072$, $th_3 = 0.072$, $\beta = 0.99$. Figures 6.3 and 6.4 show some local visual results for this grid search.

In Figure 6.3, the effects of parameters on $\hat{\phi}_1$ can be visualized. Like in other previous analyses, $\beta$ starts to have a significant effect around a value of 0.8, but the effect is still mild in comparison to the thresholding variable $th_1$, which is the most relevant parameter for performance. An increase of $th_1$ improves performance until the 6.25 - 7.82 range, and after that it starts worsening the performance. Parameter $th_2$ also has a relevant effect, as it interacts with the effect of $th_1$, as it can be seen in the lower-right plot in Figure 6.3. The other thresholding parameter, $th_3$, as already discussed, only "affects" the algorithm when its value is higher than the previous $th_i$ (otherwise, degenerate cases occur), and in those cases it doesn't provide better performance. The resulting maximum is a degenerate case, where the proposed approach has 2 active levels, defined by $th_1$ and $th_2$. There is a significant increase in the maximum value for metric $\hat{\phi}_1$ with this proposed thresholding approach.

In Figure 6.4, the effects of parameters on $\hat{\phi}_2$ can be visualized. Note that conclusions on $\beta$'s effect are the same as before. Lower values of $th_1$ offer better performance on this metric, as they highly increase the TNR, although the resulting TPR is quite worse than the obtained in $\hat{\phi}_1$'s maximum. This fact, again, indicates that $\hat{\phi}_2$ is much more permissive than $\hat{\phi}_1$ with low TPR values as long as TNR is significantly higher in exchange, which for our problem is not always the best option (having low TPR implies rejecting many valid measurements), suggesting that $\hat{\phi}_1$ can be a better

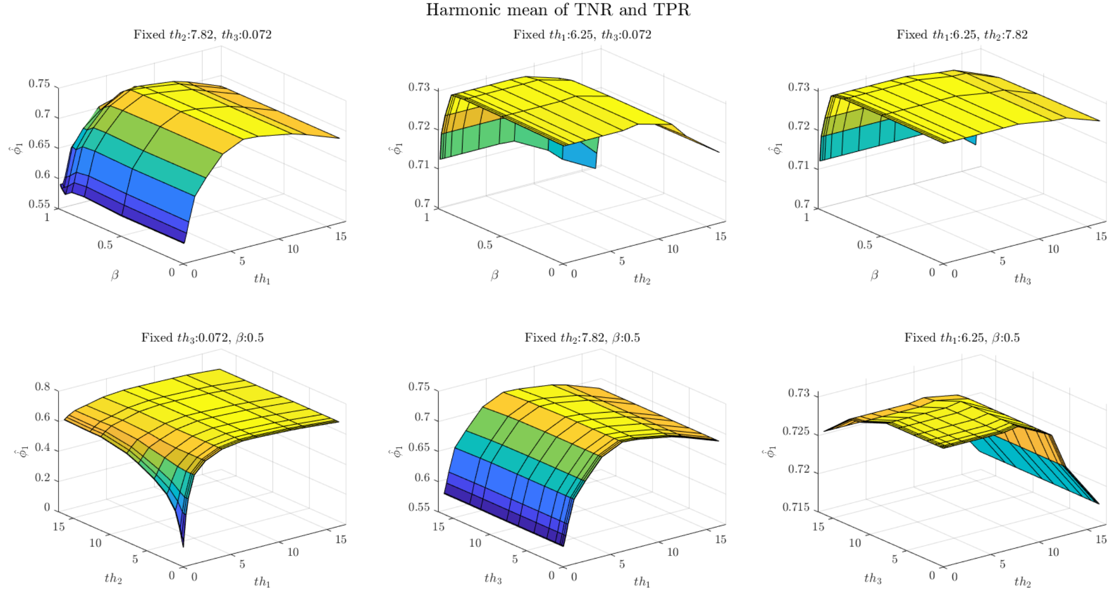


Figure 6.3: Results for the aggregated metric $\hat{\phi}_1$ on the ExNIS grid search using EWA and 3-level thresholding. Four parameters define the grid: $th_1$, $th_2$, $th_3$, and $\beta$.

choice of performance metric for this problem. Finally, for the aggregated metric $\hat{\phi}_2$, $th_2$ and $th_3$ only worsen performance when set above $th_1$ (i.e., when they are actually "doing something").

From the plots and the resulting maxima, it can be concluded that the extended 3-lvl approach offers significant performance improvements for this problem in terms of $\hat{\phi}_1$, but the maximum for $\hat{\phi}_2$ is the same. It seems that it is worth it to have a more complex solution in exchange for slightly better performance, especially due to the very tiny increase in computational load it implies.

After all the presented analyses, the final ExNIS cross-validation approach chosen for the vehicle is:

- Use fixed $Q_k$ matrices from Equations 3.3, 3.5, 3.6 and 3.11.
- Use EWA for low-pass filtering, with $\beta = 0.5$ (approx. 2 time steps averaging).
- Use 2-lvl thresholding, with $th_1 = 6.25$ and $th_2 = 7.82$ (10% and 5% probability of the values happening in nominal conditions, respectively).

With these choices, the final values for the aggregated metrics are: $\hat{\phi}_1 = 0.731$ and $\hat{\phi}_2 = 0.733$.

### 6.1.2 Decision trees

For decision trees, the only parameter tuned is the maximum number of splits, for which the following values are considered: 1, 3, 5, 7, 10, 15, 25, 50, 100. However, there is another very important factor when applying such classical ML techniques: whether to apply manual feature engineering on the inputs. As discussed in Chapter 4, there are many possible manual features to extract for this thesis' problem (e.g.: subtracting variables from different sources, computing numerical derivatives, etc.), but in this work the ExNIS parity relations are considered as relevant features, as they guide the algorithms towards paying attention on differences between sources' variables (similar to an unsigned subtraction calculation).

In order to determine whether to use $(x'_i,\ y'_i,\ \theta'_i)$ directly, use the 6 parity relations, or just use the 3 parity relations theoretically associated to the source each tree is classifying, the simple grid

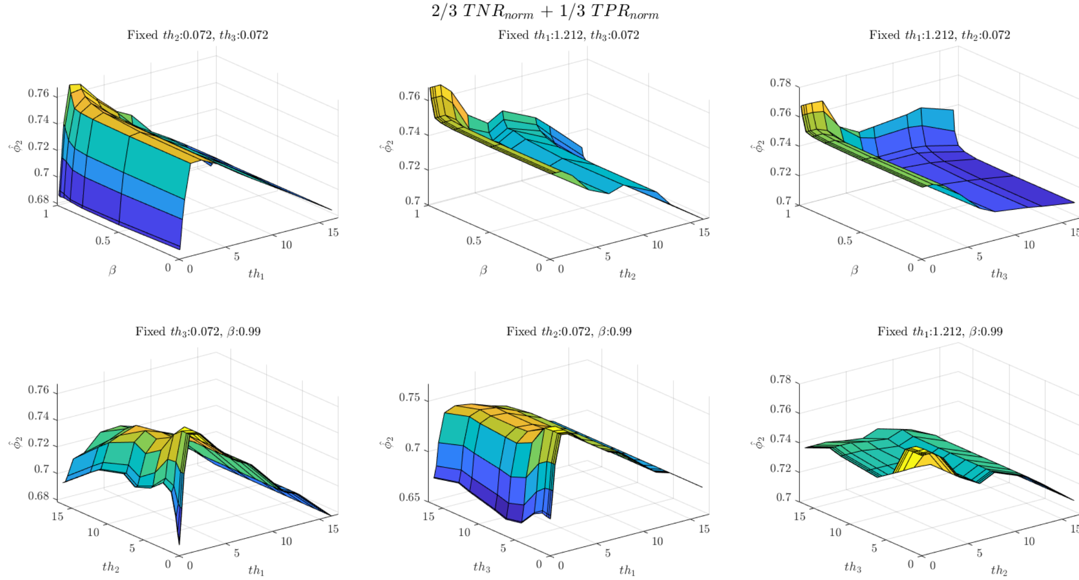


Figure 6.4: Results for the aggregated metric $\hat{\phi_2}$ on the ExNIS grid search using EWA and 3-level thresholding. Four parameters define the grid: $th_1$, $th_2$, $th_3$, and $\beta$.

search described above has been performed for each of the 3 options, yielding the results of Figure 6.5. Note that the same value for the maximum number of splits parameter has been used for each tree, in order to assess the general trade-offs and have a solution that is not heavily over-tuned for the Autonomous Plus2, but that can be easily generalized and migrated to other vehicles and similar problems.

Using the position estimates directly is not a good option, as vanilla decision trees only have decision nodes that do simple thresholding on 1 input variable. Such a structure does not allow the decision tree to compare different sources' variables directly, which is one of the main sources of insight for the decisions in this problem. Thus, here, decision trees clearly fail at providing good performance without prior feature engineering. In particular, it would take a value of around 5000 maximum splits to approach its maximum potential performance, which would still be significantly lower (this analysis is omitted from the plots for visual clarity) and wouldn't generalize as well to new samples.

The differences between using all 6 parity relations or just the 3 parity relations that are theoretically associated to the source each tree is classifying are relatively small. Using all parity relations may provide additional information that could be helpful, while using only the theoretically associated parity relations may help the algorithm focus on what should be important. The peak performance for $\hat{\phi_1}$ (which penalizes imbalanced TNR-TPR situations better than $\hat{\phi_2}$) is achieved using only the associated parity relations. Although the difference is small, the final chosen decision trees use (each) the theoretically associated parity relations as inputs.

Figure 6.5 also shows how larger values for the maximum number of splits tend to cause higher TNR, but lower TPR, until around the value 25, where the relation stabilizes for both metrics. Both metrics present a maximum around values 10-15, and from there on performance only degrades, suggesting that the additional splits cause overfitting to the training data, instead of allowing for better generalization.

In this case, as decision trees are very computationally cheap, the increase in performance is more important than the slight increase in model size and inference time with higher parameter values,

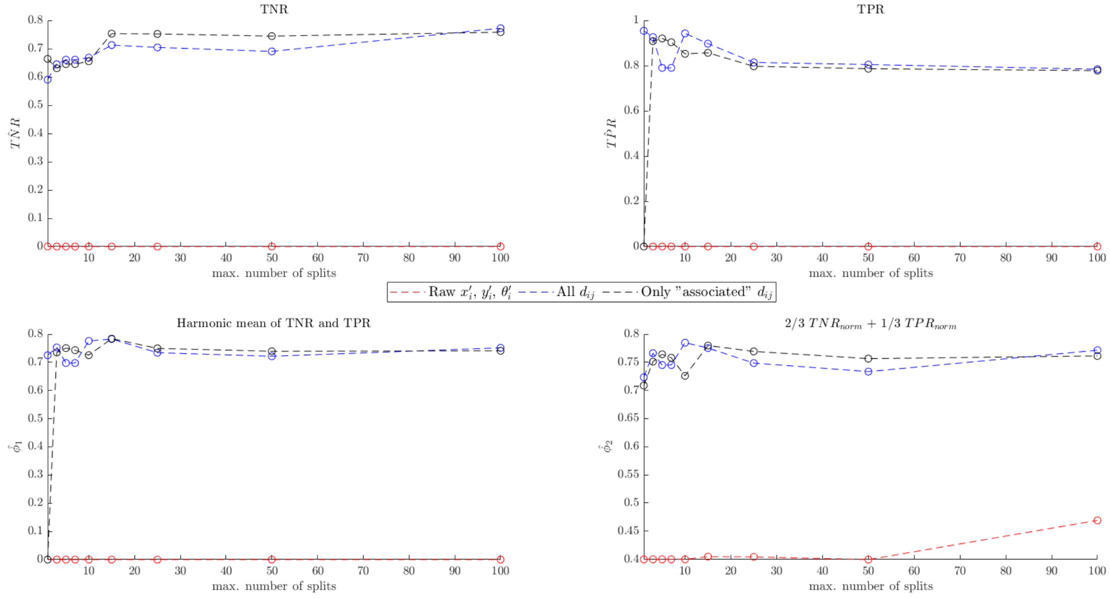


Figure 6.5: Grid search on decision trees for the 3 input options. Only one parameter defines the grid: the maximum number of splits. The same parameter value is used for each source's tree.

but as results show, once the maximum is achieved, adding new splits to the tree doesn't improve the trees' ability to classify, so the value of 15 is chosen.

