POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Seamless Indoor-Outdoor Autonomous Navigation for Unmanned Ground Vehicles

Supervisors

Prof. Marcello CHIABERGE

Candidate

Andrea OSTUNI

April 2022

Abstract

Precision Agriculture (PA) concept has been gaining popularity in the last few years, and it is seen as a possible solution to meet the needs of the growing world population. New techniques and approaches are being developed; these innovations are meant to increase productivity and maximize profit while being a foundation of sustainable agriculture. Autonomous robots have a crucial role in this application. In particular, this work is intended to investigate the problem of Autonomous Navigation to provide a solution of seamless Indoor and Outdoor Navigation. An example of employment should be the forage distribution in large farm stables, where the recharge point is usually outside the building. The Navigation problem and solution for the two environments differ. For the Outdoor navigation, a combination of Global Navigation Satellite System (GNSS) and Inertial Measurement Unit supplies enough data for estimating the position and orientation. In Indoor scenarios, the vehicle needs a static map and a range sensor (LiDaR) data to localize itself. Robot Operating System's (ROS) tools, as services and messages, have been used as a framework to develop and run the application due to its modularity and simple interfaces. In order to obtain good performances, the sensor data are processed by two different filters: a Particle Filter and an Extended Kalman Filter. The first is the foundation of the Adaptive Monte Carlo Localization (AMCL) utilized in indoor environments, while the second is for outdoor application. Then, to navigate seamlessly, a switching algorithm selects which pose estimate to use from the two filters. The feasibility and accuracy of this approach have been tested through several simulations and then deployed on a real rover. The results of these experiments are illustrated in the last chapters of this work.

Table of Contents

List of Figures

Chapter 1

Introduction

New technologies are exploited to improve profitability and reduce the impact of agriculture on the environment; they are grouped in the so-called agriculture of precision. This new system is based on the information and science-based decision tools. Precision agriculture is based on observing, measuring, and responding to crop inter and intra-field variability. The research on precision agriculture aims to define a decision support system for managing the entire farm to optimize the returns on inputs while conserving resources. These technologies do not replace the farmer's essential role in the decision process. At the foundation of precision farming there is GPS and GNSS technologies; these allow the creation of maps of all measurable variables like the characteristics of the terrain from an organic and chemical point of view. "GNSS has provided easy ways to configure autonomous vehicles or navigation systems to assist drivers in outdoor environments, especially in agriculture" [1]. GPS receivers provide location and hour: precise data to create a map, that machines also can communicate their status to owners or managers who might be miles away [2], wirelessly transmitting data, about the robot state without the need for human presence. Such information can monitor machine conditions and diagnose any mechanical problems to make adjustments quickly. An unmanned ground vehicle (UGV), must rely on other critical technologies that need to be incorporated, "such as the safety systems responsible for detecting obstacles in the robots' path and safeguarding humans and animals in the robots'

surroundings well as preventing collisions with obstacles or other robots"[1]using LiDAR and other technologies.

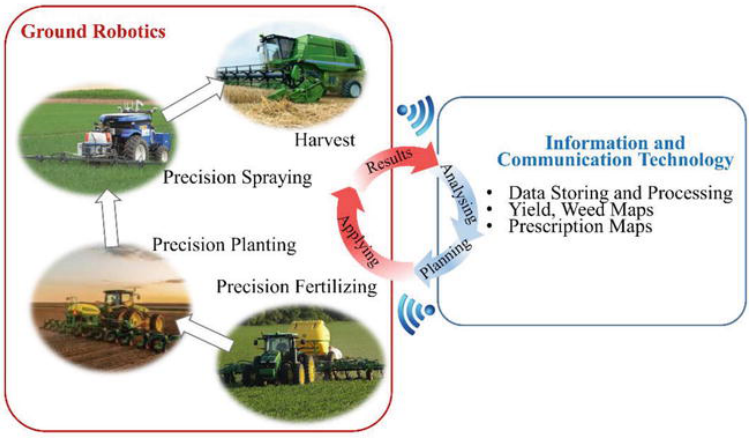

Figure 1.1: Examples of UGV in precision agriculture [2]

1.1 Navigation in different environments

A central issue in PA is mobile robot navigation and their interaction with the environment. An example is the transport of forage in large stables; on this occasion the robot has to navigate, take actions, and make decisions while localizing itself in a map or an outdoor environment. Mobile robot navigation is one of the main fields of research in robotics, there are many solutions to handle this problem, and better ones have been presented in the last years. In particular, "Autonomous mobile robots are robots which can perform desired tasks in structured or unstructured environments without continuous human guidance" as stated by Amitava Chatterjee in [3] . Moreover, "An autonomous mobile robot may also learn or gain new capabilities like adjusting strategies for accomplishing its task(s) or adapting to changing surroundings" [3]. "Where am I? Where are other places relative to me? How do I get to other places from here?" are the questions that define the navigation of a mobile robot [4]. The solution to these questions are the functions that a navigation application must provide and are articulated in:

- Localization

- Path planning
- Path-following and obstacle avoidance

An essential task in developing autonomous robot navigation applications is identifying and modeling the environment. "It is through these environment models that the robot can adapt its decisions to the current state of the world" [5] An important distinction has to be made between indoor and outdoor navigation because the problems and technologies applied to their solution differ; this makes seamless navigation a more demanding challenge because it has to handle both the difficulties.

1.2 Indoor navigation problem

Initially, studies focused only on representations of indoor environments, which are highly structured; in fact, they contain primarily linear structures such as lines and planes. For indoor navigation, two domains of representations of the surroundings have been identified:

- Geometrical representation
- Topological Maps

Each of these models has its advantages; in this work, Occupancy Grid maps will be utilized with cost maps for path planning and localization purposes.

1.2.1 Localization in Indoor navigation

"Localization is the process of establishing correspondence between the map coordinate system and the robot's local coordinate system," as written by Thurn [6]. Still, the difficulty of this problem is in the pose estimation, "most robots do not possess a noise-free sensor for measuring pose" [6], so it has to be inferred from sensor data. In indoor circumstances, the robot has to extract its pose estimate from a combination of sensor data. The existing methods include Inertial Navigation Systems (INS), Wi-Fi indoor localization, Ultra Wide Band (UWB) positioning,

and Light Detection and Ranging (LiDaR) localization as described by Ningbo Li [7]. As described later in this work, a combination of Lidar and IMU coupled with an occupancy grid map of the environment will be used to estimate the vehicle's configuration in these circumstances.

1.3 Outdoor navigation problem

"With continued progress on sensing, and on mechanical and controls aspects of mobile robots systems, it became possible to develop mobile robot systems for operations in unstructured, natural terrain" [5]. An example task for a mobile robot in an outdoor domain is to go from point A to point B using only visual landmarks or GPS waypoints. During the execution of this task, the robot has to decide whether there is a drivable path and compute it.

1.3.1 Localization in Outdoor navigation

To accomplish the localization task "Outdoor vehicles can use precision GPS, inertial measurement units, and wheel odometry to keep track of their position and orientation, typically with an ex- tended Kalman filter" as said in [5]. This method is now a standard and low-cost method that compensates for the disadvantages of single sensors. As noted in another chapter of [5], GPS offers accurate positioning information but does not solve the issue of pose estimation entirely since it does not provide any direct information about vehicle orientation. Other sensors are needed to determine the vehicle's direction, such as compasses, gyrocompasses, and IMU's data. Furthermore, "GPS receivers are generally unable to provide continuous independent estimates of position" [5] since these estimates arrive only with a particular frequency, and especially for inexpensive receivers comes with considerable delays.

1.3.2 Perception in Outdoor navigation

In the last years, the perception of outdoor environments has made significant developments. For example, we can have 3-D point clouds reconstructions of

the environments obtained by laser or depth camera images. Other challenges in outdoor perception for navigation are rough terrains with no extensive ground plane to characterize the driveability and obstacles or the pliable vegetation that may appear as an obstacle in range images. "Negative obstacles (ditches and cliffs) are difficult to detect with range information" [5] these types of obstructions may also create slippage or breaks if not handled correctly.

Chapter 2

State of the art in Robot Navigation

This chapter introduces the advances and the solution developed to handle the navigation problem.

2.1 Localization

This section is a survey on the techniques and technologies adopted for localization. "Mobile robot localization is the problem of determining the pose of a robot relative to a given map of the environment. It is often called positione stimation" [6]. Nearly all mobile robot tasks need the estimate of its self-location and perception of the relative position of the objects with which it has to interact. According to the circumstances, the localization task can be dealt with by two different approaches:

- Map-based localization
- Simultaneous localization and mapping (SLAM)

We talk of map-based or pure localization when the knowledge of the surrounding is provided a priori from a map. In the other case, the whereabouts and surrounding environment of the mobile robot is unknown, so it has to construct a map of the

environment and localize itself in this model. During the development of this thesis, for the outdoor circumstances, a geo-referenced map is assumed as given; instead, for indoor localization, a map will be constructed (also using SLAM algorithms) and then used for pure localization tasks. Either instruments and techniques for these two approaches will be presented and analyzed, focusing on the pure localization methods.

2.1.1 Taxonomy on localization problems

The localization problems are not all the same, and we can have different classification types with varying discrimination criteria. The most common criteria to classify the localization problem are:

- The prior knowledge of the initial condition (and during execution)
- Type of the environment
- Type of approach
- Number of robots involved in the localization process

Prior knowledge classification

The first example of classification is according to the prior knowledge of the UGV state or during its task execution. In particular, these problems are subdivided in:

- **Position tracking**: it is the task to localize the robot correctly when the initial configuration is known, and it has to take account of the motion noise and report as output the most likely pose. Since the noise is slight and it is confined in a small area, it is also called local localization.
- **Global localization**: in this occasion, we have no prior knowledge of the whereabouts of the robot, and no assumption of the probability distribution of the configuration can be made; it is a more general problem than position tracking, and this thesis will investigate ways to handle this problem in indoor and outdoor scenarios.

- **Kidnapped robot problem**: the last and more challenging problem is correctly localizing and relocalizing after the vehicle has been teleported to another location. It is the issue of recovering a correct estimate of the configuration after that robot believes as the actual pose a faulty one.

Type of environment classification

In this subsection, localization will be classified by the surrounding characteristics. In particular, the main categories are:

- A **static environment** is an environment where all the objects and features of the environment are immutables, whether the robot's configuration is the only variable. It is an ideal model of the robot's surroundings (not achievable in reality), with interesting properties that can be exploited for estimation. In this work, it is one of the basic assumptions.
- A **dynamic environment** is a model of the environment more affine to the real world. The robot's pose and the object's configuration may change over time; an example of those objects are people who move in the robot's same environment. This condition makes the localization problem more complex and may require new structures and computations to solve.

Other Classifications

There are other ways to classify the localization problem. We can distinguish whether the localization is active or passive; in the active localization, the application directly controls the robot to implement strategies to estimate the configuration better. Lastly, there is a distinction between localization strategies that use a single platform versus algorithms that utilize multiple robots that can recognize each other and use this information to estimate the fleet configuration better.

2.1.2 Sensor for localization

The robot can interact with the environment using its sensor and actuators. With the help of sensors, a robot can acquire information about the environment. Since

this process is noisy, a robot can only build its internal belief of its state and surroundings without absolute knowledge of the state. As a first classification between sensors is the differentiation between proprioceptive sensors and exteroceptive sensors:

- **Proprioceptive sensors** "are used to measure the internal state of a robot"[8], they may include sensors that measure the position of different joints of a robot, the internal temperature, the effort on an end effector, and so on. This sensor type helps the robot estimate and control its state. In the case of UGV, they are used to estimate the position, steering angle, and robot orientation.
- **Exteroceptive sensors** can be used to gain information about the robot's surroundings; for example, they can determine the distance to objects or the efforts exchanged between the robot and the environment. It is a more challenging problem to define a model of the surroundings than a robot's internal state.

Another classification can be made over the interaction with the environment:

- **Active sensors**: as stated in [8] "an active sensor is one that emits energy into the environment, and measures properties of the environment based on the response." This type of sensor is generally more robust because it is less dependent on the characteristics of the environment.
- **Passive sensors**: comprehend all sensors which are not active and are usually used to monitor the physical properties of the robot.

Some of the most helpful sensors in navigation and localization are listed in the following table:

Encoders

Encoders are usually classified into two main types incremental and absolute.

- Absolute encoders: are used to determine the absolute position of the shaft or wheel within one revolution. It is usually used for UGV steering angle measure.

They are made so that each disk fraction is mapped to a binary representation of the encoder position. The most common type is the absolute optical encoder, which "consists of an optical-glass disk on which concentric circles (tracks) are disposed; each track has an alternating sequence of transparent sectors and matte sectors" [9] and a light source and a light detector. An array of light beams is emitted during work, and photodiodes register the pattern that results after passing through the disk. Since each design is unique per wheel rotation, they encode a single digital signal; they can be used to determine the angle between the heading direction of the robot and the front wheels (steering angle). The resolution of this type of encoder depends on the number of tracks and the gear ratio present between axle and encoder.

- Incremental encoders are much simpler than absolute encoders and thus cheaper. As reported in [9] "incremental encoder consists of an optical disk on which two tracks are disposed, whose transparent and matte sectors (in equal number on the two tracks) are mutually in quadrature"; the presence of the second track is to determine the sign of rotation of the encoder. This sensor outputs two square waves from detecting the light beams that pass through the transparent part of the disk. From these signals, we can evaluate the relative position of the joint by counting the peaks of the square wave, while the velocity can be estimated in different ways:

  – measuring the frequency of the pulse train.

  – measuring the sampling time of the pulse train.

  Between these last two techniques, the former is suitable for high-speed measurements while the latter is suitable for low-speed measurements.

IMU Package

A UGV is often equipped with an inertial measurement unit (IMU) that includes gyroscopes and accelerometers to estimate its translational and rotational motion. It utilizes accelerometers that "are sensitive to all types of acceleration, which implies that both translation motion and rotation (centripetal forces) are measured

in combination" [10]. It is possible to reconstruct the motion signals and the relative configuration to an initial reference frame through single and double integration. "In addition to maintaining a 6-DOF pose of the vehicle, commercial IMUs also typically maintain estimates of velocity and acceleration and can usually provide a diverse set of lower-level measurements as well" as written in [10] and these pieces of information are helpful in navigation. An IMU is composed of three orthogonal gyroscopes and accelerometers in many applications, as shown in the following figure. While the gyroscopes are used to keep an approximate estimate of the vehicle's orientation, the accelerometers are used to measure the instantaneous acceleration of the robot. These data are then filtered to remove the gravitational acceleration from the others and then integrated to obtain the velocity and integrated again to get the position r. Since the integrator is an unstable system, IMUs are "extremely sensitive to measurement errors in the underlying gyroscopes and accelerometers" [10] these errors in measurements can lead to a quadratic mistake in the position estimate. Sensor fusion techniques are adopted to correct these issues and achieve better estimates; these will be better described in the next section.

