



FFI Norwegian Defence
Research Establishment

26/001

FFI-RAPPORT

Development of off-road autonomy for unmanned ground vehicles

Marius Thoresen
Magnus Baksaas
Niels Hygum Nielsen
Eilert André Mentzoni
Morten Hanevold
David Kolden
Brage G. Eikanger
Kim Mathiassen

Development of off-road autonomy for unmanned ground vehicles

Marius Thoresen
Magnus Baksaas
Niels Hygum Nielsen
Eilert André Mentzoni
Morten Hanevold
David Kolden
Brage G. Eikanger
Kim Mathiassen

Keywords

Autonomi
Ubemannede bakkekjøretøyer (UGV)
Autonome biler
Eksperimenter
Programvare

FFI report

26/001

Project number

1677

Online ISSN

2704-2383

Approvers

Kim Mathiassen, *Research Manager*
Halvor Ajer, *Director of Research*

The document is electronically approved and therefore has no handwritten signature.

Copyright

© Norwegian Defence Research Establishment (FFI). The publication may be freely cited where the source is acknowledged.

Summary

Unmanned ground vehicles (UGVs) are increasingly recognized as assets that can benefit military operations, offering the potential to reduce risk to personnel. Having autonomous capabilities on board UGVs can further enhance their utility by reducing operator workload and enabling more mission capabilities.

This report describes the development of a modular autonomy stack for UGVs and its evaluation through experiments in off-road terrain. The autonomy stack integrates three core capabilities – perception, motion planning, and trajectory tracking – implemented on our Milrem Themis UGV platform, Tor. This autonomy system relies on onboard sensors and local mapping to navigate without predefined paths or precise satellite positioning, aiming to address the fundamental challenge of autonomous terrain traversability.

To measure performance, we defined a set of benchmark routes representing diverse terrain types, from open fields and dirt roads to narrow trails and forests. These routes serve as structured test cases for assessing autonomy requirements and identifying limitations of our system. We have conducted experiments to assess our autonomy system at FFI's LandX demonstrations and in long-range trials. The experiments revealed that Tor with our autonomy stack can successfully navigate simple environments such as open terrain and wide vehicle trails, but struggles in complex scenarios involving dense vegetation and narrow passages. Key limitations include the lack of semantic terrain classification and robust decision making in challenging scenarios. Predictability and reliability remain major challenges, with decisions influenced by randomness rather than consistent reasoning.

While our current autonomy stack is not yet ready for operational use, it provides a solid foundation for continued research and experimentation. The benchmark routes and evaluation methodology offer a repeatable framework for measuring progress, ensuring that future iterations move closer to the ultimate goal: UGVs capable of operating effectively and autonomously in demanding environments.

Sammendrag

Ubemannede bakkekjøretøy (*unmanned ground vehicles* – UGV-er) blir i stadig større grad anerkjent som verktøy som kan styrke militære operasjoner ved å redusere risikoen for personell. Autonome funksjoner om bord på UGV-er kan ytterligere øke nytteverdien ved å redusere belastningen på UGV-operatørene og muliggjøre flere evner til å løse oppdrag.

Denne rapporten beskriver utviklingen av et modulært autonomisystem for UGV-er og evalueringen av dette gjennom eksperimenter i terreng utenfor vei. Autonomisystemet integrerer tre kjernefunksjoner – persepsjon, bevegelsesplanlegging og banefølgning – implementert på vår Milrem Themis-plattform, Tor. Autonomisystemet baserer seg på sensorer om bord og lokal kartlegging for å navigere uten forhåndsdefinerte ruter eller presis satellittposisjonering, med mål om å løse den grunnleggende utfordringen med autonom fremkommelighet i terreng.

For å måle ytelse har vi definert et sett med referanseruter som representerer ulike terrengetyper, fra åpne områder og grusveier til smale stier og skog. Disse rutene fungerer som standardiserte testscenarier for å vurdere autonomiytelse og identifisere systemets begrensninger. Vi har gjennomført eksperimenter for å evaluere vårt autonomisystem på LandX-demonstrasjonene og i langdistanseforsøk. Eksperimentene viste at Tor med vårt autonomisystem kan navigere vellykket i enkle miljøer som åpent terreng og brede kjørespor, men har utfordringer i komplekse scenarier med tett vegetasjon og smale passasjer. Viktige begrensninger inkluderer mangel på semantisk terrengklassifisering og robust beslutningstaking i utfordrende situasjoner. Forutsigbarhet og pålitelighet er fortsatt store utfordringer, med beslutninger som av og til påvirkes av tilfeldigheter i stedet for konsistent logikk.

Selv om systemet vårt ennå ikke er klart for operativ bruk, gir det et solid grunnlag for videre forskning og eksperimentering. Referanserutene og evalueringsmetodikken gir en repeterbar ramme for å måle fremgang og sikre at fremtidige iterasjoner beveger seg nærmere det endelige målet: UGV-er som kan operere effektivt og autonomt i krevende miljøer.