After all the presented analyses, the final decision tree approach chosen for the vehicle is:

- Use 4 decision trees, one for each source.
- For each tree, use its theoretically associated parity relations as inputs.
- For all trees, max. number of splits = 15.

With these choices, the final values for the aggregated metrics are: $\hat{\phi}_1 = 0.784$ and $\hat{\phi}_2 = 0.780$.

### 6.1.3 k-Nearest Neighbors

For k-NN, the only parameter tuned is $k$, the number of considered neighbors, for which the following (odd) values are considered: 1, 3, 5, 7, 11, 15, 25, 51, 101. Analogously to the decision tree approach, this grid search is repeated for the 3 input options, yielding Figure 6.6. Note that, again, the same value for $k$ has been used for each classifier, in order to see the general trade-offs, which can be used when migrating this solution to other vehicles.

In this case, it can be seen that using only the theoretically associated parity relations for each classifier is the best choice, while not doing manual feature engineering fails again to perform well, as the structure of the classifier doesn't allow it to easily learn cross-validation logic. As k-NN uses a distance function that does not prioritize any input feature, when computing the nearest neighbors, if all 6 parity relations are used, differences (distance) in a theoretically associated parity relation are treated the same way as differences in theoretically non-associated parity relations, which may explain the drop in performance when adding the remaining 3 parity relations.

As shown in Figure 6.6, the value of k doesn't seem to impact performance with x100 data augmentation. For k-NN, the value of $k$ impacts prediction time (which for most implementations of k-NN is $\mathcal{O}(k \cdot \log M)$), but not model size, so choosing a smaller $k$ can benefit when running predictions with k-NN.

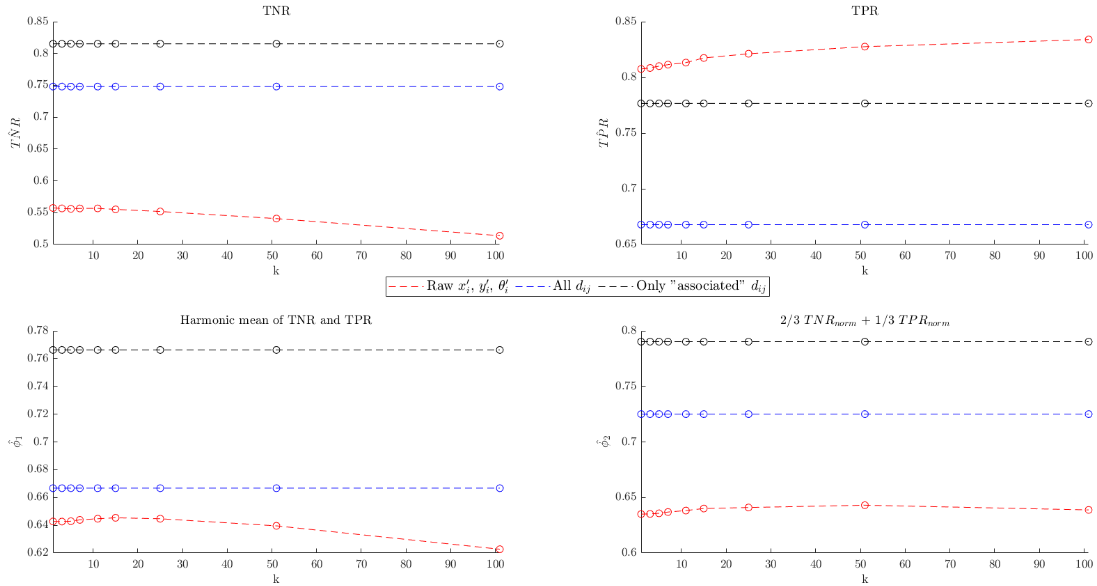


Figure 6.6: Grid search on k-NN classifiers for the 3 input options. Only one parameter defines the grid: $k$. The same parameter value is used for each source's classifier.

Once the type of inputs have been decided, and a flat profile for $k$ has been observed, in order to reduce the size and inference time of the resulting k-NN classifiers, another grid search has been performed to assess which is the minimum data augmentation that provides similar performance to the previously used x100 augmentation for k-NN. Figure 6.7 shows the results of this data augmentation grid search, for fixed values of $k$ 1, 5 and 11. Looking at different values of $k$ is interesting because for lower augmentations the flat profile for $k$ may not hold.

As seen in the figure, data augmentation does not offer an improvement in the k-NN classifiers' performance, while it does increase model size and inference time. Also, using a value of 1 for $k$ allows for peak performance with no data augmentation, which is the best situation possible. For higher values of $k$, as more neighbors are taken into account, data augmentation is needed to achieve peak performance.

After all the presented analyses, the final k-NN approach chosen for the vehicle is:

- Use 4 k-NN classifiers, one for each source.
- For each classifier, use its theoretically associated parity relations as inputs.
- For all classifiers, k = 1 with no data augmentation.

With these choices, the final values for the aggregated metrics are $\hat{\phi}_1 = 0.667$ and $\hat{\phi}_2 = 0.725$.

### 6.1.4 FC feedforward NN

For fully-connected feedforward neural networks, three parameters are tuned. For the learning rate, the values 0.1, 0.01, 1e-3, 1e-4 and 1e-5 are considered; for the number of intermediate layers, the values 1, 2, 3, 4 and 5 are considered; for the number of units in each intermediate layer, the values 2, 4, 8, 16, 32 and 64 are considered.

In this case, no previous feature engineering is performed, as neural networks can already learn appropriate feature mappings from data without a priori knowledge. Theoretically, this capabilities increase as the number of intermediate layers increase, although those benefits come with an increased proneness to overfitting the training data.

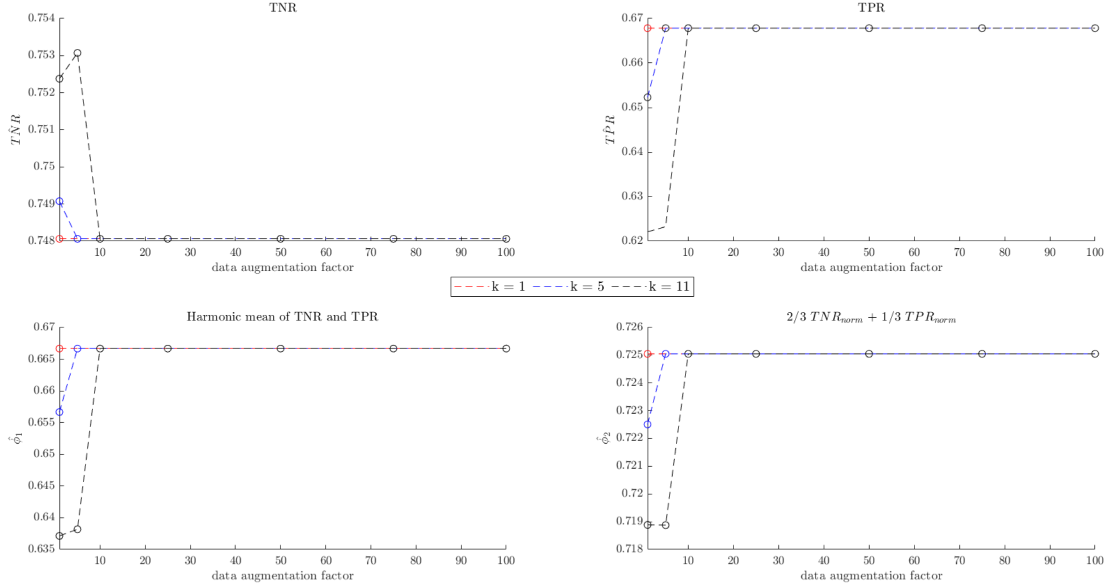


Figure 6.7: Grid search on k-NN classifiers for $k$ = 1, 5 and 10. Only one parameter defines the grid: the data augmentation factor. The same parameter value is used for each source's classifier.

Regarding the network's architecture, an input layer with 12 units is needed to ingest the $(x'_i, y'_i, \theta'_i)$ estimates. Intermediate layers use ReLu activation, while the output layer uses sigmoid activation. In the latter, 4 units provide a value from 0 to 1 that can be interpreted as a trust estimation (the value 1 indicating a 100% reliable source), which can then be thresholded to obtain a decision. The thresholding could also be adjusted for each source and using the data, but keeping with the overall approach of having a solution that is not heavily over-tuned for the Autonomous Plus2, which can be easily generalized and migrated to other vehicles and general problems, a threshold of 0.5 is used (common choice in binary classification problems with sigmoid activation functions in the output layer).

In order to perform the training, the Adam optimizer (natively supported in Keras) has been used, whose learning rate can be changed according to the current point in the grid. A batch size of 64 has been used for a maximum of 50 epochs for each grid point, adding early stopping in case the validation loss does not decrease for 5 consecutive epochs (to avoid running for an unnecessary number of epochs), and saving the model with the best AUC on validation data (saving the model with the best validation loss yielded almost the same results, but using AUC may help choose a more robust classifier).

Figures 6.8 and 6.9 show some local visual results for this grid search. The effects of parameters on $\hat{\phi}_1$ and $\hat{\phi}_2$ are almost completely analogous. Many combinations of parameters can be observed that do not provide proper learning, especially when the learning rate is too high/low (0.1 and 1e-5, extreme values) or the network is too shallow (low values of number of intermediate layers). An intermediate value of learning rate (0.01) provides the best learning results. Having the highest value of intermediate layers (5) provides great performance, both with high number of units (64) and lower number of units (8).

After all the presented analyses, the final feedforward FCNN approach chosen for the vehicle is:

- Use 1 neural network, which outputs the trust for each sensor.
- Use a 0.5 threshold for converting trust into selection.
- Use a learning rate of 0.01.

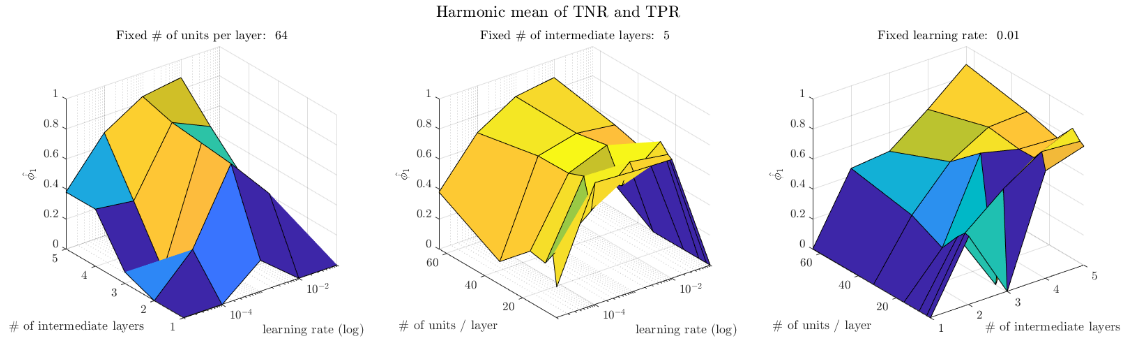


Figure 6.8: Grid search on feedforward FCNN classifiers (aggregated metric $\hat{\phi}_1$). Three parameters define the grid: the learning rate, the number of intermediate layers and the number of units in each layer.