Ranging

GPS

The global positioning system (GPS) "is a satellite-based radionavigation system owned by the United States government and operated by the United States Space Force"[11]. This is only one of the technologies used for providing geolocalization information to the GPS receiver, in fact the Global navigation satellite systems (GNSS) is composed by for constellation of artificial satellites:

- GPS : is the United States positioning system which cosists of 32 satellites.
- GLONASS: is the Russian positioning system which consists of 24 satellites
- BeiDou: is the Chinese positioning system which consists of 23 satellites
- Galileo: is the European positioning system which consists of 24 satellites

"GPS is the single most commonly used mechanism for location estimation" [10]. GPS is based on received radio signals transmitted by an ensemble of satellites orbiting the earth. The most common GPS localization system is based on the NAVSTAR satellite system, this network is maintained and designed by the United States. the term GPS should only be used when we are referring to the NAVSTAR system although it is often used to refer to any GNSS system. The "GPS satellite network is based on a base constellation of 24 orbiting satellites"[10] but there are also six supplementary additional satellites. "The satellites should provide 99.9% coverage at the worst-covered location in any 24 hour interval, and the signal should be at least 83.92% available at the worst place on earth on the worst day over a 30 day measurement interval" [10]. GPS accuracy is influenced by several factors: environmental conditions, obstacle interaction, satellite transmission accuracy, and these are often uncontrollable. The GPS signal may not always be reliable. Two types of errors are then classified: the “minor failures” that have a limited impact with an error not exceeding 150 meters; and the “main failures” that lead to greater errors or excessive loads of data processing in the receiver.

It’s established that satellites that are directly overhead provide better signal than those near the horizon. Moreover since GPS position estimation is based on the differential signal coming from various satellites, "it is best if the satellites used in the GPS computation are widely spaced in the sky" [10].

GPS signals are transmitted in the microwave band, so they can pass through plastic and glass but are absorbed by water and are reflected by different materials, so GPS is not reliable if you are in forests, in cars or boats and ships. Nevertheless, at any time, the minimum number of satellites operating is 24, 8 of which overlooking the Earth’s surface so that even unsuitable weather conditions can be tolerated.

2.1.3 Probabilistic techniques for state estimation

For localizing an autonomous vehicle, it is necessary to solve the problem of estimating the robot’s state. In fact, "State estimation addresses the problem of estimating quantities from sensor data that are not directly observable, but that can be inferred" as written in [12] and to do so, we can utilize different data coming from

sensors of different nature. Robot state variables are not directly measurable by sensors but only carry a part of the information that characterizes these quantities. Moreover, sensors measurements are characterized by noise. "State estimation seeks to recover state variables from the data" [12], and in this section, the most known probabilistic techniques to compute an estimate of the state will be illustrated. At first, we will explore the probabilistic algorithms and late their applications.

2.1.4 System representation

A mathematical model can sufficiently describe the robot and its interaction with the environment. Each physical quantity is expressed through the evolution of a variable of interest. These variables can be static (a landmark position) or dynamic like the UGV pose and velocity. For autonomous robot localization, the state is usually composed by:

- Robot pose: comprises both the position and the orientation of the UGV in the environment. It is defined by six variables $(x, y, z, roll, pitch, yaw)$ in three-dimensional spaces. In two dimensions, it has planar coordinates for position and yaw to describe the orientation.
- Robot velocity: they describe the speed for each of the pose variables.
- Location of features in the environment: if a static map is assumed, they represent the location of the landmarks and obstacles.

We will identify with x_t the state of the robot and will comprehend all the above features. A robot uses control actions to interact with the environment and change its state. To keep track of these changes and model them in the mathematical framework, they are called control actions and may be represented by the velocity commands sent to the motors or the information received by odometers. Control action will be denoted as u_t. To represent the measurements of the sensor z_t notation will be used. A probabilistic framework will be used for modeling our system and capturing the uncertainness of control actions and the stochasticity of the robot environments. The evolution of the system from the state x_{t-1} to x_t is a

probabilistic law with the following form: Also the measurement process will be represented by conditional probability distributions: From these equations, we got two resulting conditional probability distributions:

- State transition probability:
- Measurement probability:

The concept of belief is now introduced to define the robot's knowledge about its state. As illustrated in [12], belief is a conditional probability distribution that "assigns a probability (or density value) to each possible hypothesis with regards to the true state." The belief over the state will be denoted as $bel(x_t)$: which can be split into two steps, prediction and correction:

- prediction
- correction

In order to calculate these beliefs, the Bayes filter is introduced in the next section.

2.1.5 Bayes Filter

The Bayes filter is the most general algorithm to update the internal belief of the robot recursively. This algorithm computes the posterior belief distribution after using the state transition probability to predict the subsequent state distribution and correct it through measurement data.

Algorithm 1 The general algorithm for Bayes filtering[12].

1: **procedure** ALGORITHM BAYES_FILTER($bel(x_{t-1})$, u_t, z_t)
2: **for** all x_t **do**
3: $bel(x_t) = \int p(x_t \mid u_t,\ x_{t-1})\ bel(x_{t-1}) dx_{t-1}$
4: $bel(x_t) = \eta\ p(z_t \mid x_t)\ \bar{bel}(x_t)$
5: **end for**
6: return $bel(x_t)$
7: **end procedure**

As we can note from the table, the inputs of this algorithm are:

- The belief at the previous time step.
- The measurements from the sensor.
- The control data from encoders or the input command.

The output is instead the new posterior belief at time t. The algorithm only describes which is the update rule for each possible state x_t. The update is separated into two steps. The first is the prediction step and consists of an integral of the product of two distributions; in this way, it is applied to the belief of each prior distribution the motion update caused by the input u_t. The second step is the measurement update and is the product and normalization of two distributions; it incorporates the probability of taking a measurement z_t to the prediction belief *bel*.

"Bayes filters are implemented in several different ways" [12], in fact, different techniques and algorithms are instances of the Bayes filter under various assumptions. According to the initial hypothesis on the state transition and measurement probability distribution, particular algorithms are more accurate or efficient in computation. In the following sections, the most used techniques of the Bayes filter will be presented with their characteristics.

2.1.6 Gaussian Filters

This section describes the instance of Bayes filter called Gaussian filters. Gaussian filters are implementations of the Bayes filter for continuous spaces and are the more popular ones. At the foundation of the Gaussian filter, there is the assumption that multivariate normal distributions represent all the beliefs and transformations.

Kalman Filter

According to statistics and control theory, Kalman filtering, also known as linear quadratic estimation (LQE), is an algorithm that applies a series of measurements observed over time to produce estimates of unknown variables that are typically more accurate than estimates based on a single measurement alone, by estimating

a joint probability distribution over the variables over each timeframe. This filter is named after Rudolf E. Kálmán, who was one of the main developers of its theory[13].

The Kalman filter can be used for guiding, navigating, and controlling various kinds of vehicles, such as aircraft, spacecraft, and ships.

The algorithm is recursive and composed of two phases. When performing prediction, the Kalman filter produces estimates of the current state variables and their uncertainties. As soon as the next measurement (necessarily corrupted with some error, including random noise) is observed, these estimates are updated using a weighted average, with more weight being given to estimates with more certainty. Furthermore, it can work in real time utilizing only the current input measurements, the previously determined state, and its uncertainty matrix, with no need for additional historical data.

The optimality of Kalman filtering is based on the assumption of a normal (Gaussian) distribution of errors.

Algorithm 2 The Kalman filter algorithm for linear Gaussian state transitions and measurements[12].

procedure ALGORITHM KALMAN_FILTER(μ_{t-1}, Σ_{t-1}, u_t, z_t)
    $\bar{\mu}_t = A_t\mu_{t-1} + B_t u_t$
    $\bar{\Sigma}_t = A_t\Sigma_{t-1}A_t^T + R_t$
    $K_t = \bar{\Sigma}_t C_t^T (C_t\bar{\Sigma}_t C_t^T + Q_t)^{-1}$
    $\mu_t = \bar{\mu}_t + K_t(z_t - C_t\bar{\mu}_t)$
    $\Sigma_t = (I - K_t C_t)\bar{\Sigma}_t$
    return μ_t, Σ_t
end procedure

Extended Kalman Filter

The extended Kalman filter (EKF) is a nonlinear variation of the Kalman filter that linearizes around an estimate of the current mean and covariance in estimation theory. In the theory of nonlinear state estimation, navigation systems, and GPS, the EKF has been deemed the de facto standard in the case of well characterized transition models.

The state transition and observation models in the extended Kalman filter don't have to be linear functions of the state; alternatively, they might be differentiable functions.

$$x_k = f(x_{k-1}, u_k) + w_k$$

$$z_k = h(x_k) + v_k$$

The process and observation noises, w_k and v_k, are respectively considered to be zero mean multivariate Gaussian noises with covariance Q_k and R_k. The control vector is u_k.

Algorithm 3 The extended Kalman filter algorithm[14].

procedure ALGORITHM EXTENDED_KALMAN_FILTER(μ_{t-1}, Σ_{t-1}, u_t, z_t)
    $\bar{\mu_t} = g(u_t, \mu_{t-1})$
    $\bar{\Sigma_t} = G_t \Sigma_{t-1} G_t^T + R_t$
    $K_t = \bar{\Sigma_t} H_t^T (H_t \bar{\Sigma_t} H_t^T + Q_t)^{-1}$
    $\mu_t = \bar{\mu_t} + K_t(z_t - h(\bar{\mu_t})$
    $\Sigma_t = (I - K_t H_t)\bar{\Sigma_t}$
    return μ_t, Σ_t
end procedure

2.1.7 Particle Filter

Particle filters, often known as sequential Monte Carlo techniques, are a collection of Monte Carlo algorithms for solving filtering issues in signal processing and Bayesian statistical inference. When partial observations are taken and random disturbances are present in both the sensors and the dynamical system, the filtering issue entails estimating the internal states in dynamical systems. Given some noisy and incomplete observations, the goal is to compute the posterior distributions of the states of some Markov process. Del Moral created the phrase "particle filters" in 1996 in relation to mean-field interacting particle approaches that have been utilized in fluid mechanics since the 1960s. Liu and Chen created the phrase

"Sequential Monte Carlo" in 1998.

Particle filtering represents the posterior distribution of a stochastic process given noisy and/or incomplete data using a set of particles (also known as samples). The initial state and noise distributions can be nonlinear, and the state-space model can be nonlinear. Particle filter approaches give a well-established mechanism for obtaining samples from the desired distribution without making assumptions about the state-space model or state distributions. When applied to very high-dimensional systems, however, these strategies fail miserably.

Algorithm 4 The particle filter algorithm, a variant of the Bayes filter based on importance sampling[15].

1: **procedure** ALGORITHM PARTICLE_FILTER(χ_{t-1}, u_t, z_t)
2: $\bar{\chi}_t = \chi_t = \emptyset$
3: **for** $m = 1$ *to* M **do**
4: sample $x_t^{[m]} \sim p(x_t \mid u_t, x_{t-1}^{[m]})$
5: $w_t^{[m]} = p(z_t \mid x_t^{[m]})$
6: $\bar{\chi}_t = \bar{\chi}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
7: **end for**
8: **for** $m = 1$ *to* M **do**
9: draw i with probability $\propto w_t^{[i]}$
10: add $x_t^{[i]}$ to χ_t
11: **end for**
12: return χ_t
13: **end procedure**

2.1.8 EKF Localization

The extended kalman filter is implemented by the ekf localization node, which allows a large number of parameters to be configured to determine the filter's accuracy as well as which sensors data are included in the estimate process. Each parameter is bundled together in configuration files, which are represented by file name.yaml in ROS, in order to appropriately set it. The configuration matrix related to a certain source of information and the process covariance matrix are the two most crucial parameters to select. The ekf localization node may get data from sensors by subscribing to ROS topics, where each sensor broadcasts sensed data in

a specified ROS message format. It can receive both odometry and IMU messages. The odometry data is collected through the RTK-GPS receiver, which is claimed to be fairly precise in this scenario. The GPS data is converted from GPS message format to odometry message format using the navsat transform node, which takes as inputs the information from the RTK-GPS receiver, the odometry information produced by the ekf localization node, and the IMU data (for the robot's heading), and outputs odometry messages represented in the robot's world frame.

2.1.9 AMCL Localization

Amcl is a probabilistic localization system for a robot moving in 2D. It employs a particle filter to track a robot's posture against a known map, as described by Dieter Fox's adaptive (or KLD-sampling) Monte Carlo localization technique[16].

Algorithm 5 MCL, or Monte Carlo Localization, a localization algorithm based on particle filters.[17].

procedure ALGORITHM MCL(χ_{t-1}, u_t, z_t, m)
  static w_{slow}, w_{fast}
  $\bar{\chi}_t = \chi_t = \emptyset$
  for $m = 1$ *to* M **do**
    $x_t^{[m]}$ = **sample_motion_model** (u_t, $x_{t-1}^{[m]}$)
    $w_t^{[m]}$ = **measurement_model** (z_t, $x_t^{[m]}$, m)
    $\bar{\chi}_t = \bar{\chi}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
    $w_{avg} = w_{avg} + \frac{1}{M} w_t^{[m]}$
  end for
  for m=1 to M **do**
    draw i with probability $\propto w_i^{[i]}$
    add $x_i^{[i]}$ to χ_t
  end for
  return χ_t
end procedure

Algorithm 6 An adaptive variant of MCL that adds random samples. The number of random samples is determined by comparing the short-term with the long-term likelihood of sensor measurements.[17].

procedure ALGORITHM AUGMENTED_MCL(χ_{t-1}, u_t, z_t, m)
  static w_{slow}, w_{fast}
  $\bar{\chi}_t = \chi_t = \emptyset$
  for $m = 1$ *to* M **do**
    $x_t^{[m]}$ = **sample_motion_model** $(u_t,\ x_{t-1}^{[m]})$
    $w_t^{[m]}$ = **measurement_model** $(z_t,\ x_t^{[m]},\ m)$
    $\bar{\chi}_t = \bar{\chi}_t + \langle x_t^{[m]}, w_t^{[m]} \rangle$
    $w_{avg} = w_{avg} + \frac{1}{M} w_t^{[m]}$
  end for
  $w_{slow} = w_{slow} + \alpha_{slow}(w_{avg} - w_{slow})$
  $w_{fast} = w_{fast} + \alpha_{fast}(w_{avg} - w_{fast})$
  for m=1 to M **do**
    with probabilty max{0.0, 1.0 - w_{fast}/w_{slow}} **do**
      add random pose to χ_t
    else
      draw $i \in \{1, ..., N\}$ with probability $\propto w_t^{[i]}$
      add $x_t^{[i]}$*to* χ_t
    end with
  end for
  return χ_t
end procedure

2.2 Mapping

In the previous sections we had as hypothesis that an a priori map of the environment was available. But in real case scenarios not all time we can posses a map, moreover "most buildings do not comply with the blueprints generated by their architects" [18], or some other times they are not compatible from a robot's perspective because the presence of new features may cause localization issues and misbehaviours. "Being able to learn a map from scratch can greatly reduce the efforts involved in installing a mobile robot"[18], it can be considered an additional feature that makes a robot truly autonomous. This task is obviously not easy, here we can list a number of challenging problems in robot mapping:

- Autonomously making a map is a "chicken-and-egg problem" [18], because the robot has to simultaneously map the surrounding and localize itself (SLAM problem), since odometry usually fails to keep a good estimate of the configuration of the robot.
- The size of the environment, which has to be represented in an discrete approximation, and a finite memory space.
- The noise in sensor perception that may result in map inconsistencies.
- The perceptual ambiguity, making it difficult to associate a location to a particular feature in the environment.