Contents

Summary	3
Sammendrag	4
Acronyms	7
1 Introduction	9
1.1 Goals for the autonomy development	10
1.2 Necessary autonomy capabilities	13
1.3 Report Structure	15
2 Tor – Our UGV platform	16
2.1 The Themis UGV	16
2.2 Hardware additions for autonomy system	18
2.3 Controlling the Themis through software	19
3 Autonomy stack	24
3.1 Autonomy stack overview	24
3.2 Mission control – HAL and Norne	27
3.3 Localization	32
3.4 Warpath – Sensor fusion for situational awareness	39
3.5 Trailblazer – Deciding where to drive	57
3.6 Trajectory tracker	70
4 The LandX demonstrations	75
4.1 LandX21	75
4.2 LandX22	76
5 Long-range autonomous drive experiment	79
5.1 Experiment 1 – Open and flat terrain	80
5.2 Experiment 2 – Wide and open vehicle trail	82
5.3 Experiment 3 – Dirt Road	86
5.4 Experiment 4 – Varied terrain through forest	88
5.5 Long-range driving experiment conclusions	95
6 Conclusion	98
References	99
Appendix	
A Benchmark routes and autonomy requirements	103

A.1	Route A – Open and flat terrain	103
A.2	Route B – Dirt road	103
A.3	Route C – Wide and open vehicle trail	104
A.4	Route D – Vehicle trails in steep inclines	105
A.5	Route E – Narrow vehicle trail without obstacles	106
A.6	Route F – Narrow vehicle trail with small obstacles	107
A.7	Route G – Sparse and flat pine forest	107
A.8	Route H – Dense forest	107
B	Platooning controller	109
B.1	Path between vehicles	109
B.2	Forward velocity control	109

Acronyms

AUV Autonomous Underwater Vehicle

CAD Computer Aided Design

CPU Central Processing Unit

DLT Direct Linear Transform

DSU Data Synchronization Unit

ECEF Earth-centered, Earth-fixed

FFI Norwegian Defence Research Establishment

FMM Fast Marching Method

FPGA Field Programmable Gate Arrays

GNSS Global Navigation Satellite System

GPU Graphics Processing Unit

IMU Inertial Measurement Unit

INS Inertial Navigation System

IOP Interoperability Profile

lidar Light Detection And Ranging

NED North-East-Down

NTP Network Time Protocol

PoE Power over Ethernet

PPS Pulse Per Second

PTP Precision Time Protocol

ROS Robot Operating System

SIL Software-In-the-Loop

SSD Single-shot Detector

UGV Unmanned Ground Vehicle

YOLO You Only Look Once

ZUPT Zero Velocity Update

1 Introduction

Unmanned Ground Vehicles (UGVs) are robotic systems that operate on the ground without an onboard human presence. Over the past decade, UGVs have evolved significantly. Ten years ago, they were primarily limited to small, remotely driven bomb disposal robots and other specialized tasks. Today there are self-driving vehicles on the roads, and advancements have spread to the field of military vehicles as well.

In military contexts, UGVs have the potential to revolutionize land warfare by removing soldiers from dangerous or repetitive tasks and providing longer operational endurance. Some potential military applications for UGVs are, for example, logistics, reconnaissance, casualty evacuation, or as a weapons platform[24, 25, 29, 4, 39, 31].

One approach to using UGVs is teleoperation, where an operator remotely controls the vehicle. The direct control allows for precise maneuvering and real-time decision-making. At the same time, it presents technical challenges, such as requiring a high-bandwidth communication link and the ability to convey situational awareness to the operator. Additionally, teleoperation requires a dedicated operator, who cannot focus on other tasks and may be subject to fatigue. For some tasks such as mine clearing and bomb disposal, this might not be an issue, but for other tasks, such as patrolling, it may be a significant drawback.

UGVs could leverage advancements in autonomy from the civilian industry, enabling them to drive autonomously. Autonomous UGVs can replace the operator for certain tasks, such as routine patrols, obstacle avoidance, and navigation in structured environments. This autonomy frees up the operator to focus on other critical tasks, such as mission planning and monitoring, thereby enhancing overall situational awareness (SA). Moreover, reducing the need for constant operator input can decrease fatigue and allow the operator to maintain a higher level of focus on the mission. Autonomous systems also do not require the same level of situational awareness and bandwidth as teleoperation, making them more efficient in certain scenarios.

Today there are several commercially available UGVs with varying levels of autonomy. These vehicles are being tested and fielded by armies around the world. While many have demonstrated impressive capabilities in autonomous off-road driving, they are still not able to operate without human interventions, and the autonomy is still under active development. Significant advancements are necessary before they achieve the maturity required for reliable use in military operations[36].

While the industry is good at developing complete products with significant resources, both FFI and the industry are still exploring how UGVs will be used. The unique environment in Norway, with its challenging terrain and harsh winter conditions, requires UGVs to operate effectively in scenarios that may not be prioritized by commercial developers. In addition, the Norwegian Armed Forces may have specific requirements for UGVs that may not be met by commercially available systems in the future. By developing our own autonomy systems, we gain the flexibility to experiment with UGVs in a wider range of applications, ensuring they meet our specific operational intentions. This approach also allows for greater flexibility in integrating our UGVs with sensors, payloads, and other systems, enabling us to conduct research on their efficient use and interaction within a larger operational context as well.

FFI has been conducting research and experimentation with autonomy for unmanned ground vehicles since 2014. From 2015 to 2018, we used a Polaris Ranger modified for electronic control

as a demonstrator platform for autonomous driving[27]. The goal from the beginning was to demonstrate driving in off-road conditions, and we achieved some results[51] with driving in the terrain. In 2019, we switched to using the more capable Milrem Themis robotic platform, and continued developing our autonomy system [26]. This report presents the autonomy system we developed during this period up through 2022, as well as experiments we conducted to evaluate the performance of our autonomy for UGVs in off-road environments.

1.1 Goals for the autonomy development

The main goals of the project are to demonstrate autonomous driving and the autonomous execution of tasks in challenging terrain. At the core of these goals is the ability to drive autonomously to given positions. Without this capability, more complex tasks, such as autonomous patrolling, logistics and transport, reconnaissance, and casualty evacuation (casevac) cannot be performed.