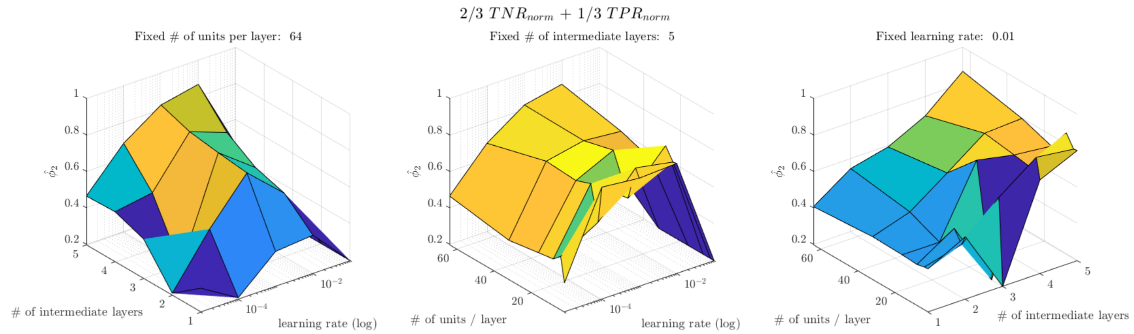


Figure 6.9: Grid search on feedforward FCNN classifiers (aggregated metric $\hat{\phi}_2$). Three parameters define the grid: the learning rate, the number of intermediate layers and the number of units in each layer.

- Use 5 intermediate layers, each with 64 units.

With these choices, the final values for the aggregated metrics are $\hat{\phi}_1 = 0.879$ and $\hat{\phi}_2 = 0.866$.

### 6.1.5 Recurrent NN

For recurrent neural networks, the same three parameters as in the feedforward case are trained, with the same considered ranges of values. Again, no previous feature engineering is performed.

Regarding the network's architecture, the input layer has the same properties as in the feedforward case (12 units for the pose estimates). Intermediate recurrent layers use ReLu activation, while the output layer uses sigmoid activation. The same units (and thus weights) are applied to every time instant in the sequence (in Keras, this is achieved by using SimpleRNN and TimeDistributed layers). Again, non-binary trust information is found in the output, which is also thresholded with a threshold of 0.5.

In order to perform the training, the Stochastic Gradient Descent optimizer (current implementation of Adam in the Keras library sometimes triggers numerical issues when using SimpleRNN layers) has been used, whose learning rate can be changed according to the current point in the grid. A batch size of 64 has been used for a maximum of 30 epochs for each grid point, again adding early stopping in case the validation loss does not decrease for 5 consecutive epochs, and saving the model with the best AUC on validation data.

Figures 6.10 and 6.11 show some local visual results for this grid search. Again, the effects of parameters on $\hat{\phi}_1$ and $\hat{\phi}_2$ are almost completely analogous. The worst performance results are obtained when using a small number of units per intermediate recurrent layer. Too small learning rates also prevent proper learning in a reasonable amount of epochs. The best results are obtained with a moderate amount of intermediate layers (3), and having a higher number of units per layer (64) also helps performance. The highest explored learning rate (0.1) gives the best results, also providing faster convergence. Nevertheless, in this case the trade-offs sometimes aren't as strong and clear, with some surfaces having a low smoothness due to abrupt local minima, so the combination of parameters that turns out to yield the highest values for $\hat{\phi}_1$ and $\hat{\phi}_2$ is directly chosen.

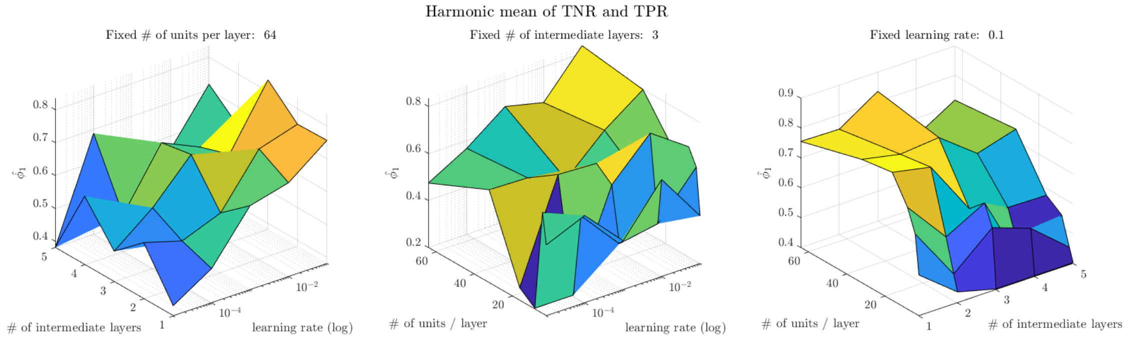


Figure 6.10: Grid search on RNN classifiers (aggregated metric $\hat{\phi}_1$). Three parameters define the grid: the learning rate, the number of intermediate layers and the number of units in each layer.

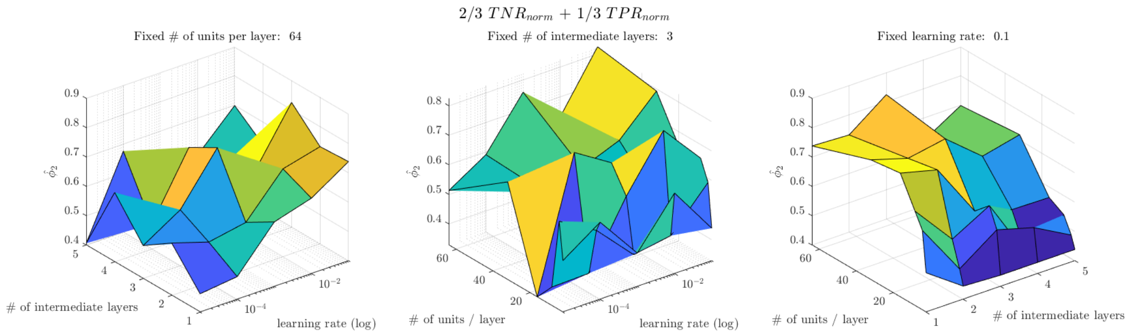


Figure 6.11: Grid search on RNN classifiers (aggregated metric $\hat{\phi}_2$). Three parameters define the grid: the learning rate, the number of intermediate layers and the number of units in each layer.

After all the presented analyses, the final RNN approach chosen for the vehicle is:

- Use 1 neural network, which outputs the trust for each sensor.
- Use a 0.5 threshold for converting trust into selection.
- Use a learning rate of 0.1.
- Use 3 intermediate layers, each with 64 units.

With these choices, the final values for the aggregated metrics are $\hat{\phi}_1 = 0.792$ and $\hat{\phi}_2 = 0.785$.

### 6.1.6 Note on performance metrics

During the tuning of the different approaches, $\hat{\phi_1}$ and $\hat{\phi_2}$ have been computed in all cases to try to condense all main performance traits into 1 number that can be used to choose the best point in each grid (and thus the best parameter for each algorithm).

From the analyses presented, the effects of parameters on both metrics are usually quite similar, with the main difference being that $\hat{\phi_1}$ penalizes more cases with low TPR, even when TNR is very high, while $\hat{\phi_2}$ can hide such situations due to arithmetically averaging and using a higher weight on TNR. For this reason, $\hat{\phi_1}$ is chosen here as the best considered performance metric to summarize the solutions' performance in 1 number (e.g.: for parameter optimization). Nevertheless, when analyzing performance in specific scenarios (which is the object of the next section of this chapter), it is interesting to use more metrics than just a single number, in order to get a complete picture of the actual performance in each case, and to be able to assess in which situations each algorithm performs better or fails to perform well at all.

## 6.2 Final performance in test scenarios

Table 6.2 presents the final performance results in the test scenarios defined in Table 5.1, obtained with the test dataset. The results are provided for each test scenario and each proposed approach.

Regarding the used metrics, as discussed in the last paragraph of the previous section, to analyze the performance of each specific scenario, apart from a single condensed metric (useful for taking tuning decisions), more metrics are used to paint a more detailed picture of the actual performance of each algorithm for each case. In particular, global TNR and TPR metrics are used, which correspond, respectively, to the total number of failure measurements successfully rejected and to the total number of valid measurements successfully kept, which answer to the 2 main questions in sensor rejection algorithms. Additionally, a condensed global metric $\phi_1$ is provided, computed with Equation 5.1 and the aforementioned global TNR and TPR metrics.

In Table 6.2, the best value for each global metric for each scenario is highlighted in bold. As already forecasted in previous chapters, each algorithm has different performance in each test scenario. The "best" algorithm for each test scenario changes from case to case, and all algorithms have at least a test scenario where they perform better than the others. In terms of the condensed metric $\phi_1$, k-NN and RNN are the algorithms that "win" the biggest number of times (5 each), while decision trees and the voting system "win" only once each. When looked globally (putting together all test scenarios), k-NN and the voting system have the highest values of the global metrics.

The ExNIS cross-validation approach manages to provide the best TPR results in many scenarios, and its TPR is very high in almost all cases, even when it doesn't "win" in terms of that metric. Its TNR, however, is quite irregular and falls behind other approaches' results. In consequence, the ExNIS approach (with the presented tuning) provides quite a "conservative" approach, which doesn't reject many valid measurements in exchange for lower rejection of actual failure measurements, only rejecting when the failure is very clear on the inputs.

The decision trees approach also provides great TPR values in most scenarios (with a global result better than that of ExNIS), while its TNR is at the same level as the ExNIS approach, being quite irregular and not the best. Thus, the decision tree approach (with the presented tuning) also provides a quite "conservative" approach. Note, however, that this approach already manages to reject more than 2 out of every 3 faulty measurements, while only losing less than 5% of the valid measurements in exchange, which is a great result.

The k-NN approach has the greatest TNR results, managing to reject almost 3 out of every 4 faulty measurements, while still having a great TPR, avoiding significant valid information loss. In terms of $\phi_1$ this is the best-performing approach, even better than the voting system that combines all 5 proposed algorithms. Note that these results, together with the decision tree results, confirm that

the parity relations $d_{ij}$ (Equation 4.6) are a great manual feature for this problem, which allow applying classical ML techniques successfully.

The feedforward NN approach has worse performance than its classical ML counterparts. Although TPR is high in most cases, TNR is very irregular and in some failure scenarios it struggles significantly in detecting failure (e.g.: in the "Heavy maneuvering" scenario). However, there are some other scenarios (e.g.: the "Hangar 3" scenario) where it clearly outperforms the other approaches in rejecting failure measurements, so even if it is a worse standalone approach, when used in combination with other approaches in the voting system, it still offers value to the decision scheme. Note that the NN approach performing worse than the classical ML counterparts can be caused by overfitting to the train data and/or non-optimal parameter tuning, as theoretically a deep-enough network should be able to learn a just-as-good or better feature map and classification logic than approaches like k-NN or decision trees. Another reason for this lower performance could be that the training data is not suitable enough for a NN to learn the underlying patterns.

The RNN approach has a very similar performance than the feedforward NN approach, only having slightly lower TPR. This can indicate that this problem doesn't require a time-series treatment of the data, being the instantaneous "snapshot" approach good enough. Also, it reinforces the idea that the neural network approaches are suffering from either overfitting or a training data that is not good enough for such an approach to learn and generalize.

The voting approach, finally, has the best TPR results, combined with the second best TNR results. Although it doesn't have the best possible performance in particular scenarios, combining the pros and cons of the 5 different algorithms allows generating high performance (70% rejection) while keeping a low impact on the valid data (less than 5% loss).