In this section we will see how the problem is faced under different assumptions.

2.2.1 SLAM

Simultaneous localization and mapping (SLAM) is the computational issue of creating or updating a map of an unknown environment while keeping track of an agent's position inside it at the same time. While this may appear to be a chicken-and-egg issue at first glance, there are numerous techniques for solving it in tractable time for particular situations. The particle filter, extended Kalman filter, covariance intersection, and GraphSLAM are all popular approximation solution

approaches. SLAM algorithms are utilized in robot navigation, robotic mapping, and odometry for virtual reality and augmented reality, and are based on principles in computational geometry and computer vision.

2.3 Global Planning

2.3.1 Dijkstra's algorithm

Dijkstra's algorithm is a greedy strategy algorithm for finding the shortest paths which connect all the nodes of a graph to a chosen source node. This algorithm has been devised by Edsger W. Dijkstra in 1956 [19]. Moreover, it is employed several fields applications under the assumption that the given problem can be modeled with a graph representation. Hence, the most important utilizations deal with network routing protocols, automated navigation systems and/or subroutine in other algorithms.

Dijkstra's approach makes use of three main data structures to store the list of the minimum distances of all the nodes from the source, the list of links of the shortest paths, and a priority queue useful for the algorithm's backbone. Moreover, the list of minimum distances and the list of links will be the output of the algorithm.

2.3.2 A-star algorithm

The A-star algorithm is an heuristic best-first search algorithm published by Peter Hart, Nils Nilsson and Bertram Raphael of Stanford Research Institute in 1968. Initially, it has been devised for a project which consists of a robot that plans its own actions; but then, due to its efficiency and optimality, it became one of the most used algorithms in several fields.

A-star is a heuristic iterative algorithm applied to weighted-graphs, hence it does not explore all the solutions state space. Conversely, at each iteration, it chooses the path which is more convenient to explore by minimizing a cost function defined as follows:

$$f(n) = g(n) + h(n)$$

where *f(n)* is the cost function evaluated for node *n*, *g(n)* is the cost paid to build the path from the source to *n*, and *h(n)* is an estimation of the cheapest cost

Algorithm 7 Dijkstra Algorithm[19].

```
1: procedure DIJKSTRA(Q, s, w)
2:     Initialization
3:     for each v in Q do
4:         dist[v] = ∞
5:         prec[v] = null
6:     end for
7:     dist[s] = 0
8:     Compute the best path
9:     while Q ≠ empty do
10:        u = vertex with the minimum dist[] in Q
11:        remove u from Q
12:        if dist[u] = ∞ then
13:            break
14:        end if
15:        for each neighbour v of u do
16:            temp = dist[u] + w(u, v)
17:            if temp ≤ dist[v] then
18:                dist[v] = temp
19:                prec[v] = u
20:            end if
21:        end for
22:    end while
23:    return dist, prec
24: end procedure
```

paid to reach the goal starting from n.

Accordingly to Dijkstra's approach, A-star makes use of a priority queue to choose which node has to be expanded. The only differences is in the criterion used to sort the queue, indeed Dijkstra uses the distance from the source while A-star the cost function *f(n)* defined before.

2.4 ROS2

ROS (Robot Operating System) is an open-source software development kit for robotics applications [21]. Despite its name, it is not an operating system but more like a Software Development Kit (SDK) since it provides a common standard platform to develop a robotic application.

> "Don't reinvent the wheel. Create something new and do it faster and better by building on ROS!" [22]

ROS was created more than ten years ago by Keenan Wyrobek and Eric Berger while at Stanford, an attempt to remove the reinventing the wheel situation from which robotics was suffering [23]. The ROS project has then been maintained by Willow Garage and the Open Source Robotic Foundation (OSRF).

The foundation of ROS is its communication system, the "middleware" or "plumbing" as defined in [24], which is a standardized messaging system already provided in ROS. It controls the communication details between the nodes (the distributed processes), which can also execute over different machines, using a publish/subscribe approach. ROS also includes tools for developers such as systems for launching, debugging the application, or testing and visualizing the results. In advance, ROS provides a series of packages that already implement the drivers or the algorithms for the most common task, such as GPS, control system, or even user interfaces; this permits the developers to focus only on the application. "The goal of the ROS project is to continually raise the bar on what is taken for granted, and thus to lower the barrier to entry to building robot applications," is stated on ROS website [24]. The ROS ecosystem has at its back a large community of students, researchers, or corporations that constantly develop and support the project. In

Algorithm 8 A-star Algorithm[20].

```
1: procedure A-STAR(start, goal)
2:     closedset := the empty set
3:     openset := set containing the initial node
4:     g_score[start] := 0
5:     came_from := the empty map
6:     h_score[start] := heuristic_estimate_of_distance(start, goal)
7:     f_score[start] := h_score[start]
8:     while openset is not empty do
9:         x := the node in openset having the lowest f_score[] value
10:        if x = goal then
11:            return reconstruct_path(came_from,goal)
12:        end if
13:        remove x from openset
14:        add x to closedset
15:        for each y in neighbor_nodes(x) do
16:            if y in closedset then
17:                continue
18:            end if
19:            tentative_g_score := g_score[x] + dist_between(x,y)
20:            if y not in openset then
21:                add y to openset
22:                tentative_is_better := true
23:            else
24:                if tentative_g_score < g_score[y] then
25:                    tentative_is_better := true
26:                else
27:                    tentative_is_better := false
28:                end if
29:            end if
30:            if tentative_is_better = true then
31:                came_from[y] := x
32:                g_score[y] := tentative_g_score
33:                h_score[y] := heuristic_estimate_of_distance(y, goal)
34:                f_score[y] := g_score[y] + h_score[y]
35:            end if
36:        end for
37:    end while
38:    return failure
39: end procedure
```

Algorithm 9 The reconstruct_path employed in A-star Algorithm[20].

```
1: procedure RECONSTRUCT_PATH(came_from, current_node)
2:     if came_from[current_node] is set then
3:         p = reconstruct_path(came_from,came_from[current_node])
4:         return (p + current_node)
5:     else
6:         return the empty path
7:     end if
8: end procedure
```

the last years, ROS has been going through a significant transformation to ROS2. The latter has been developed from scratch to incorporate new technologies such as Data Distribution Service (DDS), meet real-time constraints, and run on different Operating Systems (OS).

2.4.1 Graph Concepts

As the foundation of a ROS2 system, there is the ROS graph. "The ROS graph refers to the network of nodes in a ROS system and the connections between them by which they communicate," as stated in the official documentation [25]. The graph is composed of the following building blocks:

- **Nodes**: entity in the ROS graph communicates to other nodes through topics.
- **Messages**: data that are exchanged using topics between nodes
- **Topics**: channels where a node can publish messages or subscribe in order to get messages from it.
- **Discovery** is how nodes make their first connections and become aware of each other.

Nodes

In the ROS graph, a node is an "independent computing process" [26] and utilizes the ROS clients library to communicate with other nodes. In a single application (single executable written in C++, Python, ...), one or more nodes can be defined, and they also interact if located in different machines.

"Each node in ROS should be responsible for a single, module purpose (e.g. one node for controlling wheel motors, one node for controlling a laser range-finder, etc)" [25].

Thanks to Topics, communication, and interaction between nodes are possible; a node can publish messages over a Topic while others can receive it if they are subscribed to it. The other ways of communication provided are Services and

Actions. Moreover, there are configurable Parameters associated that can be set and associated with a node.

Messages and communication interfaces

The communication in ROS can be thanks to interfaces. There are three types of interfaces: messages, services, and actions. The interfaces have a simple data structure which is defined using "the interface definition language (IDL)" [27], ROS will automatically generate from this description interfaces source code in different programming languages such as C++ Python or DDS type. Now we will briefly describe these interfaces:

- Messages are simple data with a structure defined in a .msg file; these data are exchanged by nodes using topics. A Topic can be considered a channel for nodes to exchange messages; it is a unilateral way of communication. Only a node can send messages over a topic, but more nodes subscribed to it can receive the data stream.

- Services are another method of communication for nodes. "Services are based on a call-and-response model, versus topics' publisher-subscriber model" [28]. The services offer a Request / Response type of interface. This interface type, defined in a .srv file, is composed of two parts (two messages): one for the request and the other for the response. A server node will offer the service under a particular name and compute the answer whenever a client node asks for this service.

- "Actions are one of the communication types in ROS 2 and are intended for long-running tasks" [29]. They are defined in a .action file consisting of three message declarations: goal, feedback, and result. This structure is composed of topics and services. It is similar to services because it utilizes a client-server model but, in addition, also publishes a data stream over a feedback topic. "An "action client" node sends a goal to an "action server" node that acknowledges the goal and returns a stream of feedback and a result" as written in [29].

Discovery

In contrast with what happened in ROS1, "The Discovery of nodes happens automatically through the underlying middleware of ROS 2" [25]. This process can be analyzed into three phases of the node lifecycle. At its startup, a node will announce itself in the network; after this, the other Nodes will respond to this advertisement with the necessary information to establish the appropriate communications through the correct interfaces. During their execution, nodes will notify about their presence in the network at defined time intervals to allow the connection to new components. Before the shutdown, a node will advertise its detaching from the graph.

2.4.2 ROS 2 architecture overview

In order to meet various objectives such as real-time performances, being cross-platform, and being able to be deployed on embedded systems, ROS2 has great innovation from ROS1, first of all, the incorporation of Data Distribution Service (DDS) technology.

DDS

The DDS is "an industry-standard real-time communication system and end-to-end middleware" as declared in [26]. It enables "reliable publish/subscribe transport similar to that of ROS1" [26]. Still, in contrast with ROS1, it can be applied in real-time programming tasks and embedded systems thanks to its flexibility in the configuration of the communication implementation. As stated in [26] "DDS meets the requirements of distributed systems for safety, resilience, scalability, fault-tolerance and security"; it is a great leap forward in reliability to meet industrial standards. DDS is developed and provided by many vendors. "Several implementations of this communication system have been used in mission-critical environments (e.g., trains, aircrafts, ships, dams, and financial systems) and have been verified by NASA and the United States Department of Defense" [26].

ROS1 versus ROS2 architecture

ROS1 communication system is based on TCPROS/UDPROS protocols, a custom implementation of the ROS middleware which utilizes the TCP and UDP sockets. This implementation needs a master-node that interacts and manages the interconnection between subscribers and publisher nodes of the same topic. The master-node establishes a direct connection between the subscriber and publisher at their startup. "Actual data (i.e., a message) is transported directly between nodes" [26], in this way, the communication does not involve the ROS master anymore. At the end of the transaction, we have peer-to-peer communication between the nodes. On the contrary, ROS2 is built on top of a DDS/RTPS (Real-Time Publish-Subscribe) communication system which is the middleware implementation. "DDS is an end-to-end middleware that provides features which are relevant to ROS systems" [30]. Thanks to DDS, in ROS2, we can have a decentralized discovery (we do not need a ROS master); moreover, there is control over the Quality of Service (QoS) policies that define the data transportation behavior. The elimination of the ROS master makes ROS2 a "system more fault-tolerant and flexible" [31]. In ROS2, you can choose your middleware implementation depending on various aspects and needs, like the license or the computation performance of your device.

ROS2 to utilize DDS needs a package that can abstract the underlying DDS API to a higher level ROS interface; this leads to a "ROS Middleware interface" (rmw interface or just rmw).

ROS2 internal architecture

There is an abstraction between the ROS facilities and the DDS implementation, as introduced before. This abstraction is composed of two interfaces:

- rmw: The ROS middleware interface; stands between the ROS2 stack and the middleware distribution (DDS/RTPS). Its purpose is to catch the minimum middleware functionalities that may be used in the definition of the client libraries; they are written based on the DDS vendor-specific interfaces.

- rcl: The ROS client library interface, an abstraction of the underlying rmw,

is used to implement the client libraries ignoring the particular middleware implementation. "The purpose of rcl is to provide a common implementation for more complex ROS concepts,... while remaining agnostic to the underlying middleware being used" [32]

Above these two interfaces stand the client libraries that the typical ROS developer will use to develop the robot application. The client libraries to develop a robotic application using ROS2 are:

- rclcpp
- rclpy
- rclc or other minor client libraries not directly supported

This layer of abstraction provides to the developer all the functionalities for communication, creation, and integration of nodes in the ROS graph.

Chapter 3

Seamless Navigation in Outdoor and Indoor environments

This chapter will contain a detailed description of the proposed solution to the seamless navigation problem. The first part will focus on the existing packages in ROS2 to handle sensor fusion for outdoor navigation and the already available application to perform waypoint following. Then the significant changes and the part of the application developed from scratch will be highlighted from the rest.

3.1 Sensor fusion using Robot Localization and AMCL

In this thesis work, since there are two types of navigation, one for outdoor locations and one for indoor surroundings, two different localization filters have been chosen.

- EKF for outdoor localization
- AMCL for indoor localization

These choices have been made considering the main characteristics of the sensors and the difficulties of the different environments. The system on which the model is applied is the Husky UGV, a vehicle with a skid-steering drive system capable of navigating different types of terrain. The UGV is equipped with: a LIDAR 2D, an IMU, a GPS, and a depth camera for obstacle avoidance.