The real challenge for the autonomy system is driving when the terrain gets rough. It is challenging to correctly analyze the surroundings and determine where the terrain is traversable and where it is not. Terrain features such as roughness, slopes, holes and ditches, soil type, snow, and vegetation all affect the traversability of the terrain and can be hard to determine for an autonomous system. Without correctly classifying the terrain, the UGV risks being overly optimistic and getting stuck or hitting obstacles. Conversely, if it is too cautious, it could fail to find ways through and become stuck behind perceived obstacles, avoiding areas it could traverse. Due to the complex nature of terrain traversability, autonomous driving through these environments is still a largely unsolved problem.

Yet, for a UGV to be useful for military missions, movement through terrain is often necessary. As terrain driving is the hardest challenge to achieve, this is the defining challenge for determining if an autonomous UGV can be successful. For this reason, the focus of the project is to develop autonomy for driving in the roughest terrain possible. If we can achieve this goal, we consider the remaining challenges to be simpler and solvable in the future.

1.1.1 Benchmark routes

To evaluate our success in achieving autonomous terrain driving, we have established a benchmark. The benchmark consists of a set of routes in different types of terrain. We can attempt to drive these routes autonomously and assess how successful the UGV is in navigating each route. The idea of benchmark routes serves several purposes:

- It establishes an overall measurement of how far along the project is in developing autonomy for terrain driving.
- By defining routes based on our goals rather than our current capabilities, we avoid a bias where we only design experiments we know we can complete.
- By having a set of test routes ranging in complexity from simple to practically impossible, we ensure that we always push the limits of the autonomy. Starting simple means we have achievable initial goals and do not have to solve all problems at once.

-
-
- A diverse set of routes allows us to discover development challenges and areas that need focus during development.

The benchmark routes are not necessarily meant to be driven precisely, but can serve as representative samples of different types of terrain. Nevertheless, there is still value in repeating trials with the exact same routes, as repetition makes it easier to compare trial results.

The routes have large distances between the waypoints, typically 20–100 m, and the UGV must therefore navigate between waypoints without following a predefined path. Instead, it must rely on its perception system to determine where it is possible to drive and plan a path towards the next waypoint. This also means that the UGV does not need to rely on accurate GNSS (Global Navigation Satellite System, e.g. GPS) localization, as it will need to decide where to drive based on its perception system.

We have selected a collection of eight different routes:

Route A Open and flat terrain, without vegetation. Obstacles are large and stand out, for example, trees.

Route B Dirt road.

Route C Wide and open vehicle trail.

Route D Vehicle trails in steep inclines. Both ascent and descent.

Route E Narrow vehicle trail without small obstacles (stones, tree stumps, etc.)

Route F Narrow vehicle trail with small obstacles (stones, tree stumps, etc.)

Route G Sparse and flat forest.

Route H Dense forest, yet still traversable.

The selected routes are shown in Figure 1.1a, they are all from Rødsmoen military exercise area at Rena, Norway, and a typical environment is shown in Figure 1.1b. The routes represent different types of environments, each with their own challenges. We expect the first ones to be the easiest to complete, with the difficulty increasing throughout the list. Ideally, we would want routes from different areas as well, as all the selected have roughly the same type of vegetation and soil type. However, the routes will serve to properly challenge the autonomy system, and we can always add more routes in the future. For more details on the routes and their individual challenges, see Appendix A.

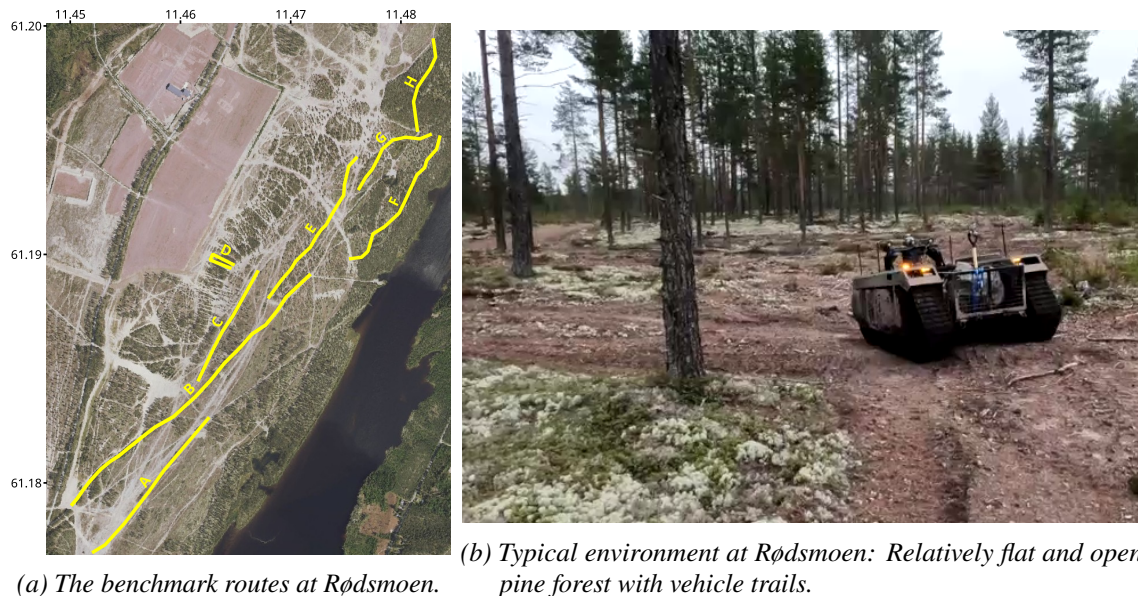


Figure 1.1 Aerial view and typical environment of the benchmark routes at Rødsmoen.