This smoothing effect from the voting system is very beneficial for this thesis' problem, focused on increasing reliability, as, although it causes a decrease in TNR from the best approach for that metric (k-NN), it promotes that the navigation doesn't suffer from performance drop from valid data loss, and it ensures that regardless of the particular failure scenario, the system doesn't rely on only 1 approach that may suffer in a particular case. Also, following the same reasoning used across this entire work, k-NN may have proven the best for the considered scenarios and with the particular gathered data for the test dataset, but when looking for a non-overtuned approach that can generalize to different vehicles and similar problems, having a combined solution is a more reasonable choice, which achieves a performance close to the best individual option, but can have several backups in case this individual option fails. Furthermore, other scenarios may arise when migrating the proposed solution to a different vehicle or set of sensors, which makes the combined solution a more "conservative" choice, ideal when reliability is the main objective.

For these reasons, the voting system is chosen as the final solution for this thesis' problem, which combines the outputs from 5 different algorithms to produce a final choice. Table 6.3 summarizes the relative performance of the algorithms in terms of the global metrics and considering all test scenarios together.

Table 6.2: Final performance results for each scenario with test data

| Test name | ExNIS | | | Decision trees | | | k-NN | | | FF NN | | | RNN | | | Voting | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | *TNR* | *TPR* | $\phi_1$ | *TNR* | *TPR* | $\phi_1$ | *TNR* | *TPR* | $\phi_1$ | *TNR* | *TPR* | $\phi_1$ | *TNR* | *TPR* | $\phi_1$ | *TNR* | *TPR* | $\phi_1$ |
| Hangar 1 | 80.6 | 98.9 | 88.8 | **87.8** | **100** | **93.5** | 75.0 | 86.6 | 80.4 | 69.6 | 98.1 | 81.4 | 70.2 | 63.6 | 66.7 | 85.0 | 98.1 | 91.1 |
| Hangar 2 | 69.2 | 97.6 | 81.0 | 61.5 | **98.7** | 75.8 | **71.7** | 98.1 | **82.8** | 62.4 | 72.7 | 67.2 | 69.4 | 88.5 | 77.8 | 63.3 | 97.7 | 76.8 |
| Hangar 3 | 57.6 | 99.6 | 73.0 | 58.0 | 93.5 | 71.6 | 76.3 | **93.3** | 83.9 | **100** | 84.5 | **91.6** | 79.6 | 70.9 | 75.0 | 62.1 | 92.9 | 74.4 |
| Around hangar (morning) | **86.2** | 87.1 | **86.6** | 73.5 | **98.8** | 84.3 | 76.2 | 96.7 | 85.3 | 60.4 | 97.0 | 74.5 | 45.2 | 97.0 | 61.7 | 74.4 | 98.6 | 84.8 |
| Around hangar (afternoon) | **87.1** | 99.9 | **93.1** | 74.7 | **100** | 85.5 | 67.5 | 99.7 | 80.5 | 58.5 | **100** | 73.8 | 45.6 | **100** | 62.7 | 67.1 | **100** | 80.3 |
| Tall building flat 1 | 79.7 | **99.8** | 88.6 | 80.1 | 93.1 | 86.1 | 85.1 | 96.3 | 90.4 | 94.4 | 79.6 | 86.4 | **96.2** | 72.3 | 82.6 | 85.3 | 92.3 | **88.7** |
| Tall building flat 2 | 60.4 | **98.3** | 74.8 | 74.6 | 79.8 | 77.1 | **75.8** | 79.1 | 77.4 | 75.1 | 72.5 | 73.7 | 74.6 | 83.6 | **78.9** | 74.9 | 80.3 | 77.5 |
| Tall building sloped 1 | 45.3 | **97.0** | 61.8 | 54.9 | 88.1 | 67.6 | 59.1 | 93.2 | 72.3 | 58.4 | 87.6 | 70.1 | **64.8** | 82.8 | **72.7** | 52.7 | 88.1 | 66.0 |
| Tall building sloped 2 | 48.9 | **100** | 65.7 | 68.8 | **100** | 81.5 | 70.7 | 98.4 | 82.3 | 68.2 | **100** | 81.1 | **76.5** | **100** | **86.7** | 70.4 | **100** | 82.7 |
| Vibration 1 | 56.5 | **100** | **72.2** | 51.8 | 96.3 | 67.4 | 55.1 | 90.9 | 68.6 | **58.6** | 94.0 | **72.2** | 51.6 | 94.0 | 66.6 | 51.8 | 96.1 | 67.3 |
| Vibration 2 | 46.2 | 83.0 | 59.4 | 41.4 | **100** | 58.6 | 59.2 | 93.6 | 72.5 | 55.5 | **100** | 71.4 | **66.3** | 87.4 | **75.4** | 55.8 | **100** | 71.6 |
| Tunnel passage | 98.4 | 62.7 | 76.6 | 96.0 | 96.4 | 96.2 | **100** | **100** | **100** | 84.8 | 95.8 | 90.0 | **100** | 93.4 | 96.6 | 98.4 | 96.4 | 97.4 |
| Open field | 61.0 | 86.1 | 71.4 | 74.3 | 97.7 | 84.4 | **100** | **100** | **100** | 78.3 | 90.5 | 84.0 | 63.0 | 95.3 | 75.8 | 70.7 | 99.9 | 82.8 |
| Main ramp 1 | 46.3 | **97.6** | 62.8 | 73.6 | 96.5 | 83.5 | **82.0** | 86.7 | **84.3** | 61.7 | 82.0 | 70.4 | 69.2 | 48.2 | 56.9 | 67.9 | 97.3 | 80.0 |
| Main ramp 2 | 62.8 | **89.9** | 74.0 | 57.7 | 86.0 | 69.1 | 59.3 | 81.2 | 68.5 | 42.9 | 85.6 | 57.2 | **77.6** | 78.0 | **77.7** | 58.4 | 86.0 | 69.6 |
| Heavy maneuvering | 54.7 | **97.5** | 70.1 | 54.7 | 85.4 | 66.7 | **77.0** | 86.4 | **81.4** | 29.5 | 67.5 | 41.1 | 69.8 | 65.5 | 67.6 | 54.7 | 86.4 | 67.0 |
| **GLOBAL RESULTS** | 69.7 | 90.7 | 78.8 | 69.6 | 95.2 | 80.4 | **74.9** | 93.4 | **83.1** | 65.0 | 88.6 | 75.0 | 65.0 | 84.2 | 73.4 | 70.0 | **95.4** | 80.8 |

Table 6.3: Summary of relative final performance for the proposed algorithms (for each metric, top algorithm is the best-performing)

| **TNR performance** | **TPR performance** | $\phi_1$ **performance** |
|---|---|---|
| k-NN | Voting | k-NN |
| Voting | Decision trees | Voting |
| ExNIS | k-NN | Decision trees |
| Decision trees | ExNIS | ExNIS |
| FF NN / RNN | FF NN | FF NN |
| | RNN | RNN |

# Chapter 7

# Conclusion

This thesis aimed to develop and deploy strategies for improving reliability of state estimation for a last-mile mobility vehicle. Based on quantitative and qualitative analysis in different relevant test scenarios, focused on triggering failure as much as possible, it can be concluded that the proposed algorithms successfully reject a significant fraction of the measurements in failure situations while keeping most of the valid measurements to preserve nominal localization performance. In particular, all algorithms are capable of rejecting at least 2/3 of failure measurements encountered in the different test scenarios, while the best-performing approach in this regard reaches a rejection of 3/4 of failure measurements. Most approaches are capable of having valid data losses below the 10%, with the best-performing approach in this regard reaching losses below 5%.

The results indicate that different failure scenarios have different algorithms as their best performers, suggesting that putting together all decisions through a voting system is the best approach in terms of reliability, especially when a general solution is desired, which can be migrated to different vehicles and sensor configurations. This way, a 70% rejection of failure measurements is achieved while keeping a loss of valid information below 5%. When it comes to individual algorithm performance, according to the collected and labeled data, the classical ML approach k-NN ranks top in rejection and overall results, as long as proper manual feature engineering is performed to the input data prior to feeding it to the algorithm (using the general parity relations proposed in previous research).

Across this work, different algorithms have been proposed for the sensor rejection problem prior to potential sensor fusion. Exploration of this problem with such general formulation, without relying on the properties and particularities of the localization sources used, helps to fill a gap in the autonomous vehicle localization literature, in which most of the focus is put in valid information fusion and algorithm-specific rejection strategies, and provides a solution to increase reliability in worst-case scenarios, which are usually the situations that end up making the difference when such autonomous systems want to be launched into the market and accepted by the user community.

Future works following this thesis could consider migrating the proposed solution to a different vehicle, especially one with a different set of sensors / localization sources, which could complement the study of individual algorithm performance and the trade-offs analyses for their parameters, as well as provide further insights on how well the proposed approaches generalize when changing the system and using new components. Another potential follow-up path would be to investigate how to adapt other promising approaches from the broad field of FDI, especially on its model-based branch, in order to further extend the proposed combined decision scheme, and/or to explore other manual features to apply to this problem that could further enhance the performance of classical ML techniques, which has been proven here to heavily rely on this feature engineering step.

# Appendix A

# Code

This appendix contains the code for the data pre-processing and algorithm ROS nodes, together with other helpful auxiliary ROS launch files. This code is also published in executable form with sample `.bag` files on the GitHub repository `https://github.com/pedroreyero/state_estimation`.