Since in an outdoor environment, it's possible to utilize an RTK capable GPS which gives accurate measurements about the UGV position, and an IMU to provide precise orientation estimation. In this circumstance, considering the non-linearity of the system and the complexity of the other filters not justified in this situation, an EKF filter has resulted as the best choice. This choice is supported by the relatively good accuracy in position estimation and the low complexity of the implementation. Other filter choices such as the UKF or the particle filter would have been more expensive in both code implementation and computational performance with a not justifiable gain in terms of accuracy.

In indoor environments, the situation is quite different. The GPS receiver does not perform well, but a map of the surroundings is available in a ROS-compatible format. In order to localize itself in this map, the robot utilizes a particle filter that takes the already fused odometry information (of the robot motion model and the IMU) and the Lidar measurements matched with the map. The use of the particle filter and, in particular, of the Adaptive Monte Carlo localization permits a robust and efficient solution that allows global localization, also if the sensors are noisy. This filter is already implemented in the AMCL package of Nav2.

3.1.1 EKF and navsat node configuration

As introduced in the last chapter ROS introduces a package to perform localization, the `robot_localization` package. This package has been chosen as the implementation of the two EKF filters through the `ekf_localization_node` `ekf_localization_node` and for the coordinate transformation between the WGS and the map reference performed by the `navsat_transform_node`.

The `ekf_localization_node` "uses an omnidirectional motion model to project the state forward in time" [33]. The node does not utilize the skid steer motion

model (the one of the Husky) to update the localization from the odometry sources. Although the differences in the motion model, the node performance doesn't decay too much because the various sensor measurements correct the state estimate prediction. Two nodes have been utilized to perform the localization of the husky:

- `ekf_filter_node_odom`: is used to fuse the data of the wheel encoders published on the */wheel/odometry* topic and the IMU data provided on the */imu/-data* topic. The state estimation result is given as a `nav_msgs/msg/Odometry` published on the /odom topic, which is the estimate of the current pose from the cumulative motion update. The node also computes and publishes the coordinate transformation between the `odom` coordinate frame and the `base_link` coordinate frame attached to the robot.

- `ekf_filter_node_map`: is another `ekf_localization_node` used for pose estimation; it uses the IMU sensor data to gain information about the vehicle orientation and the GPS odometry data for the robot's position. The GPS odometry information comes from the `navsat_transform_node`. The node provides a transformation between the `map` reference frame, the global fixed frame, and the `odom` frame, but it will not be published directly by this node for reasons explained later.

The other node utilized by the Robot Localization package is an instance of the `navsat_transform_node`:

- `navsat_transform`: these node inputs are the GPS coordinates in latitude and longitude, together with the odometry msgs published by the `/ekf_filter_node_map`. In this way, the node can correctly transform the GPS coordinates into a position in the robot's map frame.

These three nodes together can directly be used to completely localize the vehicle in outdoor surroundings, which consists of finding the map -> base-link transformation.

But since our task is to localize in both outdoor and indoor environments, the structure has been modified to include another filter and localization mechanism.

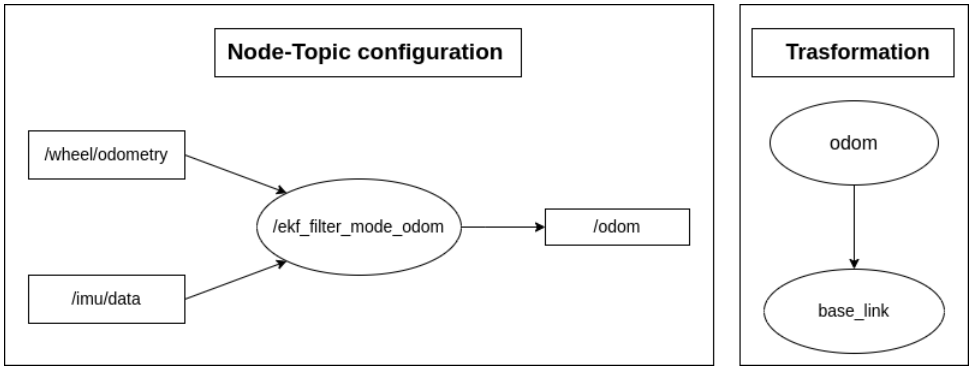

Figure 3.1: *ekf_filter_node_odom* node the related topics, and the provided transformation

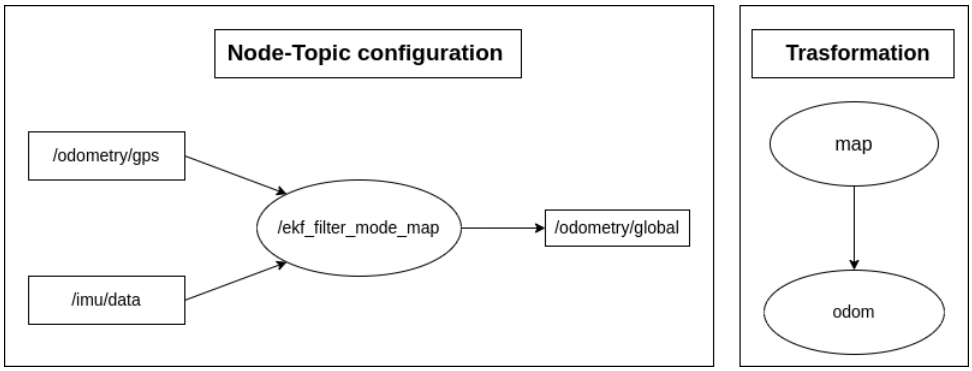

Figure 3.2: *ekf_filter_node_map* node the related topics, and the provided transformation

3.1.2 AMCL configuration

As introduced in the last chapter, the AMCL node is part of the Nav2 distribution and is utilized to localize the robot in a known environment. This is the approach chosen in the indoor environment, and when a map of the surrounding is already available or can be obtained from a SLAM first lap. The AMCL node in this configuration already provides motion models for a differential-drive or omnidirectional robot. Among these two, the differential drive motion model has been chosen because it is closely related to the skid steer drive of the husky UGV. The measurement model utilized is instead the `likelihood_field` described in more detail afterward. The advantage of this model is the smoothness and the

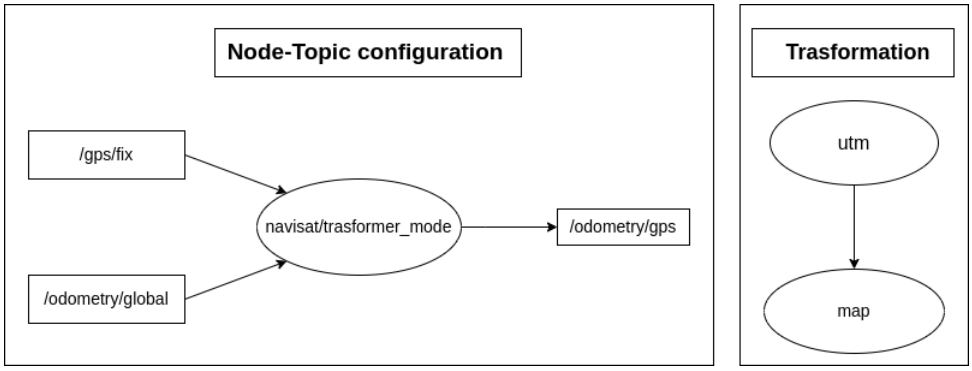

Figure 3.3: *navsat_transform_node* node the related topics, and the provided transformation

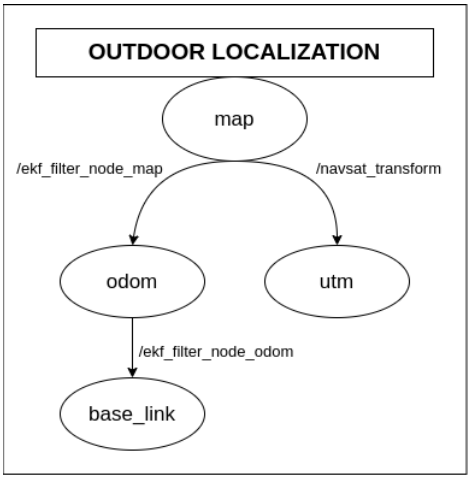

Figure 3.4: Transformations chain for outdoor localization

possible precomputation of the probability resulting in a significant reduction of online computation efforts. As the `ekf_localization_node_map`, its task is to provide a global estimate of the pose in the map. This is done using a particle filter to fuse the information coming from the Lidar and odometry (supplied by the `ekf_filter_node_odom`). The map is retrieved using the `map_server` node. The result is the configuration estimate of the husky model, available on the `/amcl_pose` and the map->odom transformation, which will not be published.

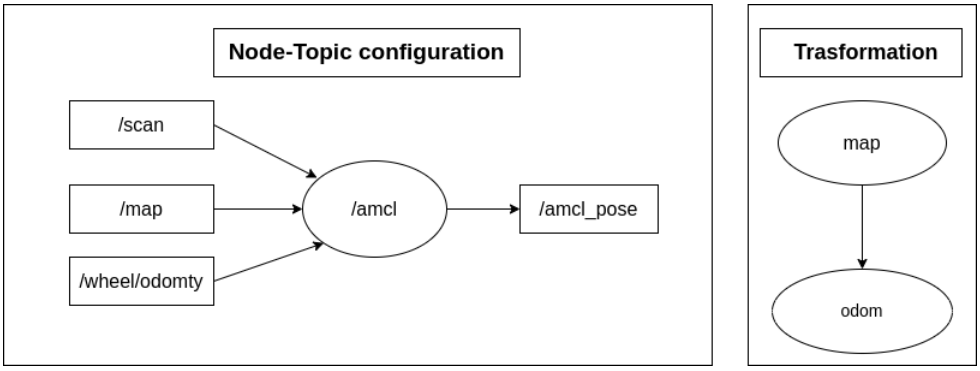

Figure 3.5: *amcl_node* node the related topics, and the provided transformation

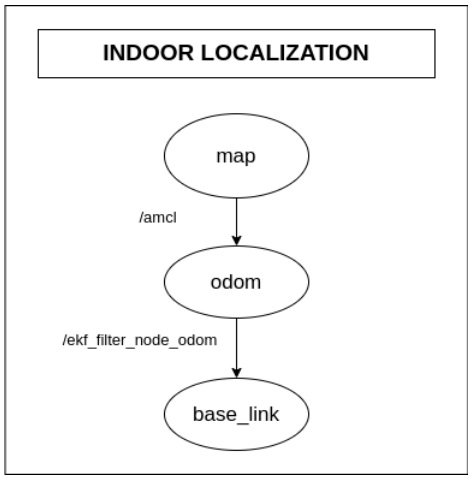

Figure 3.6: Transformations chains for indoor localization

3.2 Localization broadcaster

To obtain an actual seamless navigation application it is needed a proper localization in both environments and a method to have a transition between the two. The strategy to handle both localization filters and select which of the two estimates must be utilized is implemented in a custom node: the `localization_broadcaster`. This node takes as input the two filter estimates of the UGV configuration:

- /odometry/global is the topic on which the estimate of the UGV configuration computed by the `/ekf_localization_node` is published. It is in the form of a `nav_msgs/msg/Odometry` message and is the best possible estimate in the

outdoor environment.

- `/amcl_pose` is the topic on which the estimate of the vehicle pose computed by the AMCL filter is published. The configuration and its covariance are delivered with a `geometry_msgs/msg/PoseWithCovarianceStamped` message; this estimate must be utilized in indoor surroundings.

The node also listens to the `odom->base_link` transformation and computes its inverse. Thanks to these data, it is possible to obtain the map->odom transform for either outdoor or indoor situations and publish the best choice in that particular moment. The choice is made using a boolean parameter that depicts the initial condition (true for indoor and false for outdoor), which can be changed during the navigation activity through the `/set_bool` service.

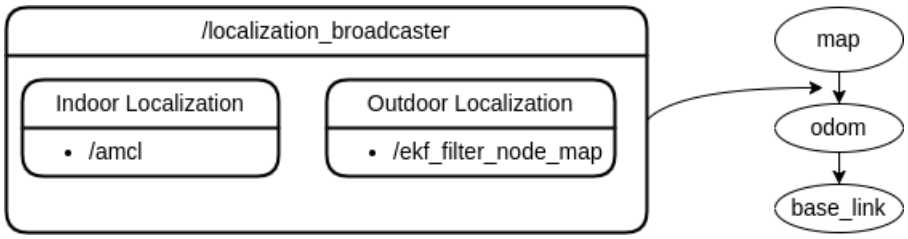

Figure 3.7: Full transformations chain for seamless localization

AS stated in the REP-105, it is necessary to "build a TF tree that contains a full `map -> odom -> base_link -> [sensor frames]` for your robot" [34]. Using this schema is possible to obtain satisfactory performances in global localization, obtaining a complete transformation chain from the fixed `map` frame to the `base_link` reference frame. All the transformations are published using the tf2 package. Since each part of the robot has an associated reference frame in ROS, we will also define a URDF file where all the relative transformations between robot components and sensors are implemented. All this structure solves the problem of global localization, which is fundamental to obtaining satisfactory results in the navigation process.

3.3 Waypoint follower

> "Waypoint following is a basic feature of a navigation system. It tells our system how to use navigation to get to multiple destinations" [34].

This package is already present in the Nav2 distribution, but it has been modified to accommodate our custom needs. The aim of the `nav2_waypoint_follower` package in its original configuration is to take a set of waypoints as inputs, navigate to each of the positions set by these waypoints, and perform a task when they reach the destination. In particular, this application implements an action server named `follow_waypoints` which handles the list of waypoints and delegates the navigation to the `bt_navigator` through the `Navigate_to_pose` action. The execution of the task is instead a duty of the waypoint task executor described in detail later. As described in its documentation, "it is a nice demo application for how to use Nav2 in a sample application" [35], but it can be more than that. It is an application that permits not only to navigate autonomously but also to perform a precise task when needed. This can be employed to make the transition between the two surrounding easier, as we will see later.

The following sections describe the main adjust to this application:

- Handle GPS waypoints.
- Use the Navigate through poses action to perform navigation.
- Modify the waypoint task executor to handle different plugins.
- Modify the waypoint follower interfaces.
- Write a custom task to switch the localization mechanism.

These adjustments make the application suitable for our goal, having a UGV capable of seamlessly navigating in indoor and outdoor surroundings.

3.3.1 GPS Waypoint follower

The first modification made to the waypoint follower is to accept GPS waypoints. The waypoint follower takes only cartesian coordinates relative to the fixed map

frame in its original form. The GPS waypoints are defined in a .yaml file formatted as follows:

```
#lat, long, alt, yaw(radians)
wp0: [47.84062029394745, 10.61997426956434,
740.0753334760666, -0.8405847300640226]
```

The first number represents the latitude, the second the longitude, then the altitude. At last, the robot's orientation, which in 2-D is only defined by the yaw angle here in radians representation. A list of these waypoints is sent to the `follow_GPS_waypoints` server through a newly defined message called `nav2_msgs/OrientedNavsatFix`.