1.1.2 Secondary goals for the autonomy system

While developing autonomy for driving the benchmark routes is the main focus of the project, it is not the only goal. Ultimately, we aim to create a UGV that functions as a general-purpose platform, capable of being used in various scenarios and applications. This requires an autonomy system that can be effectively operated in the field. With a more usable UGV, we can start experimenting with how we can use UGVs in different applications, while continuing to develop the autonomy in parallel.

One feature for a more usable UGV is dynamic tasking. We want the ability for operators to click a point on a map and have the UGV autonomously navigate to that location. Additionally, it is crucial to provide the ability to change the UGV's route dynamically in response to new information or changing mission requirements. If the UGV gets stuck or encounters a situation it cannot handle autonomously, operators should be able to take manual control. Furthermore, the system should allow operators to reset the autonomy and attempt a different approach if the UGV fails to navigate a particular section. To ensure the UGV is practical for field use, the operator must control the system in a user-friendly manner. This involves developing an intuitive user interface that is straightforward and efficient to use.

Another mode of operation we aim to explore is the ability for the UGV to follow an object, such as a person or a vehicle. This follow-mode operation could significantly enhance the versatility of the UGV, allowing it to support a variety of missions where maintaining a specific distance from a moving target is beneficial. This capability would be particularly useful in scenarios such as convoy operations, logistics tasks and close support roles.

By keeping these secondary goals in mind, we aim to develop an autonomy system that is not only capable of navigating challenging terrains but also practical and versatile for a wide range of field experiments.

1.2 Necessary autonomy capabilities

We have defined goals for what we want to achieve with autonomy, and now we can explore how we can achieve those goals by looking at what capabilities are required for the autonomy system. With a basis in the benchmark routes, we can start defining requirements that we believe are the minimum necessary for completing them.

We can divide the autonomy capabilities into three core categories:

Perception How the UGV perceives its surrounding environment, and in doing so determines where it is possible to drive. It could produce binary obstacle maps, classify the environment into different categories, or create a cost map which associates a virtual cost with driving in different areas. Without perception, the UGV cannot react to its environment; it would only be able to follow pre-programmed paths.

Motion planning Planning a movement forward where it is possible to drive, based on what the UGV perceives. There are two available strategies: Reactive collision avoidance where the UGV reacts to the nearest obstacle and steers away from it, and deliberate motion planning where the planner uses the obstacle map to plan a path to the goal that avoids obstacles. The reactive strategy is simpler, but depending on how the environment looks, it might not always succeed.

Trajectory/path following Making the UGV follow the planned movement and creating control inputs to the UGV. The planned movement can either be simple movements or paths depending on the chosen motion planning strategy.

All of these components are necessary to achieve autonomous driving in some form. With basic implementations that provide a baseline functionality, the UGV should be able to drive autonomously in simple environments. However, when the complexity of the environment increases, so do the requirements for the autonomy system. The different routes will have different requirements for each of these components, and some will require significantly more advanced autonomy capabilities than others. For example, to complete *Route A – Open and flat terrain*, the perception system only needs to be able to differentiate between large obstacles and open space. The motion planner can be relatively simple, as it only needs to plan a path to the goal that avoids the distinct obstacles, and it can be conservative by planning wide evasive maneuvers which still complete the route. This also relaxes requirements for the trajectory follower to be dead on the planned path, as the distance between obstacles allows for larger margins for error.

On the other hand, for a route such as *Route E – Narrow vehicle trail without obstacles*, the margins of error are much smaller for all the core autonomy components. In Appendix A, we have tried to break down roughly what we believe is required of the core components to successfully complete the benchmark routes.

1.2.1 Creating an autonomy stack

To achieve the core autonomy capabilities described in Section 1.2 and Appendix A on a real UGV, we need an autonomy software framework or an *autonomy stack*. The autonomy stack will enable

us to create different software components, each with their own responsibility, and allow them to communicate with each other. The autonomy stack will contain the core components *perception*, *motion planning* and *trajectory following*, as well as other necessary modules such as sensor drivers, localization, an interface for controlling the vehicle and a way for an operator to interact with the vehicle.

From the breakdown of the routes in Section 1.2, it would appear to be easiest to implement and achieve the autonomy capabilities for Route A. It would therefore be natural to start with this route and then move on to the more challenging routes. At the same time, we do not want to develop something that would be limited to the simplest capabilities without being able to expand on it. Instead, we want a flexible autonomy stack that can contain all the different methods that we implement and test, and continue to extend and improve the autonomy system.

To achieve the flexibility we want, we have some guiding principles for the autonomy stack that we want to adhere to:

Modularity A modular design is essential for flexibility and ease of updates. This allows different components to be updated or replaced without affecting the entire system, ensuring long-term adaptability and maintainability.

Scalability We should be able to run different methods and algorithms in the system with minimal reconfigurations. This will enable us to switch between different methods for different situations, which can increase the overall capabilities of the complete system.

Platform independence The system should be scalable to accommodate different types and sizes of UGVs. This means trying to create generic interfaces to sensors and the vehicle platform itself. Achieving this would allow experimenting with different platforms with broader application across various missions and environments.

Compatible with research standards There are many open-source tools and projects available for developing autonomy systems. We should aim to be compatible with these to be able to experiment with the innovations in the field, and thereby making it easy to develop our own system.

In addition to the core components, there are other components that are necessary for the autonomy system to work, and we have some secondary requirements for these as well.

Localization We want to minimize the reliance on GNSS localization, as it may be unreliable and inaccurate. While GNSS-free localization is outside the scope for now, it may be something we want to explore in the future.

Mission manager We want a mission manager that can handle different types of missions, such as driving to a given position, following a vehicle, or patrolling an area.

Interfacing with a ground control station (GCS) We want a ground control station that can send missions to the UGV, read status information, and view output from the situational awareness system.