## A.1 Data pre-processing

Listing A.1: Data pre-processing node

```
#!/usr/bin/env python

import rospy
from nav_msgs.msg import Odometry
from geometry_msgs.msg import PoseWithCovarianceStamped
from sensor_msgs.msg import NavSatFix
from state_estimation.msg import ProcessedData
import numpy as np
from lltoutm import LLtoUTM
from unwrap import unwrap
from proparker_msgs.msg import TruckState
from tf.transformations import euler_from_quaternion, quaternion_from_euler
from geometry_msgs.msg import Quaternion
from std_srvs.srv import Empty, EmptyResponse
from std_msgs.msg import Float32


class DataProcessingNode:

    def __init__(self):

        rospy.init_node('node_data', anonymous=True)

        self.data_publisher = rospy.Publisher('state_estimation/processed_data', ProcessedData, ↵
        queue_size=1)

        self.ins_sub = rospy.Subscriber('ins_raw/state', TruckState, self.ins_callback)
        self.odom_sub = rospy.Subscriber('odom', Odometry, self.odom_callback)
        self.cartographer_sub = rospy.Subscriber('cartographer/tracked_pose', ↵
        PoseWithCovarianceStamped, self.cartographer_callback)
        self.dragonfly_sub = rospy.Subscriber('dragonfly_manager/tracked_pose', ↵
        PoseWithCovarianceStamped, self.dragonfly_callback)
        self.gps_sub = rospy.Subscriber('gps', NavSatFix, self.gps_callback)

        self.reset_origins_srv = rospy.Service('state_estimation/reset_origins', Empty, self. ↵
        handle_reset_origins)

        self.processed_data = ProcessedData()
```

```
        self.debug_pub = rospy.Publisher('state_estimation/debug', Float32, queue_size=1)

        self.ins_available = False
        self.ins_data = None
        self.odom_available = False
        self.odom_data = None
        self.last_deadr_d = 0
        self.deadr_x = 0
        self.deadr_y = 0
        self.deadr_refth = None
        self.deadr_th = None # prev th for unwrappping
        self.cartographer_available = False
        self.cartographer_data = None
        self.cartographer_turn = None
        self.cartographer_refx = None
        self.cartographer_refy = None
        self.cartographer_refth = None
        self.cartographer_th = None # prev th for unwrappping
        self.dragonfly_available = False
        self.dragonfly_data = None
        self.dragonfly_turn = None
        self.dragonfly_refx = None
        self.dragonfly_refy = None
        self.dragonfly_refth = None
        self.dragonfly_th = None # prev th for unwrappping
        self.gps_available = False
        self.gps_data = None
        self.gps_turn = None
        self.gps_refx = None
        self.gps_refy = None
        self.gps_refth = None
        self.gps_th = None # prev th for unwrappping


    def ins_callback(self, data):

        self.ins_data = data
        self.ins_available = True

        self.check_and_fuse()


    def odom_callback(self, data):

        self.odom_data = data
        self.odom_available = True

        self.check_and_fuse()


    def cartographer_callback(self, data):

        self.cartographer_data = data
        self.cartographer_available = True

        self.check_and_fuse()


    def dragonfly_callback(self, data):

        self.dragonfly_data = data
        self.dragonfly_available = True

        self.check_and_fuse()


    def gps_callback(self, data):
```

```
            self.gps_data = data
            self.gps_available = True

            self.check_and_fuse()


    def handle_reset_origins(self, req):

            #### Make sure data is available
            if (self.ins_data is None) or (self.gps_data is None) or (self.odom_data is None) or \
                            (self.cartographer_data is None) or (self.dragonfly_data is None):
                            rospy.logerr("Reset␣origins␣was␣called␣when␣no␣data␣is␣available␣from␣the␣ ↵
                            required␣topics!!")
                            return EmptyResponse()


            #### If data is indeed available (expected behaviour):

            heading = self.ins_data.heading

            #### FIXED TRANSFORMATION
            ## GPS origin
            UTMNorthing, UTMEasting, UTMZone = LLtoUTM(self.gps_data.latitude, self.gps_data. ↵
            longitude)
            self.gps_turn = heading − np.pi/2
            self.gps_refx = UTMEasting
            self.gps_refy = UTMNorthing
            self.gps_refth = −heading # there is no way to unwrap upon initialization

            ## Dead reckoning "origin"
            self.last_deadr_d = self.odom_data.pose.pose.position.x
            self.deadr_x = 0
            self.deadr_y = 0
            q = self.odom_data.pose.pose.orientation
            eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
            self.deadr_refth = −eul[2] # there is no way to unwrap upon initialization
            self.deadr_th = self.deadr_refth # prev th for unwrappping

            ## Cartographer origin
            q = self.cartographer_data.pose.pose.orientation
            eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
            self.cartographer_turn = − eul[2]
            self.cartographer_refx = self.cartographer_data.pose.pose.position.x
            self.cartographer_refy = self.cartographer_data.pose.pose.position.y
            q = self.cartographer_data.pose.pose.orientation
            eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
            self.cartographer_refth = eul[2] # there is no way to unwrap upon initialization
            self.cartographer_th = self.cartographer_refth # prev th for unwrappping

            ## Dragonfly origin
            q = self.dragonfly_data.pose.pose.orientation
            eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
            self.dragonfly_turn = − eul[2]
            self.dragonfly_refx = self.dragonfly_data.pose.pose.position.x
            self.dragonfly_refy = self.dragonfly_data.pose.pose.position.y
            self.dragonfly_refth = eul[2] # there is no way to unwrap upon initialization
            self.dragonfly_th = self.dragonfly_refth # prev th for unwrappping


            return EmptyResponse()


    def check_and_fuse(self):

            if (self.ins_available and self.odom_available and self.cartographer_available and self. ↵
            dragonfly_available and self.gps_available):
                    # Everything available :)
```

```
#### FIXED FRAME TRANSFORMATIONS
## Process INS
heading = self.ins_data.heading

## Process GPS
UTMNorthing, UTMEasting, UTMZone = LLtoUTM(self.gps_data.latitude, self. ↵
gps_data.longitude)
if self.gps_turn is None:
        self.gps_turn = heading - np.pi/2
        self.gps_refx = UTMEasting
        self.gps_refy = UTMNorthing
        self.gps_refth = -heading # there is no way to unwrap upon initialization
        self.gps_th = self.gps_refth # prev th for unwrappping
mat_turn = np.array([[np.cos(self.gps_turn), -np.sin(self.gps_turn)],[np.sin(self. ↵
gps_turn), np.cos(self.gps_turn)]])
gps_x, gps_y = (np.matmul(mat_turn,np.array([[UTMEasting - self.gps_refx],[ ↵
UTMNorthing-self.gps_refy]]))).ravel()
gps_th = -heading
gps_th = unwrap(gps_th, self.gps_th) - self.gps_refth
self.gps_th = gps_th # prev th for wrapping
CovGPS = np.array([[0.1,0,0],[0,0.1,0],[0,0,0.001]])

## Process Dead reckoning
if self.deadr_refth is None:
        self.last_deadr_d = self.odom_data.pose.pose.position.x
        q = self.odom_data.pose.pose.orientation
        eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
        self.deadr_refth = -eul[2] # there is no way to unwrap upon initialization
        self.deadr_th = self.deadr_refth # prev th for wrapping
deadr_d = self.odom_data.pose.pose.position.x
delta_d = deadr_d - self.last_deadr_d
self.last_deadr_d = deadr_d
CovDR = np.array([[0.2,0,0],[0,0.2,0],[0,0,0.001]])

deadr_turn = -(heading - np.pi/2) + self.gps_turn
mat_turn = np.array([[np.cos(deadr_turn), -np.sin(deadr_turn)],[np.sin(deadr_turn ↵
), np.cos(deadr_turn)]])
corrected = np.matmul(mat_turn, np.array([[delta_d],[0]]))
self.deadr_x += corrected[0,0]
self.deadr_y += corrected[1,0]
deadr_x, deadr_y = self.deadr_x, self.deadr_y
q = self.odom_data.pose.pose.orientation
eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
deadr_th = -eul[2]
deadr_th = unwrap(deadr_th, self.deadr_th) - self.deadr_refth
self.deadr_th = deadr_th # prev th for wrapping

## Process Cartographer
if self.cartographer_turn is None:
        q = self.cartographer_data.pose.pose.orientation
        eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
        self.cartographer_turn = - eul[2]
        self.cartographer_refx = self.cartographer_data.pose.pose.position.x
        self.cartographer_refy = self.cartographer_data.pose.pose.position.y
        self.cartographer_refth = eul[2] # there is no way to unwrap upon ↵
        initialization
        self.cartographer_th = self.cartographer_refth # prev th for wrapping
mat_turn = np.array([[np.cos(self.cartographer_turn), -np.sin(self.cartographer_turn ↵
)], \
                                        [np.sin(self.cartographer_turn), np. ↵
                                        cos(self.cartographer_turn)]])
cartographer_x, cartographer_y = (np.matmul(mat_turn,np.array( \
                        [[self.cartographer_data.pose.pose.position.x - self. ↵
                        cartographer_refx], \
                        [self.cartographer_data.pose.pose.position.y - self. ↵
                        cartographer_refy]]))).ravel()
q = self.cartographer_data.pose.pose.orientation
```

```
eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
cartographer_th = eul[2]
cartographer_th = unwrap(cartographer_th, self.cartographer_th) − self. ↵
cartographer_refth
self.cartographer_th = cartographer_th # prev th for wrapping
CovCG = np.array([[0.01,0,0],[0,0.01,0],[0,0,0.001]])

## Process Dragonfly
if self.dragonfly_turn is None:
        q = self.dragonfly_data.pose.pose.orientation
        eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
        self.dragonfly_turn = − eul[2]
        self.dragonfly_refx = self.dragonfly_data.pose.pose.position.x
        self.dragonfly_refy = self.dragonfly_data.pose.pose.position.y
        self.dragonfly_refth = eul[2] # there is no way to unwrap upon initialization
        self.dragonfly_th = self.dragonfly_refth # prev th for wrapping
mat_turn = np.array([[np.cos(self.dragonfly_turn), −np.sin(self.dragonfly_turn)], \
                                        [np.sin(self.dragonfly_turn), np.cos( ↵
                                        self.dragonfly_turn)]])
dragonfly_x, dragonfly_y = (np.matmul(mat_turn,np.array( \
                        [[self.dragonfly_data.pose.pose.position.x − self. ↵
                        dragonfly_refx], \
                        [self.dragonfly_data.pose.pose.position.y − self. ↵
                        dragonfly_refy]]))).ravel()
q = self.dragonfly_data.pose.pose.orientation
eul = euler_from_quaternion([q.x, q.y, q.z, q.w])
dragonfly_th = eul[2]
dragonfly_th = unwrap(dragonfly_th, self.dragonfly_th) − self.dragonfly_refth
self.dragonfly_th = dragonfly_th # prev th for wrapping
CovDF = np.array([[0.02,0,0],[0,0.02,0],[0,0,0.001]])

## ExNIS parity relations computation
dX = np.array([cartographer_x, gps_x, dragonfly_x, deadr_x])
dY = np.array([cartographer_y, gps_y, dragonfly_y, deadr_y])
dTh = np.array([cartographer_th, gps_th, dragonfly_th, deadr_th])

dXYTh = np.stack((dX,dY,dTh))
I12 = CovCG + CovGPS;
aux = (dXYTh[:,0] − dXYTh[:,1])[np.newaxis]
d12 = np.matmul(np.matmul(aux,np.linalg.inv(I12)),aux.T).squeeze()
I13 = CovCG + CovDF;
aux = (dXYTh[:,0] − dXYTh[:,2])[np.newaxis]
d13 = np.matmul(np.matmul(aux,np.linalg.inv(I13)),aux.T).squeeze()
I14 = CovCG + CovDR;
aux = (dXYTh[:,0] − dXYTh[:,3])[np.newaxis]
d14 = np.matmul(np.matmul(aux,np.linalg.inv(I14)),aux.T).squeeze()
I23 = CovGPS + CovDF;
aux = (dXYTh[:,1] − dXYTh[:,2])[np.newaxis]
d23 = np.matmul(np.matmul(aux,np.linalg.inv(I23)),aux.T).squeeze()
I24 = CovGPS + CovDR;
aux = (dXYTh[:,1] − dXYTh[:,3])[np.newaxis]
d24 = np.matmul(np.matmul(aux,np.linalg.inv(I24)),aux.T).squeeze()
I34 = CovDF + CovDR;
aux = (dXYTh[:,2] − dXYTh[:,3])[np.newaxis]
d34 = np.matmul(np.matmul(aux,np.linalg.inv(I34)),aux.T).squeeze()


## Publish processed data
self.processed_data.stamp = rospy.Time.now()

self.processed_data.cartographer_x = cartographer_x
self.processed_data.cartographer_y = cartographer_y
self.processed_data.cartographer_th = cartographer_th
self.processed_data.cartographer_cov = CovCG.reshape(−1,).tolist()

self.processed_data.gps_x = gps_x
self.processed_data.gps_y = gps_y
self.processed_data.gps_th = gps_th
```

```
                self.processed_data.gps_cov = CovGPS.reshape(-1,).tolist()

                self.processed_data.dragonfly_x = dragonfly_x
                self.processed_data.dragonfly_y = dragonfly_y
                self.processed_data.dragonfly_th = dragonfly_th
                self.processed_data.dragonfly_cov = CovDF.reshape(-1,).tolist()

                self.processed_data.deadr_x = deadr_x
                self.processed_data.deadr_y = deadr_y
                self.processed_data.deadr_th = deadr_th
                self.processed_data.deadr_cov = CovDR.reshape(-1,).tolist()

                self.processed_data.dij = [d12, d13, d14, d23, d24, d34]

                self.data_publisher.publish(self.processed_data)


                #### RESET FLAGS
                self.ins_available = False
                self.odom_available = False
                self.cartographer_available = False
                self.dragonfly_available = False
                self.gps_available = False

                #### DEBUGGING PUBLISHER :)
                self.debug_pub.publish(Float32(gps_th))

            else:
                return


    def main_loop(self):
        while not rospy.is_shutdown():
            rospy.spin()



if __name__ == '__main__':
    state_estimation_node = DataProcessingNode()

    try:
        state_estimation_node.main_loop()
    except rospy.ROSInterruptException:
        pass
```