When the server receives the goal, it converts the list of GPS waypoints to poses in the map fixed frame using the `geometry_msgs/PoseStamped` message format. This process is performed using the `/fromLL` service provided by the `/navsat_trasform_node` included in the robot localization package. The node loads and transforms all the GPS waypoints before starting the navigation task. Using this approach, the application can handle routes outside the map borders since we know the destination and we can handle obstacles dynamically.

3.3.2 Navigate through poses

The second modification to the original waypoint follower application is the use of the *NavigateThroughPose* action to perform navigation. The *NavigateThroughPose* action differentiates from *NavigateToPose* in the ability to specify a set of intermediate configurations that the Husky UGV must assume before reaching the final destination.

The Navigate through poses action is used to navigate through a series of configurations defined in a vector of *PoseStamped* message. Its execution utilizes the `bt_navigator` package, which then loads the following behavior tree `navigate_throgh_poses_w_replanning_and_recovery` (that can be seen in the picture).
The tree defines the actions necessary for navigation and the recovery procedures

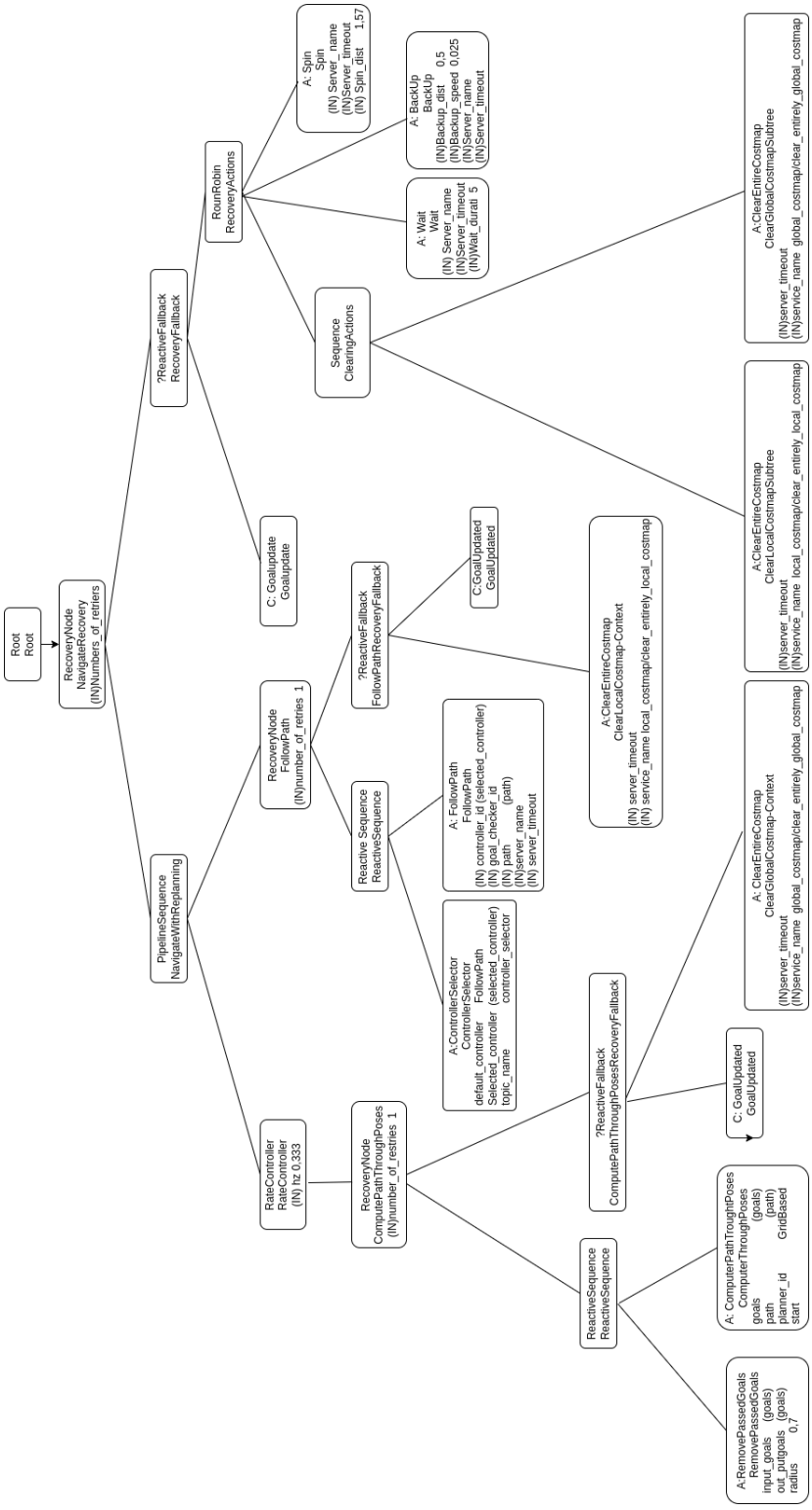

Figure 3.8: *navigate_throgh_poses_w_replanning_and_recovery[36]*

for handling unexpected events. The tree can be split into two parts the navigation subtree and the recovery subtree.

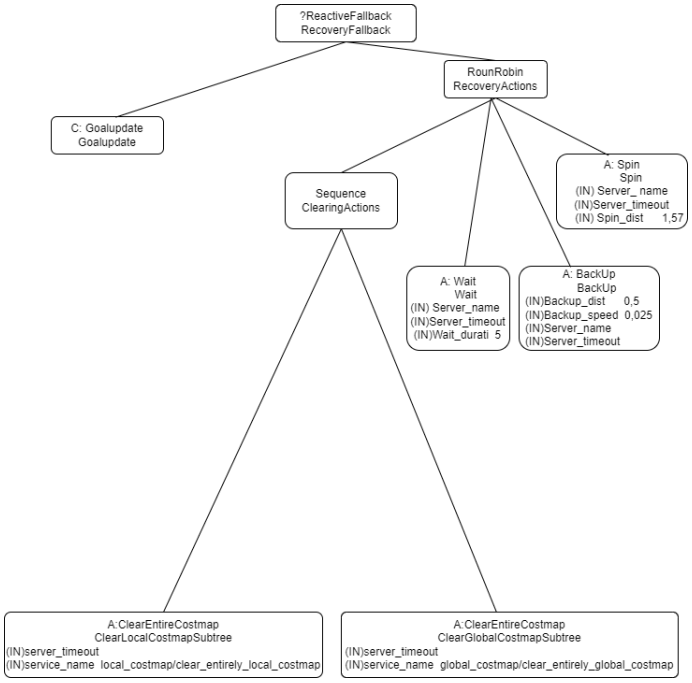

Figure 3.9: *behavior tree navigation[36]*

In the navigation part, we can distinguish the following navigation actions:

- calculating a path through the poses and removing the ones that are already completed
- following a path
- recovery behaviors which are related to the first two categories

The *ComputePathThroughPoses* action handles the planning and will use an approach loaded as a plugin in the planner server (see nav2). In normal execution, there will be a replanning every three seconds, and the *RemovePassedGoals* node will remove the goals that the UGV has already passed on its path. An intermediate pose will be removed from the course when the robot is inside an imaginary circle around this pose of a parametrized radius. This is done to avoid the robot continuing to try to replan through them in the future. After obtaining a feasible global

plan, the *FollowPath* node will begin the navigation using a suitable controller for the path following task defined in the controller server by another plugin. If either the planning or path following task has some problems, "contextually aware recoveries" [36] will be attempted, such as clearing the global or local costmap The "*GoalUpdated* node allows us to exit recovery conditions" [36], this happens when we have set another goal for the system. This node ensures that the navigation system will respond immediately when a new goal is present. If neither of the contextual recoveries are capable of resolving the situation, the recovery subtree will take action.

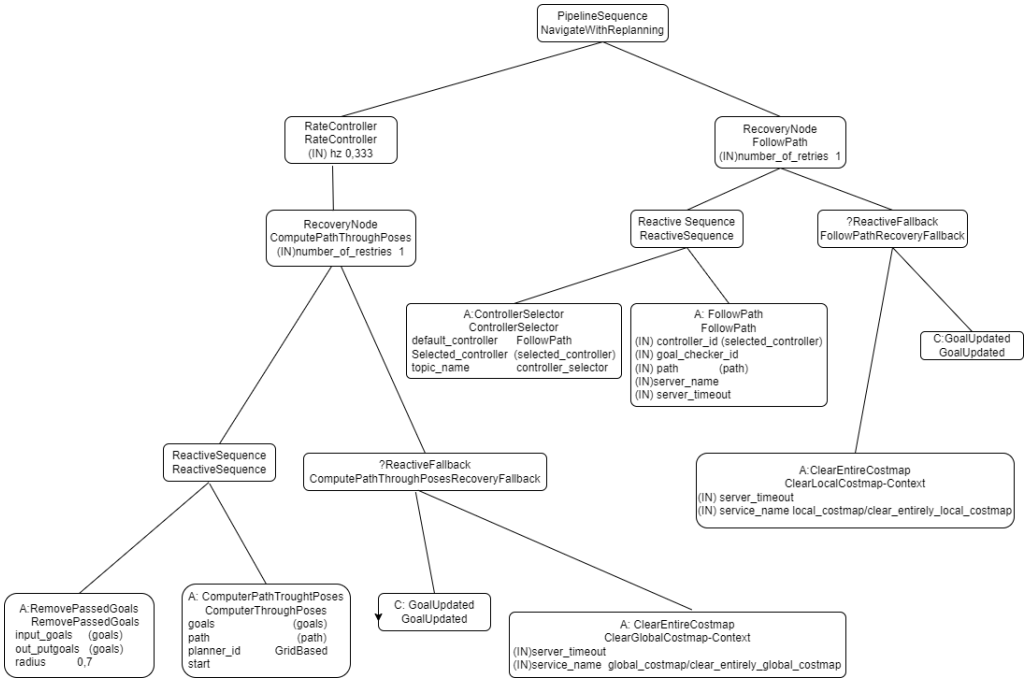

Figure 3.10: *behavior tree navigation[36]*

"This subtree is reserved for system-level failures to help resolve issues like the robot being stuck or in a bad spot" [36] In this subtree, recovery operations such as clearing both costmaps, spinning, and waiting will occur. After each of them, navigation will be attempted again before shutdown procedures. As seen before, the behavior tree structure allows great flexibility and modularity, which are two fundamental principles to have an autonomous navigation system. The particular

choices of the plugins for planning and path following will be discussed later.

3.3.3 Waypoint task executor plugins

Another modification has been made to the waypoint follower application in order to accommodate more than a single task plugin. In particular, it is possible to load a list of plugins that define the robot's behavior when it reaches the final goal. The already available task the robot can handle at a waypoint is:

- *WaitAtWaypoint*: a demo plugin that tells the robot to wait for a specified amount of time
- *InputAtWaypoint*: a demo plugin that puts the robot on pause when it reaches the destination till an input is given on a specified topic.
- *PhotoAtWaypoint*: a demo plugin that takes photos when it reaches the waypoint destination and saves it to a specified directory.
- *SwitchLocalizationAtWaypoint*: a custom plugin developed to perform a switch between the outdoor or indoor localization mode.

switch localization at waypoint

The *SwitchLocalizationAtWaypoint* is a plugin that has the task of switching the localization mode. In particular, when the robot reaches the desired goal, it calls the `/set_bool` service of the `localization_broadcaster` service. This task is used for particular waypoints positioned at the interfaces of a building or a structure. The idea is to change the type of localization when we are facing a transition between an indoor and an outdoor environment or vice versa. The `/set_bool` service is used to trigger the internal parameter of the `localization_broadcaster` application, which decides if the localization is performed using the EKF filter and GPS data or the AMCL filter.

When the transition points are chosen correctly, the re-localization task succeeds in a few update steps, and the robot navigates seamlessly from one environment to another.

3.3.4 The new Waypoint Follower interfaces

The *FollowWaypoints* action has been further modified to manage :

- A goal and some intermediate poses that the robot should cross
- A task associated with the path

The new actions *FollowWaypontPluginPath* (and *FollowWaypontPluginPath*) can handle both of the above requirements. The goal of this action is a vector of "paths"; the idea is to specify the UGV route as a group of different sub-paths: each of these has its final goal where must perform the task and optionally a set of intermediate poses to specify the route better. Each subpath and task plugin is specified in a *WPathPlugin* (or *GPSWPathPlugin*) message. A custom node loads the subpaths from a .yaml file with the following structure:

```
1    paths: ["path1","path2"]
2    path1:
3      task: "switch_localization_at_waypoint"
4      waypoints: [wp0, wp1, wp2, wp3]
5      #lat, long, alt, yaw(radians)
6      wp0: [47.84062029394745, 10.61997426956434,
   740.0753334760666, -0.8405847300640226]
7      wp1: [47.84059982635257, 10.62000124690127,
   738.8925913609564, -0.7768276186670429]
8      wp2: [47.84057298379028, 10.62000168506552,
   740.265083713457, -1.577444088578223]
9      wp3: [47.8405469428392, 10.6199984385851,
   739.3662512088194, -2.316651401838845]
10   path2:
11     task: "wait_at_waypoint"
12     waypoints: [wp0, wp1, wp2, wp3, wp4]
13     wp0: [47.84062029394745, 10.61997426956434,
   740.0753334760666, -0.8405847300640226]
14     wp1: [47.84059982635257, 10.62000124690127,
   738.8925913609564, -0.7768276186670429]
```

```
15      wp2: [47.84057298379028, 10.62000168506552,
    740.265083713457, -1.577444088578223]
16      wp3: [47.8405469428392, 10.6199984385851,
    739.3662512088194, -2.316651401838845]
17      wp4: [47.84054256418474, 10.61995912998031,
    739.4493419714272, -3.001265257485784]
```

This example shows two subpaths where the waypoints are specified GPS coordinates with the relative tasks.

3.4 Nav2 configuration

The navigation2 distribution, as seen before, is a group of packages that offers a highly configurable stack for navigation, offering various plugins that implement different strategies for planning, path following, and recovery. As described before, the Behaviour Tree navigator calls the execution of the various action servers to complete different navigation-related actions. In particular with the `navigate_throgh_poses_w_replanning_and_recovery` behavior tree has been used as already discussed. The *SmacPlannerHybrid* has been chosen for computing a collision-free path between the waypoints. On the controller server instead is loaded the plugin of the *DWBLocalPlanner* that handles the path following task. The global costmap and the local costmap share both an obstacle layer and an inflation layer, the global costmap in addiction has a static layer.