Vehicle interface We need a module that handles the low-level communication with the vehicle and acts as an interface for controlling the vehicle for the rest of the autonomy stack.

These components are not strictly necessary to the goal of driving in terrain. However, they provide benefits such as increasing the ease of use of the system, and the overall robustness of the system.

Also, as we continue to develop the autonomy and capabilities increase, we eventually want to use the UGV for more than driving in terrain. Having these capabilities in mind during development will facilitate us creating a system that is flexible enough to accommodate future use.

1.3 Report Structure

In this chapter we have presented the motivation for the autonomy system that we have developed, what we want to achieve with it and what types of scenarios we want to use it for. There is not a single use case, a single route we want to traverse, or a single way of using the UGV. Rather, we have a set of challenges that we believe are relevant for a wide variety of UGV applications. In this report, we present the autonomy stack we have developed as a basis for tackling these challenges, as well as results from running the stack on a UGV in off-road environments.

The report structure is as follows:

- Chapter 2 describes our UGV platform, which we have used for developing autonomy, and the adaptations we have made to make it suitable for autonomous running.
- Chapter 3 presents the autonomy stack that we have developed, with details on the individual components.
- Chapter 4 presents the UGV contributions at the LandX experiments in 2021 and 2022.
- Chapter 5 presents experiments we have conducted with our UGV in off-road environments.
- Chapter 6 comes with some concluding remarks about the autonomy system we have developed.

In addition, the report contains the following appendices:

- Appendix A lists the benchmark routes we have defined for evaluating the autonomy system, and the requirements we believe are necessary to autonomously drive them.
- Appendix B describes the platooning controller we have implemented for following a lead vehicle using visual markers.

2 Tor – Our UGV platform

To develop an autonomous UGV, it is necessary to have a flexible UGV platform we can use to test and develop the autonomy software stack. After using *Olav*[27], a Polaris Ranger 4x4 UTV as a UGV development platform in the period 2015–2018 we needed a UGV designed as a robotic vehicle that is more suited to the task. Therefore, in 2018 we purchased a Milrem Themis 4.5, a military commercial off-the-shelf UGV to replace the aging *Olav* and use it as our new UGV autonomy development platform. From 2019, we fully switched to the new Themis, which we have named *Tor*, as our primary UGV platform.

2.1 The Themis UGV

The Themis is made by the Estonian company Milrem Robotics. It is a diesel-electric tracked vehicle which is 2.4 m long, 2 m wide, 1.15 m tall and weighs 1630 kg without a payload. It has a max speed of 20 km/h and a payload capacity of 750 kg. Table 2.1 lists additional technical specifications for the Themis UGV and an image of *Tor* can be seen in Figure 2.1.



Figure 2.1 Our Milrem Themis, Tor, without sensors or payload.

Milrem initially developed the Themis as a robotic infantry support vehicle. The main purpose was for a squad to bring the vehicle with them on a mission and use it to carry equipment, supplies, and wounded soldiers. Sensors and remote weapon systems could also be mounted to the Themis, and the squad could use it in offensive and defensive actions and thereby reducing the risk to the soldiers. The Themis is operated with a remote control and a tablet with control software, see Figure 2.2.

The Themis operator can either operate the UGV directly by observing it, or use forward-facing cameras to teleoperate the UGV. At the time of purchase, the Themis did not have any autonomous capabilities. However, Milrem designed the Themis from the ground up as a robotic vehicle, which

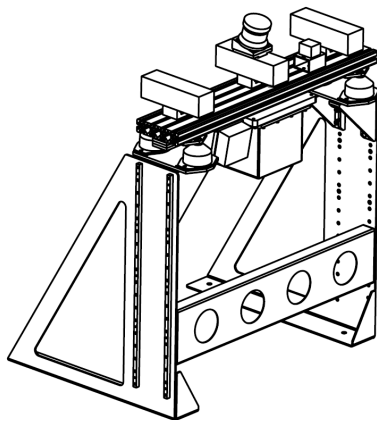
Max Speed	20 km/h
Length x Width x Height	240 cm × 200 cm × 115 cm
Track width	39 cm
Weight	1630 kg
Rated Payload Weight	750 kg
Maximum payload weight	1200 kg
Maximum grade	60 %
Maximum side slope	30 %
Ground clearance	40 cm to 60 cm
Pull force N	15 000 N
Run time hybrid	up to 15 h
Run time electric	up to 1.5 h
Battery type	Lead acid
Power generator	Diesel engine

Table 2.1 Technical specifications for the Milrem Themis 4.5 UGV[41].

means that it features full electronic control over all its systems. It has a software interface for controlling it, which makes it easy to create external software, such as an autonomy stack, that can control the UGV.



Figure 2.2 The hand controller and tablet for controlling the Themis. The tablet can show video from forward-facing cameras so the driver can teleoperate the UGV.



(a) CAD drawing of the Tor's sensor rig with sensor platform and sensors mounted.



(b) Tor, fitted with sensor rig and sensors.

Figure 2.3 The custom sensor rig for our UGV Tor.

2.2 Hardware additions for autonomy system

Our goal with our Themis *Tor* was to develop an autonomy system that would make it capable of navigating in terrain by itself. As Tor did not have any sensors or software for autonomous navigation when we purchased it, we added a sensor package with all the necessary sensors, well as computers for running the autonomy software.