Listing A.2: UTM projection [38]

```
### -------------------------------------------------------------------
# Convert lat/long to UTM coords. Equations from USGS Bulletin 1532
#
# East Longitudes are positive, West longitudes are negative.
# North latitudes are positive, South latitudes are negative
# Lat and Long are in fractional degrees
#
# Adapted from the code written by Chuck Gantz- chuck.gantz@globalstar.com
# Link to source:
# https://github.com/rosbook/effective_robotics_programming_with_ros/blob/master/chapter8_tutorials/src
/c8_fixtoUTM.cpp
### -------------------------------------------------------------------

import math

def LLtoUTM(Lat, Long):

    RADIANS_PER_DEGREE = math.pi/180.0
    DEGREES_PER_RADIAN = 180.0/math.pi
```

```
### WGS84 Parameters
WGS84_A = 6378137.0 # major axis
WGS84_B = 6356752.31424518 # minor axis
WGS84_F = 0.0033528107 # ellipsoid flattening
WGS84_EP = 0.0820944379 # second eccentricity

WGS84_E = 0.0818191908 # first eccentricity

### UTM Parameters
UTM_K0 = 0.9996 # scale factor
UTM_FE = 500000.0 # false easting
UTM_FN_N = 0.0 # false northing on north hemisphere
UTM_FN_S = 10000000.0 # false northing on south hemisphere
UTM_E2 = (WGS84_E*WGS84_E) # e^2
UTM_E4 = (UTM_E2*UTM_E2) # e^4
UTM_E6 = (UTM_E4*UTM_E2) # e^6
UTM_EP2 = (UTM_E2/(1-UTM_E2)) # e'^2

a = WGS84_A
eccSquared = UTM_E2
k0 = UTM_K0

### Make sure the longitude is between -180.00 .. 179.9
LongTemp = (Long+180)-int((Long+180)/360)*360-180

LatRad = Lat*RADIANS_PER_DEGREE
LongRad = LongTemp*RADIANS_PER_DEGREE

ZoneNumber = int((LongTemp + 180)/6) + 1;

if ( Lat >= 56.0 and Lat < 64.0 and LongTemp >= 3.0 and LongTemp < 12.0 ):
    ZoneNumber = 32
# print( "ZoneNumber: ", ZoneNumber )

### Special zones for Svalbard
if( Lat >= 72.0 and Lat < 84.0 ):
    if( LongTemp >= 0.0 and LongTemp < 9.0 ):
        ZoneNumber = 31
    elif( LongTemp >= 9.0 and LongTemp < 21.0 ):
        ZoneNumber = 33
    elif( LongTemp >= 21.0 and LongTemp < 33.0 ):
            ZoneNumber = 35
    elif( LongTemp >= 33.0 and LongTemp < 42.0 ):
            ZoneNumber = 37
### +3 puts origin in middle of zone
LongOrigin = (ZoneNumber - 1)*6 - 180 + 3
LongOriginRad = LongOrigin * RADIANS_PER_DEGREE
# print( "Letter: ", UTMLetterDesignator(Lat) )
### compute the UTM Zone from the latitude and longitude
UTMZone = str(ZoneNumber) + UTMLetterDesignator(Lat)

eccPrimeSquared = (eccSquared)/(1-eccSquared)
N = a/math.sqrt(1-eccSquared*math.sin(LatRad)*math.sin(LatRad))
T = math.tan(LatRad)*math.tan(LatRad)
C = eccPrimeSquared*math.cos(LatRad)*math.cos(LatRad)
A = math.cos(LatRad)*(LongRad-LongOriginRad)
M = a*((1 - eccSquared/4 - 3*eccSquared*eccSquared/64 - 5*eccSquared*eccSquared*eccSquared/256)* ↵
LatRad \
                    - (3*eccSquared/8 + 3*eccSquared*eccSquared/32 + 45*eccSquared* ↵
                    eccSquared*eccSquared/1024)*math.sin(2*LatRad) \
                                        + (15*eccSquared*eccSquared/256 + ↵
                                         45*eccSquared*eccSquared* ↵
                                        eccSquared/1024)*math.sin(4* ↵
                                        LatRad) \
                                        - (35*eccSquared*eccSquared* ↵
                                        eccSquared/3072)*math.sin(6* ↵
                                        LatRad))
UTMEasting = (k0*N*(A+(1-T+C)*A*A*A/6 \
```

```
                                  + (5-18*T+T*T+72*C-58*eccPrimeSquared)*A*A*A*A*A/120) \
                                  + 500000.0)
    UTMNorthing = (k0*(M+N*math.tan(LatRad)*(A*A/2+(5-T+9*C+4*C*C)*A*A*A*A/24 \
                             + (61-58*T+T*T+600*C-330*eccPrimeSquared)*A*A*A*A*A*A/720)))

    if(Lat < 0):
        UTMNorthing = UTMNorthing + 10000000.0 # 10000000 meter offset for southern hemisphere

    return UTMNorthing, UTMEasting, UTMZone



### -------------------------------------------------------------
# Determine the correct UTM letter designator for the given latitude
#
# @returns 'Z' if latitude is outside the UTM limits of 84N to 80S
#
# Adapted from the code written by Chuck Gantz- chuck.gantz@globalstar.com
### -------------------------------------------------------------

def UTMLetterDesignator(Lat):

    if ((84 >= Lat) and (Lat >= 72)): LetterDesignator = 'X'
    elif ((72 > Lat) and (Lat >= 64)): LetterDesignator = 'W'
    elif ((64 > Lat) and (Lat >= 56)): LetterDesignator = 'V'
    elif ((56 > Lat) and (Lat >= 48)): LetterDesignator = 'U'
    elif ((48 > Lat) and (Lat >= 40)): LetterDesignator = 'T'
    elif ((40 > Lat) and (Lat >= 32)): LetterDesignator = 'S'
    elif ((32 > Lat) and (Lat >= 24)): LetterDesignator = 'R'
    elif ((24 > Lat) and (Lat >= 16)): LetterDesignator = 'Q'
    elif ((16 > Lat) and (Lat >= 8)): LetterDesignator = 'P'
    elif (( 8 > Lat) and (Lat >= 0)): LetterDesignator = 'N'
    elif (( 0 > Lat) and (Lat >= -8)): LetterDesignator = 'M'
    elif ((-8 > Lat) and (Lat >= -16)): LetterDesignator = 'L'
    elif((-16 > Lat) and (Lat >= -24)): LetterDesignator = 'K'
    elif((-24 > Lat) and (Lat >= -32)): LetterDesignator = 'J'
    elif((-32 > Lat) and (Lat >= -40)): LetterDesignator = 'H'
    elif((-40 > Lat) and (Lat >= -48)): LetterDesignator = 'G'
    elif((-48 > Lat) and (Lat >= -56)): LetterDesignator = 'F'
    elif((-56 > Lat) and (Lat >= -64)): LetterDesignator = 'E'
    elif((-64 > Lat) and (Lat >= -72)): LetterDesignator = 'D'
    elif((-72 > Lat) and (Lat >= -80)): LetterDesignator = 'C'
    # 'Z' is an error flag, the Latitude is outside the UTM limits
    else: LetterDesignator = 'Z'

    return LetterDesignator
```

Listing A.3: Angle unwraping function

```
### -------------------------------------------------------------
# new_th = unwrap(th,prev_th) unwraps the radian phase angle th.
# Whenever the jump between consecutive angles is greater than or equal
# to pi radians, unwrap shifts the angles by adding multiples of +/-2pi
# until the jump is less than pi.
#
# Inspired by Matlab's unwrap function:
# https://www.mathworks.com/help/matlab/ref/unwrap.html
### -------------------------------------------------------------

import math

def unwrap(th, prev_th):

        new_th = th
        #print(new_th, "0")
        while (new_th - prev_th) >= math.pi:
                new_th -= 2*math.pi
                #print(new_th, "1")
```

```
    while (new_th - prev_th) <= -math.pi:
        new_th += 2*math.pi
        #print(new_th, "2")

    return new_th
```

## A.2 Algorithm nodes

Listing A.4: ExNIS cross-validation node

```
#!/usr/bin/env python

import rospy
import numpy as np
from state_estimation.msg import ProcessedData, AlgDecision


class ExnisDecisionNode:

    def __init__(self):

        rospy.init_node('node_exnis', anonymous=True)

        self.decision_publisher = rospy.Publisher('state_estimation/exnis_decision', AlgDecision, ↵
        queue_size=1)

        self.data_sub = rospy.Subscriber('state_estimation/processed_data', ProcessedData, self. ↵
        data_callback)

        self.alg_decision = AlgDecision()

        self.v1j = np.zeros((1,3))
        self.v2j = np.zeros((1,3))
        self.v3j = np.zeros((1,3))
        self.v4j = np.zeros((1,3))
        self.k_corr = 1

        self.th1 = 6.25 # 6.25 = chi-square 3-dof 10% prob. of exceeding that value
        self.th2 = 7.82 # 7.82 = chi-square 3-dof 5% prob. of exceeding that value
        self.beta = 0.5 # 0.5 = approx. 2 time steps average (higher weights to recent values)


    def data_callback(self, data):

        ## ExNIS cross-validation

        ### Collect dij from message data
        d1j = np.array([data.dij[0], data.dij[1], data.dij[2]])
        d2j = np.array([data.dij[0], data.dij[3], data.dij[4]])
        d3j = np.array([data.dij[1], data.dij[3], data.dij[5]])
        d4j = np.array([data.dij[2], data.dij[4], data.dij[5]])

        ### EWA
        # Filtering
        self.v1j = self.beta * self.v1j + (1 - self.beta) * d1j
        self.v2j = self.beta * self.v2j + (1 - self.beta) * d2j
        self.v3j = self.beta * self.v3j + (1 - self.beta) * d3j
        self.v4j = self.beta * self.v4j + (1 - self.beta) * d4j
        # Bias correction
        if self.k_corr < 700: # until it has no significant effect
            c1j = self.v1j / (1 - self.beta ** self.k_corr)
            c2j = self.v2j / (1 - self.beta ** self.k_corr)
            c3j = self.v3j / (1 - self.beta ** self.k_corr)
            c4j = self.v4j / (1 - self.beta ** self.k_corr)
            self.k_corr += 1
        else:
            c1j = self.v1j
            c2j = self.v2j
            c3j = self.v3j
            c4j = self.v4j

        ### 2-lvl combined thresholding
        selected1 = np.array([np.sum(c1j <= self.th1) >= 1, np.sum(c2j <= self.th1) >= 1, \
                              np.sum(c3j <= self.th1) >= 1, np.sum(c4j <= self.th1) >= 1])
```

```
        selected2 = np.array([np.sum(c1j <= self.th2) >= 2, np.sum(c2j <= self.th2) >= 2, \
                              np.sum(c3j <= self.th2) >= 2, np.sum(c4j <= self.th2) >= 2])
        selected = np.logical_or(selected1, selected2)

        if not selected.any(): # in case rule discards everyone
            selected = np.array([True, False, False, False]) # rely on Cartographer

        ### Publish topic with algorithm decision
        self.alg_decision.stamp = rospy.Time.now()
        self.alg_decision.selected = selected
        self.decision_publisher.publish(self.alg_decision)


    def main_loop(self):
        while not rospy.is_shutdown():
            rospy.spin()




if __name__ == '__main__':
    state_estimation_node = ExnisDecisionNode()

    try:
        state_estimation_node.main_loop()
    except rospy.ROSInterruptException:
        pass
```