3.4.1 Global Planner

The *SmacPlannerHybrid* is a Hybrid-A* reconfigurable implementation highly configurable and suitable for both differential and Ackermann motion models. The planner is implemented by the `nav2_smac_planner` that, as written in the documentation, "contains an optimized templated A* search algorithm used to create multiple A*-based planners for multiple types of robot platforms" [37]. The plugin supports robots with different motion models and is also indicated for differential, not circular-shaped robots or ones with Ackermann drive.

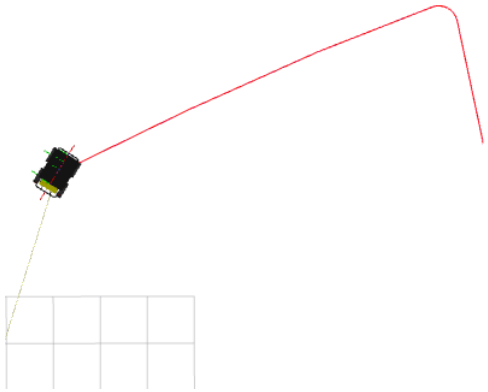

Figure 3.11: Global path

3.4.2 Local Planner

The *DWBLocalPlanner* is the successor of the DWA controller already implemented in ROS1. It is more configurable than DWA and supports local planning for both differential and omnidirectional motion models. It uses plugins to generate a set of possible trajectories and score them. It is an extension of the DWA, which can also be reconfigured during runtime. In the following table, there is a recap of the already described DWA approach for local planning:

With this approach, the robot can dynamically avoid obstacles also not present in the static map. The trajectories generated are then evaluated using a cost function that sums the scores of different factors represented by these plugins:

- **BaseObstacle** - evaluates the trajectory using depending on the presence of obstacles. It uses the local costmap and scores the course accordingly.
- **GoalAlign** - evaluates the trajectory depending on its alignment with the following goal pose.
- **GoalDist** - evaluates the trajectory depending on its final distance to the goal pose.
- **PathAlign** - evaluates the trajectory alignment with the path provided by the planner.

1. Discretely sample in the robot's control space $(dx, dy, d\theta)$
2. For each sampled velocity, perform forward simulation from the robot's current state to predict what would happen if the sampled velocity were applied for some (short) period of time.
3. Evaluate (score) each trajectory resulting from the forward simulation, using a metric that incorporates characteristics such as: proximity to obstacles, proximity to the goal, proximity to the global path, and speed. Discard illegal trajectories (those that collide with obstacles).
4. Pick the highest-scoring trajectory and send the associated velocity to the mobile base.
5. Rinse and repeat.

InfoBox 3.1: [38]

- **PathDist** - evaluates the trajectory depending on its final distance to the global plan.
- **RotateToGoal** - Used to allow the UGV to rotate for aligning with the goal orientation when it is close enough to the destination.
- **Oscillation** - It is used to avoid repetitive motions backward and forward.

As seen in the plugin description, the DWA planner also depends on the local costmap, which collects obstacle information. For this reason, "tuning the parameters for the local costmap is crucial for optimal behavior of DWA local planner," as stated in [39]. The resulting trajectory will then be used for the robot motion.

3.4.3 Global costmap

The global costmap is configured to consider the static and dynamic obstacles. The static obstacles are loaded using the `static_layer`, which loads the occupancy information from the static map. In contrast, the dynamic ones are loaded using the obstacle layer that uses the Lidar data as input. The inflation layer can assign a cost

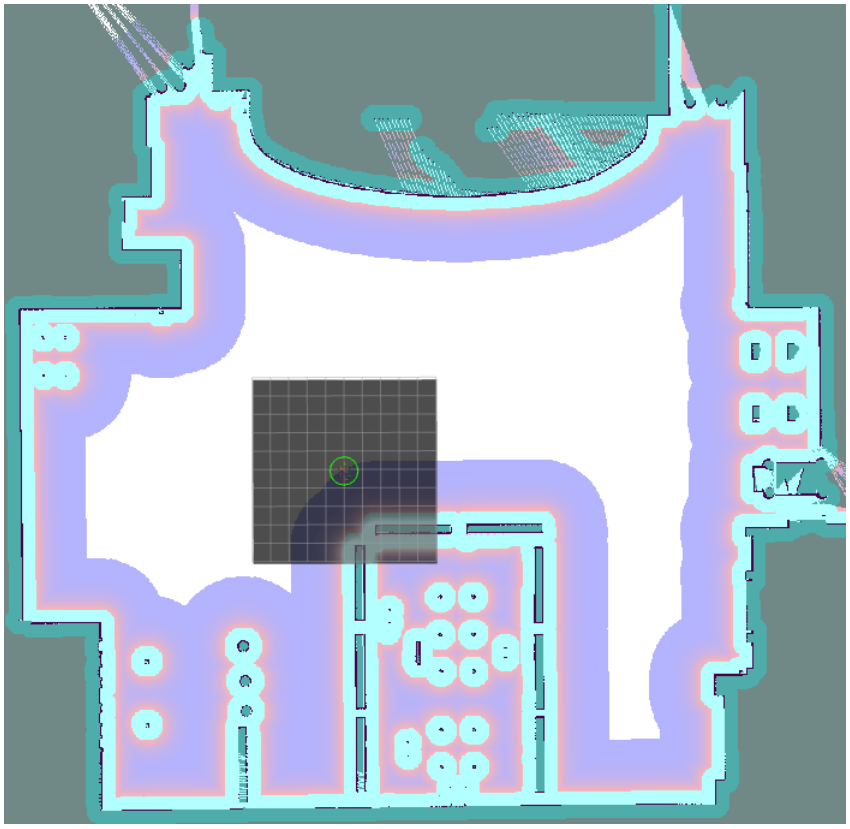

Figure 3.12: Global costmap

to each cell of the map (between 0 and 255) according to the robot's footprint and the surrounding obstacles. The obstacles appear inflated by a range proportional to the robot footprint. In addition, the inflation layer is used to create a smooth gradient of costs around obstacles to improve the planner's performances, keep distances from obstructions, and avoid risks to remain stuck. The smooth gradient is created using two additional parameters, the inflation radius and the cost scaling factor.

3.4.4 Local costmap

This costmap is used only to consider the dynamic obstacles and help the local planner compute the best trajectories. The local costmap uses the voxel layer instead of the `obstacle_layer`; it is used to evaluate obstacles that can be perceived

not only by 2D laser scans but also using *PointCloud* or *PointCloud2* messages coming from a depth camera. It is also used the inflation layer as viewed in the global costmap. The costs provided by the inflation layer are used to evaluate the different trajectories in the DWA evaluation process.

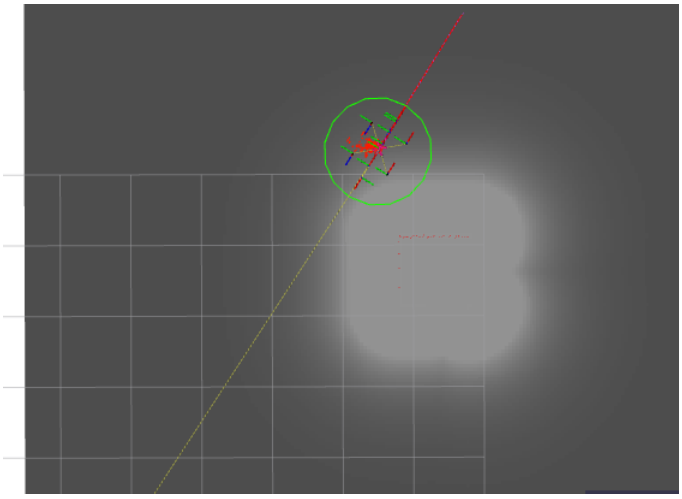

Figure 3.13: Obstacle representation in the local map

Chapter 4

Simulation

Simulation is one of the essential aspects of software development; it is a necessary step to verify and discover the potentiality and problems of a newly designed algorithm or software component. In a simulation, we have a protected environment to verify our application's functionality and see how it interacts with the other entities, analyzing the results rapidly. Using this strategy the developer can directly test new features without the assistance of the final hardware platform, and also without the risk of damaging the equipment or people. Simulation is useful also to better understand the behaviour and find the critical aspects of the new function and its integration with other software components. Moreover it is possible perform regression testing, and train AI system using realistic scenarios. In this chapter will be described the simulation tools and the environments used for the simulation. At the end of the chapter the performance of the autonomous navigation application will be evaluated by the simulation results.

4.1 Simulation Tools

The seamless navigation application,which has been described in the previous chapter (3), has been tested using the Gazebo simulator. In the ROS2 distribution is also provided the Rviz2 visualization tool that allows to visualize the data that the model of the robot receives from the simulation.

4.1.1 Gazebo simulator

Gazebo is an open-source 3D robotics simulator that can be integrated with ROS through a set of packages and since the beginning has been designed "to simulate robots in outdoor environments under various conditions" [40]. It is a realistic and highly customizable simulation environment with the use of new plugins and new structures. Through the GUI it is also possible to interact with the simulation, build new environments using the available models, or new custom ones. It is also possible to take screenshots, or create video presentation. Gazebo has been used in various technology challenges and competitions, including the DARPA Robotics challenge or the NASA space robotics challenge [41]. Its main technical features are:

- Dynamics Simulation: it supports various high-performance physics engines (ODE, Bullet, Simbody, DART) to reproduce the 3D dynamic motion of the robots and their interaction with the surroundings.

- Advanced 3D Graphics: it utilizes an open-source graphics rendering engine (OGRE) for rendering the environments in a realistic way, with lighting effects, shadows and textures.

- Sensors and Noise: it is possible to utilize a large set of simulated sensors, for example LiDARs, depth cameras, GPS or contact or force sensors.

- Plugins: custom plugins can be developed to add new functionalities and behaviours through the Gazebo API.

- Robot models: many robot models are included in the Gazebo distribution, additional models can be inported or built using the SDF format.

It is possible to create your simulation word, with structures buildings, terrain features, using both the GUI or the SDF language; the setup will be saved in a `.word` file.

4.1.2 Rviz2

Rviz2 is the second version of a 3D visualization tool, it is used to analyze and visualize the internal state of the robot and its perception of the environment. In Rviz2 through the GUI it is possible to choose which information and about the robot or sensor measurement we want to visualize. It is possible for example to visualize in real time the robot's configuration or the images published by a camera. A useful feature is the possibility to load a map; on the static map the costmaps will highlight the obstacles and the other features of the environment. Through the navigation plugins it is possible to directly control the robot in real-time, asking it to navigate to a specific location or through a route. Rviz2 can be used either when the robot is tested in a Gazebo simulation environment or when it is in real surroundings. Using both Gazebo and Rviz2 is possible to fully have a perception about the application work on the robot, promptly confronting the new features and modifications, and analyzing as it works under various conditions. This is particularly indicated for localization applications, obstacle avoidance and path planning.

4.2 Husky model

In order to work in simulation is needed a realistic model of the Husky UGV. This model is provided directly by the Clearpath Robotics, by a Unified Robot Description Format (URDF) file,which "is a file format for specifying the geometry and organization of robots in ROS" [25]. The Husky model comprehend a complete 3D description of all the body parts and surfaces also using 3D meshes. Every body part is also correlated with their dynamical characteristics (stiffness, mass, inertia,...) that will be than used by Gazebo to simulate the UGV motion more accurately. It is also possible to add various sensor to the robot model and their output will be also simulated by Gazebo.

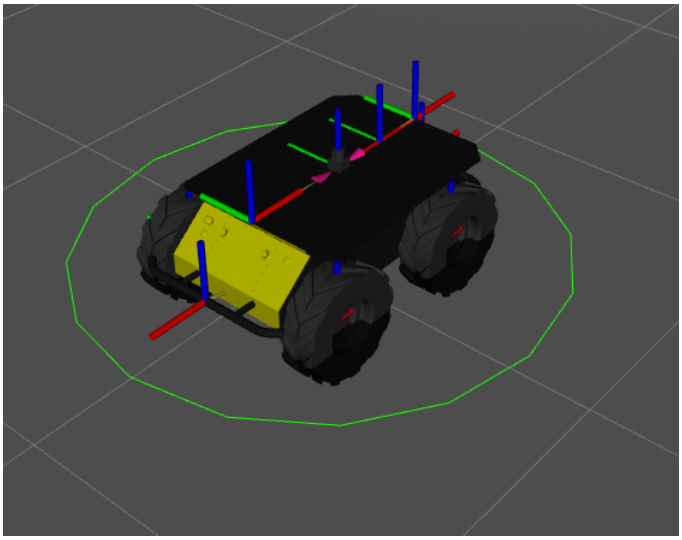

Figure 4.1: Husky UGV model

4.2.1 Husky setup

In order to obtain satisfactory performances during localization and in obstacle avoidance the Husky URDF model has been equipped with the following sensors:

- a GPS receiver: which tracks the GPS location of the robot and its velocity.
- a 2D LiDAR: to scan the surroundings in search for obstacles or feature of the environment.
- an IMU: to accurately track the orientation of the UGV and esimate its acceleration.
- a depth camera (optional): to dynamically track obstacles of different heights.

The model of each sensor is optionally loaded at the start of the simulation and Gazebo will simulate and forward the measurement to the ROS topics defined in the URDF file of these sensors. Using the gazebo plugins it is also possible to reproduce the noise for the measurements to have more realistic working conditions.

4.3 Simulation setup

The simulation environment can be highly customized. It is possible to add buildings and modify the terrain configuration, an also set the WGPS coordinates of the origin of the environment in order to simulate a GPS receiver.

4.3.1 Simulation environment

The simulation environment used is called `small_town.world`, an open source available word [42] that represent a small outdoor environment with defined borders. The environment has a planar surface, mostly composed of asphalt and a playground in the middle as we can note in the figures.