To mount the required sensors, we designed a sensor rig that could be mounted at the front of the payload platform on Tor. The sensor rig featured a rigid extruded aluminum profile that we could mount sensors directly onto. By mounting the sensors on a rigid frame, we could ensure that the sensors would not move relative to each other, which is important for the sensor fusion algorithms. We mounted an aluminum profile to the sensor rig itself using vibration dampers to reduce vibrations from the vehicle. Figure 2.3 shows a CAD drawing of the sensor rig itself, as well as Tor with the sensor rig mounted. These modifications were documented in [2, 30, 26], and the reader is referred to these reports for more in-depth information about the modifications.

The main sensors that we mounted on Tor were:

Lidar A 3D lidar (Light Detection And Ranging) used by the perception system.

Cameras Two monochrome cameras and one color camera used by the perception system.

IMU An inertial measurement unit used for measuring the accelerations and orientation of the vehicle.

GNSS receiver A receiver used for global localization.

Ground speed radar A radar used for measuring the speed of the vehicle and aiding the localization system.

For a detailed list of the sensors and computers that we added to Tor, we refer to Table 2.2.

To run the autonomy software stack, we added two computers to Tor, one rugged main computer that would run most of the control system, as well as a high-performance semi-rugged computer

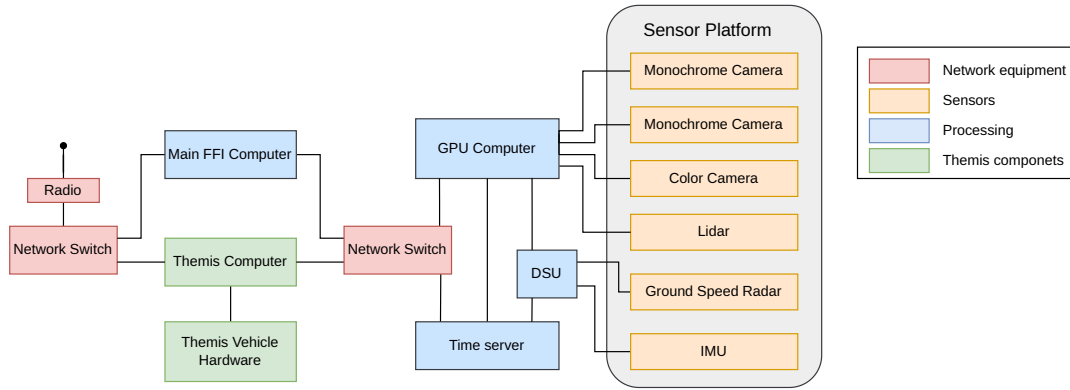


Figure 2.4 Illustration of the hardware setup on Tor.

with a GPU that would run the perception system. High precision time synchronization between all sensors is critical for achieving satisfactory sensor fusion for the perception and localization systems. We therefore added a dedicated time synchronization server to Tor that serves as the master clock for all the sensors. The time-server receives time from Global Navigation Satellite System (GNSS) when it is available, and otherwise uses its own high-accuracy internal clock, and reliably serves one of these to the rest of the system. The time-server has three means of providing time to the sensors:

- NTP¹
- PTP²
- NMEA-0183 together with PPS³⁴

The lidar and the cameras use PTP to set their internal clocks, which are then used to time stamp their sensor data. These sensors are not connected directly to the time-server. Instead, they are connected to the processing computer which relays the time synchronization data from the time-server to the sensors. The radar and the Inertial Measurement Unit (IMU) do not have a network interface for time synchronization. Instead, they are connected to a Data Synchronization Unit (DSU) made by FFI that receives the raw data, timestamps it, and transmits the timestamped data to the processing computer. The DSU is synchronized to the time-server using NMEA-0183 and PPS. An illustration of the network structure on Tor can be seen in Figure 2.4. Additional technical information on the sensors and computers on Tor can be found in Table 2.2.

2.3 Controlling the Themis through software

The Themis has an interface for controlling its systems such as generator state, lights, fans and power state. It also has a number of parameters that can be read such as track speed, battery charge,

¹Network Time Protocol (NTP): A network protocol for synchronizing computer systems to within a few milliseconds of each other.

²Precision Time Protocol (PTP): A network protocol capable of sub-microsecond time synchronization between devices.

³NMEA-0183: A protocol for GNSS data which contains current time information.

⁴Pulse Per Second (PPS): a signal that is sent precisely once per second. Together with NMEA-0183 it can be used to synchronize time.

Make	Model	Description
FLIR	BFS-PGE-31S4C-C	Blackfly S color camera, 3.1 MP, Gigabit Ethernet interface
FLIR	BFS-PGE-23S3M-C	Two Blackfly S monochrome cameras, 2.3 MP, Gigabit Ethernet interface
Ouster	OS-1-64	Lidar, 64 beams, 850 nm, $\pm 16.6^\circ$ vertical field of view, 360° horizontal field of view
RADARXENSE	RXS-SOG-10	Ground speed radar with an $11^\circ \times 11^\circ$ beam.
Honeywell	HG9900	IMU used for inertial navigation
uBlox	NEO M8T	GNSS receiver
Onlogic	K700-X2	Perception processing computer. Rugged computer with Nvidia RTX- 2060 GPU, Intel i7 CPU and PoE ports that powers the cameras.
SDK Embedded Systems	SDK-A77RMG	Rugged IP67 certified computer, which is used to control Tor.
FFI	DSU	Data Synchronization Unit. The DSU is using custom FPGA to timestamp raw data from the IMU and the ground speed radar.
Oriola	VersaSync	Time synchronization server. Rugged master clock that can use a GNSS as a time source. It provides time synchronization through Ethernet using NTP or PTP protocols or a fixed frequency like 1PPS, or by serial link using NMEA-0183.

Table 2.2 Detailed information on the sensors and hardware added to Tor.

fuel level, and engine temperature. The interface is based on Robot Operating System (ROS) – a robotics software communication framework widely used for robots and autonomous vehicles. It is implemented as a ROS2 node that runs on the internal Themis computer that we can communicate with. We can therefore publish to and read from regular ROS2 topics to control and read information from the UGV.