Listing A.5: Decision trees node

```
#!/usr/bin/env python3

import rospy
import rospkg
import numpy as np
import os
import sklearn, joblib
from state_estimation.msg import ProcessedData, AlgDecision


class TreeDecisionNode:

    def __init__(self):

        rospy.init_node('node_tree', anonymous=True)

        self.tree_model = []
        r = rospkg.RosPack()
        for source in ['CG','GPS','DF','DR']:
            path = os.path.join(r.get_path('state_estimation'), "models/tree_" + source + '.sav')
            self.tree_model.append(joblib.load(path))

        self.decision_publisher = rospy.Publisher('state_estimation/tree_decision', AlgDecision, ↵
        queue_size=1)

        self.data_sub = rospy.Subscriber('state_estimation/processed_data', ProcessedData, self. ↵
        data_callback)

        self.alg_decision = AlgDecision()

        self.associated = [[0,1,2], [0,3,4], [1,3,5], [2,4,5]] # dij associated to each source


    def data_callback(self, data):

        ## Decision trees

        ### Tree prediction
```

```
        pred = np.zeros((4))
        for i in range(4):
            input_dij = np.array([data.dij[k] for k in self.associated[i]])
            input_dij = np.expand_dims(input_dij, axis=0)
            pred[i] = self.tree_model[i].predict(input_dij)

        selected = pred.astype(np.bool)

        ### Publish topic with algorithm decision
        self.alg_decision.stamp = rospy.Time.now()
        self.alg_decision.selected = selected
        self.decision_publisher.publish(self.alg_decision)


    def main_loop(self):
        while not rospy.is_shutdown():
            rospy.spin()



if __name__ == '__main__':
    state_estimation_node = TreeDecisionNode()

    try:
        state_estimation_node.main_loop()
    except rospy.ROSInterruptException:
        pass
```

Listing A.6: k-Nearest Neighbors node

```
#!/usr/bin/env python3

import rospy
import rospkg
import numpy as np
import os
import sklearn, joblib
from state_estimation.msg import ProcessedData, AlgDecision


class KNNDecisionNode:

    def __init__(self):

        rospy.init_node('node_knn', anonymous=True)

        self.knn_model = []
        r = rospkg.RosPack()
        for source in ['CG','GPS','DF','DR']:
            path = os.path.join(r.get_path('state_estimation'), "models/knn_" + source + '.sav')
            self.knn_model.append(joblib.load(path))

        self.decision_publisher = rospy.Publisher('state_estimation/knn_decision', AlgDecision, ↵
        queue_size=1)

        self.data_sub = rospy.Subscriber('state_estimation/processed_data', ProcessedData, self. ↵
        data_callback)

        self.alg_decision = AlgDecision()

        self.associated = [[0,1,2], [0,3,4], [1,3,5], [2,4,5]] # dij associated to each source


    def data_callback(self, data):

        ## k-Nearest Neighbors
```

```
        ### Tree prediction
        pred = np.zeros((4))
        for i in range(4):
            input_dij = np.array([data.dij[k] for k in self.associated[i]])
            input_dij = np.expand_dims(input_dij, axis=0)
            pred[i] = self.knn_model[i].predict(input_dij)

        selected = pred.astype(np.bool)

        ### Publish topic with algorithm decision
        self.alg_decision.stamp = rospy.Time.now()
        self.alg_decision.selected = selected
        self.decision_publisher.publish(self.alg_decision)


    def main_loop(self):
        while not rospy.is_shutdown():
            rospy.spin()




if __name__ == '__main__':
    state_estimation_node = KNNDecisionNode()

    try:
        state_estimation_node.main_loop()
    except rospy.ROSInterruptException:
        pass
```

Listing A.7: Feedforward FCNN node

```
#!/usr/bin/env python3

import rospy
import rospkg
import numpy as np
import os
import onnxruntime as rt
from state_estimation.msg import ProcessedData, AlgDecision


class FFNNDecisionNode:

    def __init__(self):

        rospy.init_node('node_ffnn', anonymous=True)

        r = rospkg.RosPack()
        path = os.path.join(r.get_path('state_estimation'), "models/ffnn_model.onnx")
        self.nn_model = rt.InferenceSession(path)

        self.decision_publisher = rospy.Publisher('state_estimation/ffnn_decision', AlgDecision, ↵
        queue_size=1)

        self.data_sub = rospy.Subscriber('state_estimation/processed_data', ProcessedData, self. ↵
        data_callback)

        self.alg_decision = AlgDecision()


    def data_callback(self, data):

        ## Feed-forward FC NN

        ### NN prediction
        nn_input = [data.cartographer_x, data.cartographer_y, data.cartographer_th, \
                        data.gps_x, data.gps_y, data.gps_th, \
```

```
                              data.dragonfly_x, data.dragonfly_y, data.dragonfly_th, \
                              data.deadr_x, data.deadr_y, data.deadr_th]
        nn_input = np.array([nn_input]).astype(np.float32)
        nn_input = np.reshape(nn_input, (1, -1)) # from 1D to 2D array
        input_name = self.nn_model.get_inputs()[0].name
        label_name = self.nn_model.get_outputs()[0].name

        pred = self.nn_model.run([label_name], {input_name: nn_input})[0]
        pred = pred.squeeze() # from 2D to 1D array

        selected = (pred > 0.5)

        ### Publish topic with algorithm decision
        self.alg_decision.stamp = rospy.Time.now()
        self.alg_decision.selected = selected
        self.alg_decision.trust = pred
        self.decision_publisher.publish(self.alg_decision)


    def main_loop(self):
        while not rospy.is_shutdown():
            rospy.spin()




if __name__ == '__main__':
    state_estimation_node = FFNNDecisionNode()

    try:
        state_estimation_node.main_loop()
    except rospy.ROSInterruptException:
        pass
```

Listing A.8: Recurrent NN node

```
#!/usr/bin/env python3

import rospy
import rospkg
import numpy as np
import os
import onnxruntime as rt
from state_estimation.msg import ProcessedData, AlgDecision


class RNNDecisionNode:

    def __init__(self):

        rospy.init_node('node_rnn', anonymous=True)

        r = rospkg.RosPack()
        path = os.path.join(r.get_path('state_estimation'), "models/rnn_model.onnx")
        self.nn_model = rt.InferenceSession(path)

        self.decision_publisher = rospy.Publisher('state_estimation/rnn_decision', AlgDecision, ↵
        queue_size=1)

        self.data_sub = rospy.Subscriber('state_estimation/processed_data', ProcessedData, self. ↵
        data_callback)

        self.alg_decision = AlgDecision()

        Tx = 300
        input_size = self.nn_model.get_inputs()[0].shape[2]
        self.last_inputs = np.zeros((1,Tx,input_size))
```

```
    def data_callback(self, data):

        ## Recurrent NN

        ### NN prediction
        nn_input = [data.cartographer_x, data.cartographer_y, data.cartographer_th, \
                            data.gps_x, data.gps_y, data.gps_th, \
                            data.dragonfly_x, data.dragonfly_y, data.dragonfly_th, \
                            data.deadr_x, data.deadr_y, data.deadr_th]
        self.last_inputs = np.roll(self.last_inputs, -1, axis=1)
        self.last_inputs[:,-1,:] = nn_input
        input_name = self.nn_model.get_inputs()[0].name
        label_name = self.nn_model.get_outputs()[0].name

        pred = self.nn_model.run([label_name], {input_name: self.last_inputs.astype(np.float32)})[0]
        pred = pred[0,-1,:]

        selected = (pred > 0.5)

        ### Publish topic with algorithm decision
        self.alg_decision.stamp = rospy.Time.now()
        self.alg_decision.selected = selected
        self.alg_decision.trust = pred
        self.decision_publisher.publish(self.alg_decision)


    def main_loop(self):
        while not rospy.is_shutdown():
            rospy.spin()



if __name__ == '__main__':
    state_estimation_node = RNNDecisionNode()

    try:
        state_estimation_node.main_loop()
    except rospy.ROSInterruptException:
        pass
```

Listing A.9: Voting node

```
#!/usr/bin/env python

import rospy
import numpy as np
from state_estimation.msg import AlgDecision


class VotingDecisionNode:

    def __init__(self):

        rospy.init_node('node_voting', anonymous=True)

        self.decision_publisher = rospy.Publisher('state_estimation/voting_decision', AlgDecision, ↵
        queue_size=1)

        self.exnis_sub = rospy.Subscriber('state_estimation/exnis_decision', AlgDecision, lambda ↵
        data: self.decision_callback(data,0))
        self.tree_sub = rospy.Subscriber('state_estimation/tree_decision', AlgDecision, lambda data: ↵
         self.decision_callback(data,1))
        self.knn_sub = rospy.Subscriber('state_estimation/knn_decision', AlgDecision, lambda data: ↵
         self.decision_callback(data,2))
        self.ffnn_sub = rospy.Subscriber('state_estimation/ffnn_decision', AlgDecision, lambda data: ↵
         self.decision_callback(data,3))
```

```
        self.rnn_sub = rospy.Subscriber('state_estimation/rnn_decision', AlgDecision, lambda data: ↵
        self.decision_callback(data,4))

        self.alg_decision = AlgDecision()

        n_alg = 5
        n_sources = 4
        self.decisions = np.ones((n_alg,n_sources), dtype = bool)
        self.available = np.zeros(n_alg, dtype = bool)


    def decision_callback(self, data, alg):


        self.decisions[alg,:] = data.selected
        self.available[alg] = True

        self.vote()


    def vote(self):

        ## Voting system (majority)

        if self.available.all():

            ### Compute the majority vote (mode)
            # Numpy doesn't have a mode function, but knowing that labels are either 0 or 1
            # we can compute the mode by just averaging and thresholding by 0.5
            final_decision = np.mean(self.decisions.astype(float), axis=0)
            final_decision = final_decision > 0.5

            # Last-resort source: Cartographer
            if not final_decision.any():
                final_decision = np.array([True, False, False, False])

            ### Publish topic with algorithm decision
            self.alg_decision.stamp = rospy.Time.now()
            self.alg_decision.selected = final_decision
            self.decision_publisher.publish(self.alg_decision)

            ### Reset availability flags
            self.available.fill(False)

        else:
            pass


    def main_loop(self):
        while not rospy.is_shutdown():
            rospy.spin()



if __name__ == '__main__':
    state_estimation_node = VotingDecisionNode()

    try:
        state_estimation_node.main_loop()
    except rospy.ROSInterruptException:
        pass
```

## A.3 Other ROS files

Listing A.10: Overall solution launcher

```
<launch>
  <!-- State estimation -->
  <!-- 1) Data processing -->
  <node name="node_data" pkg="state_estimation" type="node_data.py" required="true"/>
  <!-- 2) Individual sensor rejection algorithms -->
  <node name="node_exnis" pkg="state_estimation" type="node_exnis.py" required="true"/>
  <node name="node_tree" pkg="state_estimation" type="node_tree.py" required="true"/>
  <node name="node_knn" pkg="state_estimation" type="node_knn.py" required="true"/>
  <node name="node_ffnn" pkg="state_estimation" type="node_ffnn.py" required="true"/>
  <node name="node_rnn" pkg="state_estimation" type="node_rnn.py" required="true"/>
  <!-- 3) Voting system (putting together all algorithms) -->
  <node name="node_voting" pkg="state_estimation" type="node_voting.py" required="true"/>

  <!-- rqt (optional) -->
  <!--node name="rqt_gui" pkg="rqt_gui" type="rqt_gui"/-->

</launch>
```

Listing A.11: Data recording launcher

```
<launch>

  <!-- rosbag record -->
  <node name="bag_record" pkg="rosbag" type="record"
        args="-o /home/$(env USER)/Downloads/ /rosout /tf /odom /gps /gps_odom /cartographer/ ↵
        tracked_pose /dragonfly_manager/tracked_pose /ins_raw/state /vehicle_speed /reverse_travel / ↵
        state_estimation/fused_pose /tf_static /tf2_web_republisher/status /state_estimation/ ↵
        processed_data /state_estimation/exnis_decision /state_estimation/tree_decision / ↵
        state_estimation/knn_decision /state_estimation/ffnn_decision /state_estimation/rnn_decision / ↵
        state_estimation/voting_decision"/>

</launch>
```

# Bibliography


[1] Z. Wang, Y. Wu, and Q. Niu, "Multi-sensor fusion in automated driving: A survey," *IEEE Access*, vol. 8, pp. 2847–2868, 2020.