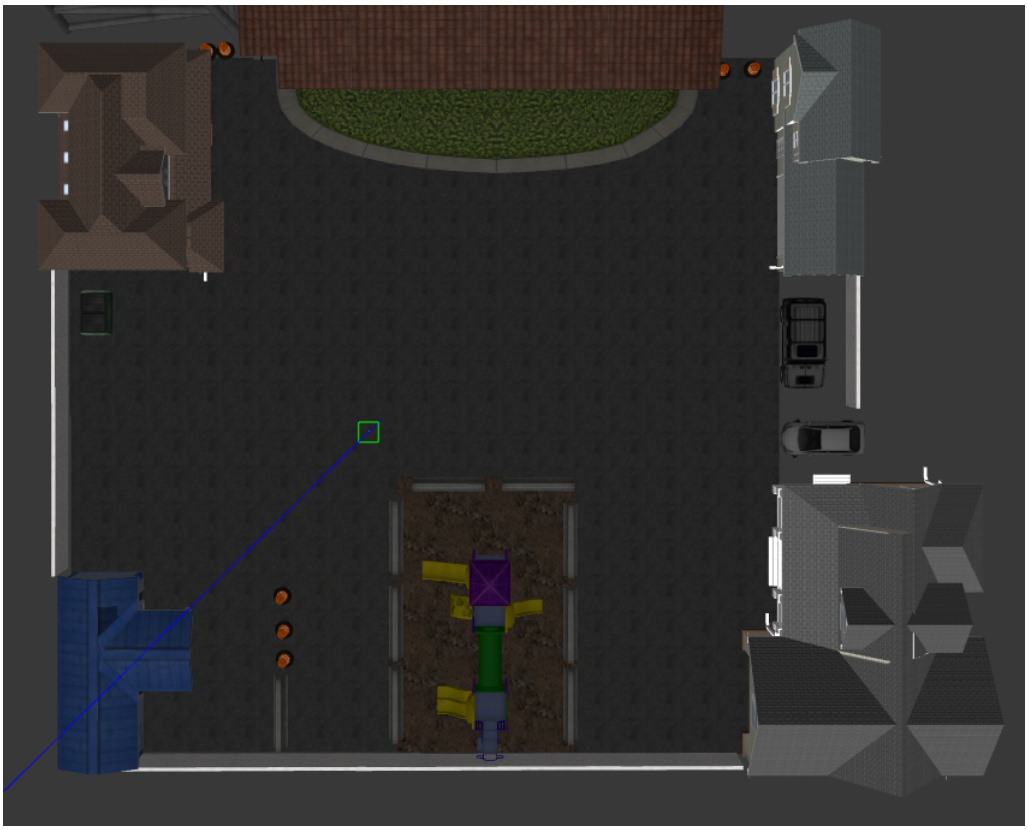

Figure 4.2: *small_town.world* [42]

The environment is mostly free of overhead impediments that may deteriorate

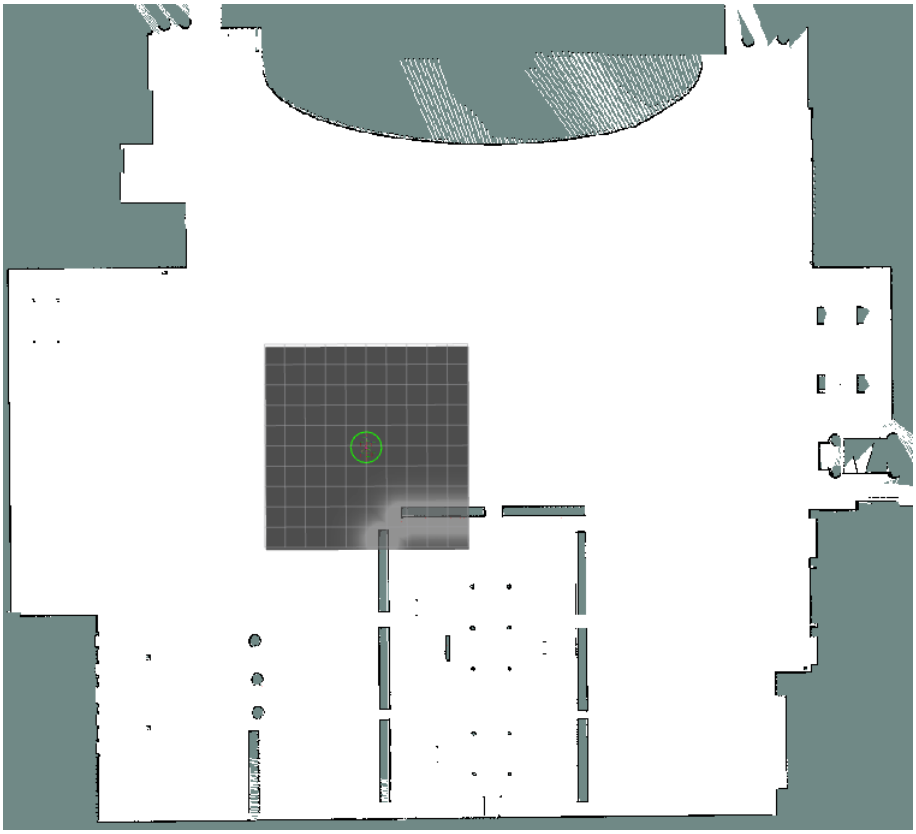

Figure 4.3: Map of *small_town.world* [42]

the GPS signal, as a result we expect to obtain a pretty accurate estimate of the Husky configuration by the Extended Kalman Filter. At the contrary since there are a lot of open spaces the estimate provided AMCL node may not be as accurate in these area because they are free of features and obstacle which may be used as reference to incorporate LiDAR measurements. The simulation world is delivered together with the relative map generated using `slam_toolbox`.

4.3.2 Simulation Results

In order to properly evaluate the results of the simulation from the Gazebo simulation we have extracted the real path performed by the Husky UGV using a plugin that publishes the real pose at each instant.
Looking at the images we can distinguish the real pose and the the estimates

provided by the two filters.

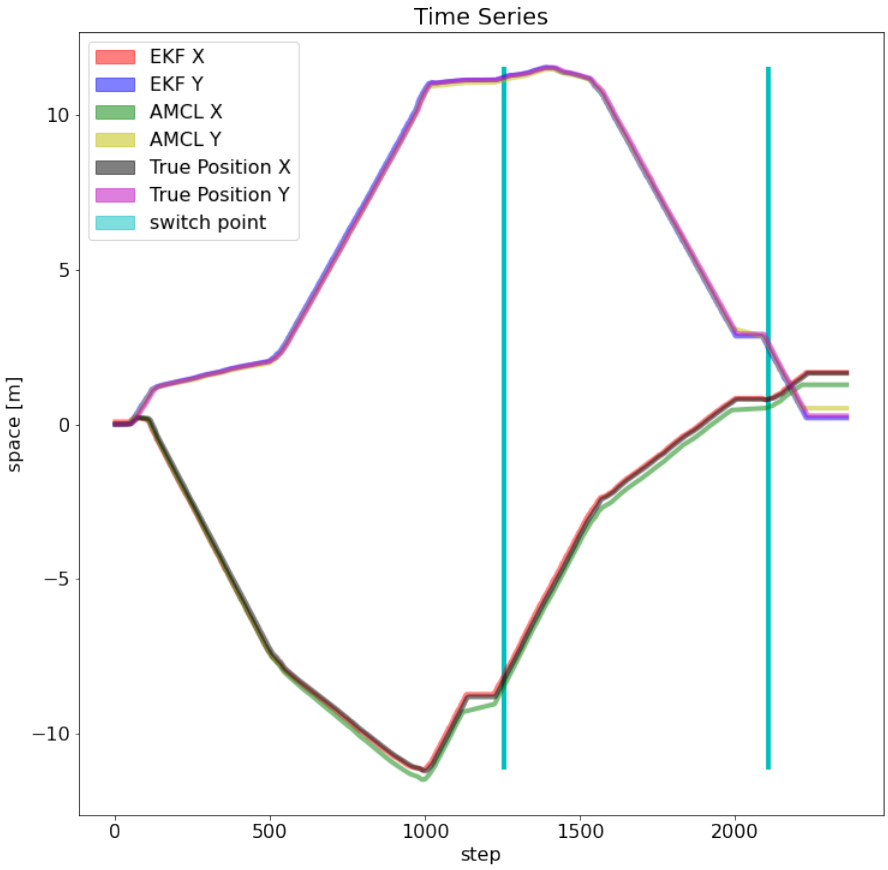

Figure 4.4: Time evolution of x and y position coordinates of the Husky UGV

From the following plots we can note that the results are satisfactory since the localization error remains limitated during the full route of the UGV.

We can notice that the EKF node have greater accuracy with respect to the AMCL, because as evaluated before there are not enough features in the environment to provide an accurate localization using LIDAR scans. Another issue is the EKF estimation which constantly anticipates the true position during the whole simulation. This may arise from a sub-optimal choice in the EKF parameters.