The UGV can be put in different speed control modes:

Normal mode, slow and fast The vehicle is controlled by setting efforts on forwards/backwards movement and turning. In the slow normal mode, the effort is reduced, and the maximum speed is therefore limited to 10 km/h. In fast mode, the full effort is available, and the vehicle can reach its maximum speed of 20 km/h.

Track mode The vehicle is controlled by setting the individual track efforts, without any restriction. Milrem refers to this drive mode as *tank mode*, but a tank is not driven this way, and *track mode* is a more accurate description.

Velocity mode The vehicle is controlled by setting linear and rotational velocities instead of effort. This mode uses an internal controller to convert the velocity commands to track efforts.

An operator controlling the Themis with the original Milrem remote control will typically use the normal mode, as this has the most natural feel. When a computer drives the Themis, it is more important with precise and predictable control, and using either track mode or velocity mode is therefore preferred.

For the velocity control mode, the vehicle drive command $\mathbf{u}_{normal}$ consists of two inputs to the vehicle: a linear velocity along the forward axis of the vehicle, v , and a rotational velocity around the vertical axis of the vehicle, ω :

$$\mathbf{u}_{normal} = \begin{bmatrix} v \\ \omega \end{bmatrix} \quad (2.1)$$

From these two velocities, the speeds for the left and right track, v_l and v_r , are given using the following equations:

$$v_l = v - \frac{L}{2}\omega \quad (2.2)$$

$$v_r = v + \frac{L}{2}\omega \quad (2.3)$$

where $L = 1.58$ m is the distance between the center of the tracks of the vehicle. An illustration of the velocities can be seen in Figure 2.5. When using the track mode, we command the individual track efforts directly, and the control input $\mathbf{u}_{track}$ is given by:

$$\mathbf{u}_{track} = \begin{bmatrix} u_l \\ u_r \end{bmatrix} \quad (2.4)$$

where u_l and u_r are dimensionless efforts on a scale from -1 to 1 for the left and right track respectively. If we wanted to control individual track speeds instead of efforts, we could, for example, implement a simple proportional controller that converts the desired track speeds v_l and v_r to track efforts u_l and u_r . However, as this need is mostly covered by the velocity mode of the Themis, we have not implemented such a controller ourselves.

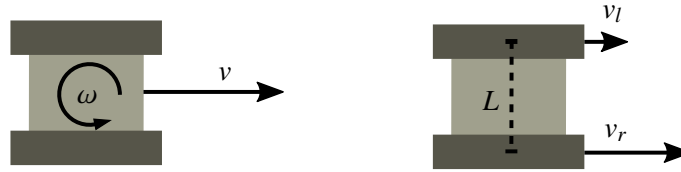


Figure 2.5 On the left: A Themis with linear speed v and a rotational speed ω . On the right: The equivalent track speeds v_l and v_r for the left and right track. The exact relationship between the velocities is given in Equation 2.2 and Equation 2.3.

To provide a standardized way of interacting with the UGV for our software, we have created an abstraction to the low-level interface, called *ThemisController*. The interface accepts a standard ROS velocity command with linear and rotational velocities for controlling the UGV, and translates it to the Themis specific messages. This means that an autonomy stack does not need to know specifics of how the Themis works internally, but instead can use a generic command that can be used to control any vehicle. *ThemisController* handles the internal logic related to controlling the vehicle interface, such as turning on the high voltage system and generator when it receives a command to power on the UGV. If some of the underlying hardware subsystems fail or go into an error state, *ThemisController* also attempts to gracefully handle them and restore them to a normal operating state.

Themis controller will accept control commands from different sources:

Xbox controller For teleoperation when connected to the system with a computer.

Tango2 controller A wireless long-range controller using the TBU Cross-fire link. It can be used for controlling the UGV directly, without intermediate computers like the Xbox controller needs.

Interoperability Profile (IOP) A standard for communication between Unmanned Ground Vehicle (UGV) and operator control units. Allows us to use operator control stations that supports IOP for controlling Tor. For more information, see[23, 42]. A slightly modified version of IOP is becoming STANREC 4818.

Autonomy stack An autonomy stack can also drive the UGV by sending the same types of velocity commands as the other control sources.

All the control sources function the same way, they send velocity commands that *ThemisController* interprets and sends to the low-level controller using the vehicle interface. *ThemisController* will only accept commands from one control source at the time, and the operator will choose which one that is. *ThemisController* can also be put in an *IDLE* mode where it does not accept any incoming control commands. If anybody presses one of the emergency stops, *ThemisController* will stop accepting vehicle commands until an operator can perform a manual reset. For a more thorough description of the subjects covered in this section, we refer the reader to [2, 26].