[2] Z. Chong, B. Qin, T. Bandyopadhyay, T. Wongpiromsarn, E. Rankin, M. A. Jr., E. Frazzoli, D. Rus, D. Hsu, and K. Low, "Autonomous personal vehicle for the first- and last-mile transportation services," 2011.

[3] Q. Li, J. P. Queralta, T. N. Gia, Z. Zou, and T. Westerlund, "Multi-sensor fusion for navigation and mapping in autonomous vehicles: Accurate localization in urban environments," *Unmanned Systems*, vol. 08, no. 03, pp. 229–237, 2020.

[4] B. Li, S. Liu, J. Tang, J.-L. Gaudiot, L. Zhang, and Q. Kong, "Autonomous Last-mile Delivery Vehicles in Complex Traffic Environments," *arXiv e-prints*, 2020.

[5] O. Cohen and Y. Edan, "A sensor fusion framework for on-line sensor and algorithm selection," in *Proceedings of the 2005 IEEE International Conference on Robotics and Automation*, 2005, pp. 3155–3161.

[6] Y. Lu, E. Collins, and M. F. Selekwa, "Parity relation based fault detection, isolation and reconfiguration for autonomous ground vehicle localization sensors," 2004.

[7] L. Wei, C. Cappelle, and Y. Ruichek, "Camera/laser/gps fusion method for vehicle positioning under extended nis-based sensor validation," *IEEE Transactions on Instrumentation and Measurement*, vol. 62, no. 11, pp. 3110–3122, 2013.

[8] J. A. Farrell and P. F. Roysdon, "Advanced vehicle state estimation: A tutorial and comparative study," *20th IFAC World Congress*, vol. 50, no. 1, pp. 15 971–15 976, 2017.

[9] W. Rahiman and Z. Zainal, "An overview of development gps navigation for autonomous car," in *2013 IEEE 8th Conference on Industrial Electronics and Applications (ICIEA)*, 2013, pp. 1112–1118.

[10] M. Schmandt, *GIS Commons*, ch. 2. [Online]. Available: https://giscommons.org/chapter-2-input/

[11] U. I. Bhatti and W. Y. Ochieng, "Failure modes and models for integrated gps/ins systems," *Journal of Navigation*, vol. 60, no. 2, p. 327–348, 2007.

[12] T. Kos, I. Markezic, and J. Pokrajcic, "Effects of multipath reception on gps positioning performance," in *Proceedings ELMAR-2010*, 2010, pp. 399–402.

[13] L. Teschler, "Inertial measurement units will keep self-driving cars on track," *Microcontroller Tips*, 2018. [Online]. Available: https://www.microcontrollertips.com/inertial-measurement-units-will-keep-self-driving-cars-on-track-faq/

[14] J. Kocić, N. Jovičić, and V. Drndarević, "Sensors and sensor fusion in autonomous vehicles," in *2018 26th Telecommunications Forum (TELFOR)*, 2018, pp. 420–425.

[15] C. Rablau, "Lidar: a new self-driving vehicle for introducing optics to broader engineering and non-engineering audiences," in *Fifteenth Conference on Education and Training in Optics and Photonics: ETOP 2019*, A.-S. Poulin-Girard and J. A. Shaw, Eds., International Society for Optics and Photonics. SPIE, 2019, pp. 84 – 97.

[16] M. Khader and S. Cherian, "An introduction to automotive lidar," *Texas Instruments*, 2020. [Online]. Available: https://www.ti.com/lit/wp/slyy150a/slyy150a.pdf

[17] C. de Castro Bonfim, "Reliability and safety improving methods for the evaluation of driving environment information multiple fault and traget tracking for radar and lidar based sensors," Ph.D. dissertation, 2009.

[18] Z. Alsayed, G. Bresson, A. Verroust-Blondet, and F. Nashashibi, "Failure detection for laser-based slam in urban and peri-urban environments," in *2017 IEEE 20th International Conference on Intelligent Transportation Systems (ITSC)*, 2017, pp. 1–7.

[19] P. Merriaux, Y. Dupuis, R. Boutteau, P. Vasseur, and X. Savatier, "Lidar point clouds correction acquired from a moving car based on can-bus data," *ArXiv*, vol. abs/1706.05886, 2017.

[20] F. Rosique, P. J.Navarro, C. Fernández, and A. Padilla, "A systematic review of perception system and simulators for autonomous vehicles research," *Sensors*, vol. 19, no. 3, 2019. [Online]. Available: https://www.mdpi.com/1424-8220/19/3/648

[21] Z. Yan, L. Sun, T. Krajník, and Y. Ruichek, "Eu long-term dataset with multiple sensors for autonomous driving," in *2020 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2020, pp. 10 697–10 704.

[22] N. Kehtarnavaz, N. C. Griswold, and J. K. Eem, "Comparison of mono- and stereo-camera systems for autonomous vehicle tracking," in *Applications of Artificial Intelligence IX*, M. M. Trivedi, Ed., vol. 1468, International Society for Optics and Photonics. SPIE, 1991, pp. 467 – 478.

[23] M. O. A. Aqel, M. H. Marhaban, M. I. Saripan, and N. B. Ismail, "Review of visual odometry: types, approaches, challenges, and applications," *SpringerPlus*, vol. 5, no. 1, oct 2016.

[24] T. Taketomi, H. Uchiyama, and S. Ikeda, "Visual slam algorithms: A survey from 2010 to 2016," *IPSJ Transactions on Computer Vision and Applications*, vol. 9, 2017.

[25] "Can i use dragonfly outdoors?" Apr 2021. [Online]. Available: http://dragonflycv.com/support/knowledge-base/can-dragonfly-be-used-outdoors/

[26] K. Kotay, "Odometry," 2001. [Online]. Available: https://groups.csail.mit.edu/drl/courses/cs54-2001s/odometry.html

[27] M. Brossard and S. Bonnabel, "Learning wheel odometry and imu errors for localization," in *2019 International Conference on Robotics and Automation (ICRA)*, 2019, pp. 291–297.

[28] J. Borenstein, "Experimental results from internal odometry error correction with the omnimate mobile robot," *IEEE Transactions on Robotics and Automation*, vol. 14, no. 6, pp. 963–969, 1998.

[29] A. Martinelli, "Modeling and estimating the odometry error of a mobile robot," *IFAC Proceedings Volumes*, vol. 34, no. 6, pp. 407–412, 2001, 5th IFAC Symposium on Nonlinear Control Systems 2001, St Petersburg, Russia, 4-6 July 2001.

[30] M. Holder, S. Hellwig, and H. Winner, "Real-time pose graph slam based on radar," in *2019 IEEE Intelligent Vehicles Symposium (IV)*, 2019, pp. 1145–1151.

[31] Z. Hong, Y. Petillot, and S. Wang, "Radarslam: Radar based large-scale slam in all weathers," 2020.

[32] K. Siantidis, "Side scan sonar based onboard slam system for autonomous underwater vehicles," in *2016 IEEE/OES Autonomous Underwater Vehicles (AUV)*, 2016, pp. 195–200.

[33] F. Demim, A. Nemra, H. Abdelkadri, A. Bazoula, K. Louadj, and M. Hamerlain, "Slam problem for autonomous underwater vehicle using svsf filter," in *2018 25th International Conference on Systems, Signals and Image Processing (IWSSIP)*, 2018, pp. 1–5.

[34] Google, "Cartographer." [Online]. Available: https://opensource.google/projects/cartographer

[35] ——, "Cartographer ros integration." [Online]. Available: https://google-cartographer-ros.readthedocs.io/en/latest/

[36] W. Hess, D. Kohler, H. Rapp, and D. Andor, "Real-time loop closure in 2d lidar slam," in *2016 IEEE International Conference on Robotics and Automation (ICRA)*, 2016, pp. 1271–1278.

[37] A. Mahtani, L. Sanchez, E. Fernandez, and A. Martinez, *Effective Robotics Programming with ROS*, 3rd ed. Packt Publishing, 2016, ch. 8.

[38] Geokov, "Utm - universal transverse mercator." [Online]. Available: http://geokov.com/education/utm.aspx

[39] E. Fernandez Perdomo, "Latitude-longitude to utm," Dec 2016. [Online]. Available: https://github.com/rosbook/effective_robotics_programming_with_ros/blob/master/chapter8_tutorials/src/c8_fixtoUTM.cpp

[40] Onit, "Dragonfly." [Online]. Available: https://dragonflycv.com/support/documentation/introduction/

[41] "Robot operating system." [Online]. Available: http://wiki.ros.org/ROS/Introduction

[42] I. Samy and D.-W. Gu, *Fault Detection and Isolation (FDI)*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 5–17.

[43] S. X. Ding, *Model-Based Fault Diagnosis Techniques: Design Schemes, Algorithms, and Tools*, 1st ed. Springer Publishing Company, Incorporated, 2008.

[44] Y. Bar-Shalom, X. Li, and T. Kirubarajan, "Estimation with applications to tracking and navigation: Theory, algorithms and software." Wiley, 2001.

[45] F. Gustafsson, *Adaptive Filtering and Change Detection*. Wiley, 2000.

[46] S. Grigorescu, B. Trasnea, T. Cocias, and G. Macesanu, "A survey of deep learning techniques for autonomous driving," *Journal of Field Robotics*, vol. 37, no. 3, p. 362–386, Apr 2020.

[47] J. Fürnkranz, *Decision Tree*. Boston, MA: Springer US, 2017, pp. 330–335.

[48] E. Keogh, *Instance-Based Learning*. Boston, MA: Springer US, 2017, pp. 672–673.

[49] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2017.

ETH zürich

Institute for Dynamic Systems and Control

Prof. Dr. R. D'Andrea, Prof. Dr. E. Frazzoli, Prof. Dr. Lino Guzzella, Prof. Dr. C. Onder, Prof. Dr. M. Zeilinger

**Title of work:**

Sensor rejection for reliable state estimation of an autonomous last-mile delivery vehicle

**Thesis type and date:**

Master's Thesis, July 2021

**Supervision:**

Dr. Erik Wilhelm
Prof. Dr. Christopher Onder

**Student:**

| Name: | Pedro Reyero Santiago |
|---|---|
| E-mail: | preyero@student.ethz.ch |
| Legi-Nr.: | 20-909-115 |
| Semester: | FS 2021 |

**Statement regarding plagiarism:**

By signing this statement, I affirm that I have read and signed the Declaration of Originality, independently produced this paper, and adhered to the general practice of source citation in this subject-area.

Declaration of Originality:

`https://www.ethz.ch/content/dam/ethz/main/education/rechtliches-abschluesse/leistungskontrollen/declaration-originality.pdf`

Zurich, 16.7.2021: ____________________