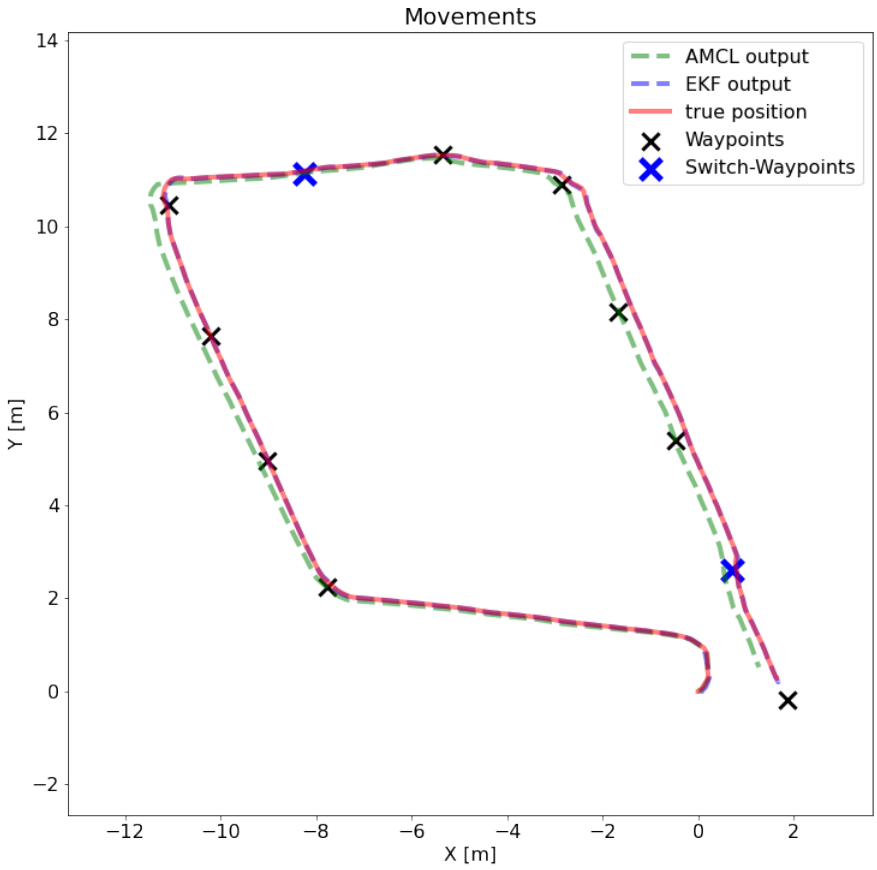

Figure 4.5: Comparison of the estimated path through the waypoints

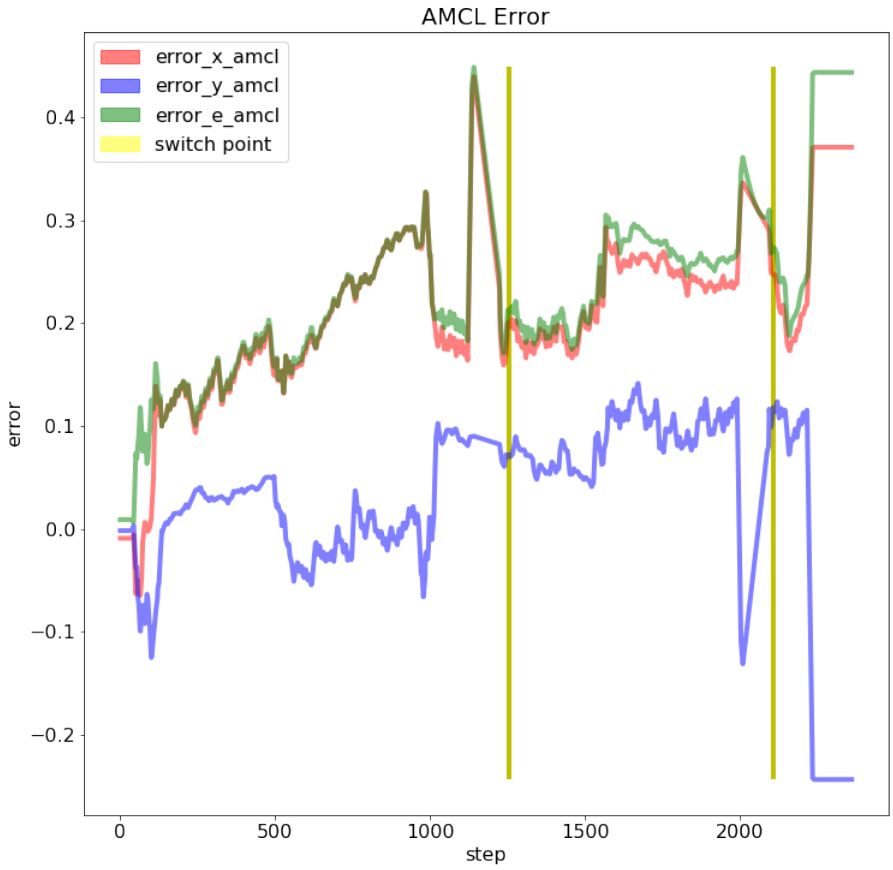

Figure 4.6: AMCL estimate localization error

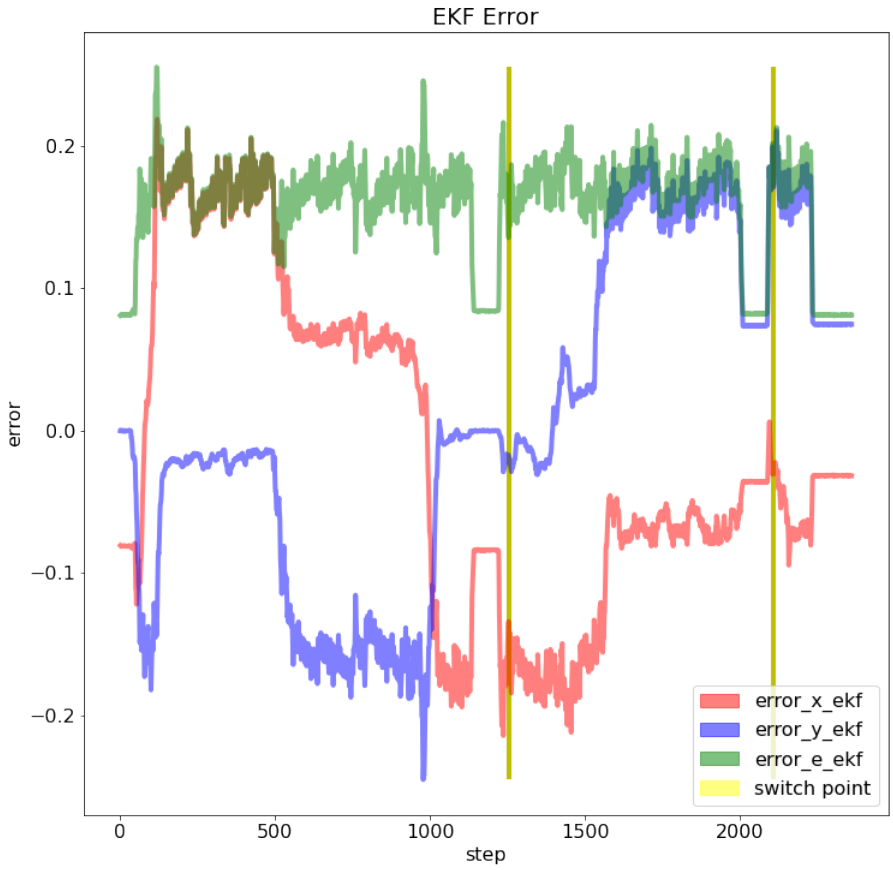

Figure 4.7: EKF estimate localization error

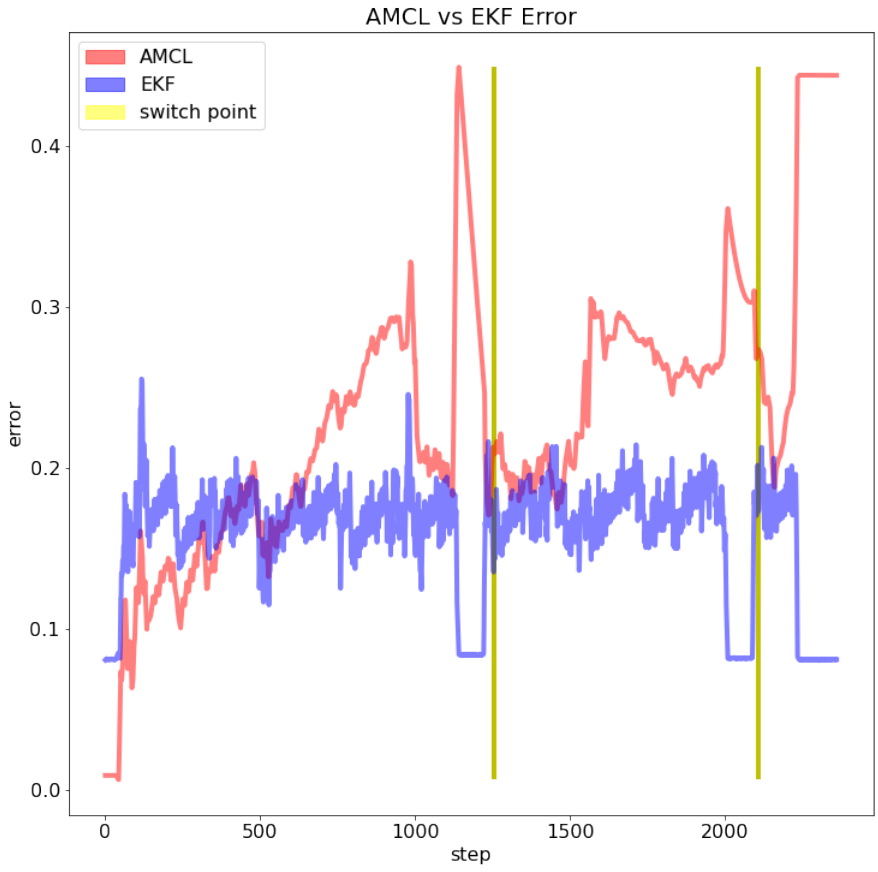

Figure 4.8: Estimate localization error comparison between AMCL and EKF

Chapter 5

Conclusion

In this chapter we will analyze the main characteristics of the proposed solution, focusing on the strengths of this approach and the possible weaknesses and improvements that can be made to achieve better results.

5.1 Main characteristics of the seamless navigation system

The implementation of the navigation application described in this thesis is designed in order to be:

- Adaptable: since this approach is has very general starting points it can be applied for working under various conditions and environments, its only limitation is to have a map in a ROS compatible format for indoor environments, which can be overcome by using SLAM techniques.
- Modular: since it is implemented as a collection of ROS packages it is really portable and new features can be added as new packages, moreover new task and routines can added to the application as plugins.
- Safe: It is capable of dynamically detect and avoid obstacles through the Dynamic Window Approach and to make a transition to recovery state if it remain stuck or other problem arises.

The work done till now still has limitation that leave space for improvements:

- Only consider obstacles on a planar plane: this problem can be easily overcome by adding the support to 3D sensors such as 3D LiDARs or depth cameras and add their measures to the local costmap through a voxel layer.
- Improve and smooth the indoor-outdoor transition: since in this point we are changing the estimation method the transition in not always smooth because the 2 estimates are not always aligned, some reference or landmark can be used in those point to reduce the difference between the two estimate before transition.
- Tuning the localization filter parameters: as seen in the result part the EKF is slightly in advance with respect to the true state of the robot, this can be reduced by tuning the filter parameters.
- Implement new motion models for the AMCL: a sample motion model for a skid-drive system can be added in order to increase the accuracy of the motion model transition.
- Implement new sensor models in the AMCL: new sensor such as a depth camera or GPS can be used to implement new measurement updates and have a better correction of the prediction estimate.

The autonomous navigation application has provided satisfactory results in simulation environments and it is currently ready to be deployed on the real hardware.

5.2 Next steps

The application will be deployed on the Husky UGV of Clearpath Robotics, to see how it behaves on real environments. Some improvements will be made to integrate a depth camera, as described in the previous section, in order to have a working solution for avoiding dynamic obstacles positioned at different heights. All considered, this solution can be seen as a starting point to reach a fully autonomous navigation system.

Bibliography

[1] Pablo Gonzalez-De-Santos, Roemi Fernández, Delia Sepúlveda, Eduardo Navas, and Manuel Armada. «Unmanned Ground Vehicles for Smart Farms». In: *Agronomy*. Ed. by Amanullah. Rijeka: IntechOpen, 2020. Chap. 6. DOI: 10.5772/intechopen.90683. URL: https://doi.org/10.5772/intechopen.90683 (cit. on pp. 1, 2).

[2] D. Kent Shannon, David E. Clay, and Kenneth A. Sudduth. «An Introduction to Precision Agriculture». In: *Precision Agriculture Basics*. John Wiley & Sons, Ltd, 2018. Chap. 1, pp. 1–12. ISBN: 9780891183679. DOI: https://doi.org/10.2134/precisionagbasics.2016.0084. eprint: https://acsess.onlinelibrary.wiley.com/doi/pdf/10.2134/precisionagbasics.2016.0084. URL: https://acsess.onlinelibrary.wiley.com/doi/abs/10.2134/precisionagbasics.2016.0084 (cit. on pp. 1, 2).

[3] Amitava Chatterjee, Anjan Rakshit, and N. Nirmal Singh. «Mobile Robot Navigation». In: *Vision Based Autonomous Robot Navigation: Algorithms and Implementations*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 1–20. ISBN: 978-3-642-33965-3. DOI: 10.1007/978-3-642-33965-3_1. URL: https://doi.org/10.1007/978-3-642-33965-3_1 (cit. on p. 2).

[4] Tod S. Levitt and Daryl T. Lawton. «Qualitative navigation for mobile robots». In: *Artificial Intelligence* 44.3 (1990), pp. 305–360. ISSN: 0004-3702. DOI: https://doi.org/10.1016/0004-3702(90)90027-W. URL: https://www.sciencedirect.com/science/article/pii/000437029090027W (cit. on p. 2).

[5] Bruno Siciliano and Oussama Khatib. *Springer Handbook of Robotics*. Berlin, Heidelberg: Springer-Verlag, 2007. ISBN: 354023957X (cit. on pp. 3–5).

[6] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. ISBN: 0262201623 (cit. on pp. 3, 7).

[7] Ningbo Li, Lianwu Guan, Yanbin Gao, Shitong Du, Menghao Wu, Xingxing Guang, and Xiaodan Cong. «Indoor and Outdoor Low-Cost Seamless Integrated Navigation System Based on the Integration of INS/GNSS/LIDAR System». In: *Remote Sensing* 12.19 (2020). ISSN: 2072-4292. DOI: 10.3390/rs12193271. URL: https://www.mdpi.com/2072-4292/12/19/3271 (cit. on p. 4).

[8] Henrik I. Christensen and Gregory D. Hager. «Sensing and Estimation». In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 87–107. ISBN: 978-3-540-30301-5. DOI: 10.1007/978-3-540-30301-5_5. URL: https://doi.org/10.1007/978-3-540-30301-5_5 (cit. on p. 10).

[9] «Actuators and Sensors». In: *Robotics: Modelling, Planning and Control*. London: Springer London, 2009, pp. 191–231. ISBN: 978-1-84628-642-1. DOI: 10.1007/978-1-84628-642-1_5. URL: https://doi.org/10.1007/978-1-84628-642-1_5 (cit. on p. 11).

[10] Gregory Dudek and Michael Jenkin. «Inertial Sensors, GPS, and Odometry». In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 477–490. ISBN: 978-3-540-30301-5. DOI: 10.1007/978-3-540-30301-5_21. URL: https://doi.org/10.1007/978-3-540-30301-5_21 (cit. on pp. 12, 13).

[11] Wikipedia contributors. *Global Positioning System — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-March-2022]. 2022. URL: https://en.wikipedia.org/w/index.php?title=Global_Positioning_System&oldid=1079313567 (cit. on p. 12).

[12] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. Chap. 2. ISBN: 0262201623 (cit. on pp. 13–17).

[13] Stepanov, O. A. (15 May 2011). "Kalman filtering: Past and present. An outlook from Russia. (On the occasion of the 80th birthday of Rudolf Emil Kalman)". Gyroscopy and Navigation. 2 (2): 105. doi:10.1134/S2075108711020076| S2CID 53120402. (cit. on p. 17).

[14] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. Chap. 3. ISBN: 0262201623 (cit. on p. 18).

[15] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. Chap. 4. ISBN: 0262201623 (cit. on p. 19).

[16] ROS.org, http://wiki.ros.org/amcl (cit. on p. 20).

[17] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. Chap. 8. ISBN: 0262201623 (cit. on pp. 20, 21).

[18] Sebastian Thrun, Wolfram Burgard, and Dieter Fox. *Probabilistic Robotics (Intelligent Robotics and Autonomous Agents)*. The MIT Press, 2005. Chap. 9. ISBN: 0262201623 (cit. on p. 22).

[19] Edsger W Dijkstra. «A note on two problems in connexion with graphs». In: *Numerische mathematik* 1.1 (1959), pp. 269–271 (cit. on pp. 24, 25).

[20] Wikipedia. *Algoritmo A* — Wikipedia, L'enciclopedia libera*. [Online; in data 27-marzo-2022]. 2022. URL: `http://it.wikipedia.org/w/index.php?title=Algoritmo_A*&oldid=126018331` (cit. on pp. 27, 28).

[21] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. «Exploring the performance of ROS2». In: *Proceedings of the 13th ACM SIGBED International Conference on Embedded Software (EMSOFT)*. 2016, pp. 1–10 (cit. on p. 26).

[22] *Why Ros?* URL: `https://www.ros.org/blog/why-ros/` (cit. on p. 26).

[23] *A history of ROS (robot operating system)*. July 2020. URL: https://www.theconstructsim.com/history-ros/ (cit. on p. 26).

[24] *The ros ecosystem*. URL: https://www.ros.org/blog/ecosystem/ (cit. on p. 26).

[25] *Ros Concepts*. URL: https://docs.ros.org/en/galactic/Concepts.html (cit. on pp. 29, 31, 57).

[26] Yuya Maruyama, Shinpei Kato, and Takuya Azumi. «Exploring the performance of ROS2». In: *2016 International Conference on Embedded Software (EMSOFT)*. 2016, pp. 1–10. DOI: 10.1145/2968478.2968502 (cit. on pp. 29, 31, 32).

[27] *About Ros 2 interfaces*. URL: https://docs.ros.org/en/foxy/Concepts/About-ROS-Interfaces.html (cit. on p. 30).

[28] *Understanding Ros 2 services*. URL: https://docs.ros.org/en/foxy/Tutorials/Services/Understanding-ROS2-Services.html (cit. on p. 30).

[29] *Understanding Ros 2 actions*. URL: https://docs.ros.org/en/foxy/Tutorials/Understanding-ROS2-Actions.html (cit. on p. 30).

[30] *About different Ros 2 DDS/RTPS vendors*. URL: https://docs.ros.org/en/foxy/Concepts/About-Different-Middleware-Vendors.html (cit. on p. 32).

[31] URL: https://design.ros2.org/articles/ros_on_dds.html (cit. on p. 32).

[32] *About internal ROS 2 interfaces*. URL: https://docs.ros.org/en/foxy/Concepts/About-Internal-Interfaces.html (cit. on p. 33).

[33] *Robot_localization wiki*. URL: http://docs.ros.org/en/noetic/api/robot_localization/html/index.html (cit. on p. 36).

[34] *Nav2*. URL: https://navigation.ros.org/index.html (cit. on pp. 41, 42).

[35] Ros-Planning. *nav2_waypoint_follower*. URL: https://github.com/ros-planning/navigation2/tree/main/nav2_waypoint_follower (cit. on p. 42).

[36] *Navigate through poses*. URL: https://navigation.ros.org/behavior_trees/trees/nav_through_poses_recovery.html (cit. on pp. 44–46).

[37] Steve Macenski. *Smac Planner*. URL: https://github.com/ros-planning/navigation2/tree/main/nav2_smac_planner (cit. on p. 49).

[38] *DWA local planner*. URL: http://wiki.ros.org/dwa_local_planner (cit. on p. 51).

[39] Kaiyu Zheng. *Ros Navigation Tuning Guide*. 2019. URL: https://arxiv.org/abs/1706.09068v2 (cit. on p. 51).

[40] Osrf. *Why gazebo?* URL: http://gazebosim.org/ (cit. on p. 56).

[41] Wikipedia contributors. *Gazebo simulator — Wikipedia, The Free Encyclopedia*. [Online; accessed 27-March-2022]. 2021. URL: https://en.wikipedia.org/w/index.php?title=Gazebo_simulator&oldid=1039967070 (cit. on p. 56).

[42] Author automaticaddison. *Smalltown gazebo world*. 2021. URL: https://automaticaddison.com/ (cit. on pp. 59, 60).