2.3.1 2D simulator

Performing simulations of the UGV has many benefits such as enabling faster development and testing of the autonomy stack, allowing initial testing of new features to find and fix software bugs,

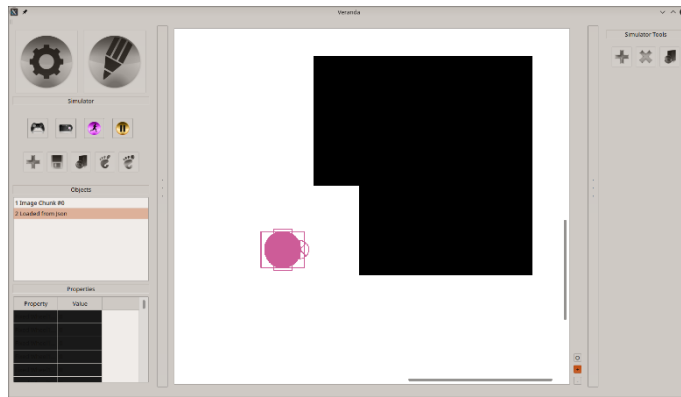


Figure 2.6 Screenshot from the Veranda user interface from a simulation of Tor.

and enabling testing of the autonomy stack in a controlled environment without risk to humans, equipment, or the environment. To enable us to do basic simulations of our autonomy stack on Tor, we have created a 2D simulator which emulates the internal Themis states and has the same interface as the physical Themis UGV. The simulator features a simple physics model that behaves roughly the same way as the real Themis. The Software-In-the-Loop (SIL) simulation environment consists of a ROS2 node called *ThemisSimulator* and the open source ROS2 2D physics simulator called *Veranda*[19]. The Veranda developers are no longer maintaining the project, but at the time it was one of the first simulation environments that supported ROS2, and it has served its purpose since.

Using a simple 2D simulator cannot provide the same level of realism as a full 3D simulator, but it is much easier to set up and run. In particular, it can provide benefit in the early stages of development, for example, by allowing us to check if all nodes are communicating correctly. As the simulation environment emulates the whole vehicle interface, it also means that *ThemisController* cannot see the difference between a simulation and the physical Tor in the real world. The Veranda simulator can generate simulated GNSS and IMU, which means that we can simulate the UGV at coordinates corresponding to the real world. Another useful feature is that it can generate simulated obstacle maps, similar to the ones typically produced by perception modules of autonomy stacks. Together, this enables us to simulate many of the required autonomy stack subsystems. Having a way of testing logic and internal communication of many of the autonomy stack subsystems is highly valuable. The graphical user interface for the Veranda simulator can be seen in Figure 2.6.

3 Autonomy stack

This chapter presents the autonomy stack that we have developed. We have deployed it to our UGV Tor, but it could be adapted to other vehicles as well. Section 3.1 contains an overview of the autonomy stack, and brief explanations of what each component does. Section 3.2 – 3.6 goes into more detail on each of the autonomy components.

3.1 Autonomy stack overview

The autonomy stack is the software that makes the UGV drive autonomously. The stack consists of individual components with different responsibilities, that work together to achieve self-driving capabilities. We have developed our own autonomy stack and tailored it to our needs described in Section 1.2. The stack is implemented using ROS1 as the communication framework. ROS allows us to create individual components or *nodes* that can communicate with each other through network traffic. Each node has a specific responsibility and well-defined interfaces for both input and output. Dividing the autonomy stack into individual components achieves the important property of modularity, where one node can be replaced with another one, as long as the interfaces are kept the same.

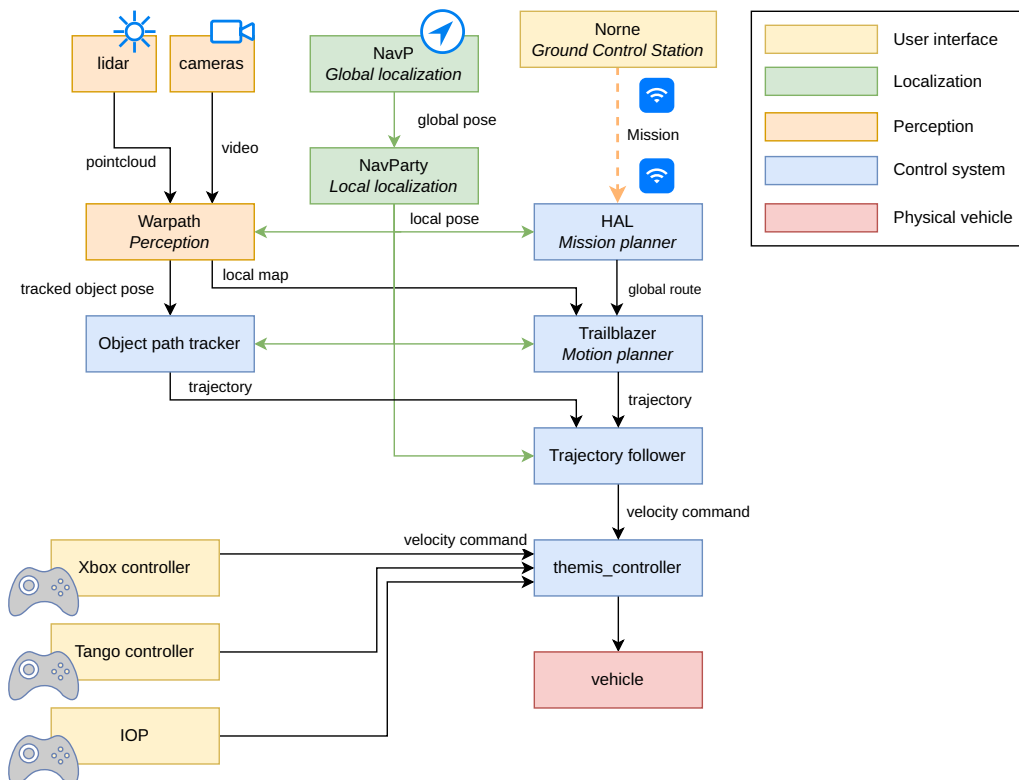


Figure 3.1 Autonomy stack overview

Figure 3.1 shows an overview of the autonomy stack. The individual components are:

Norne The ground control station (GCS) where an operator can send missions to the UGV.

HAL The mission manager which handles the missions and breaks them into smaller tasks.

NavP The inertial navigation system – where the UGV is on the globe.

NavParty The local localization node – how the UGV moves in local coordinates.

Warpath The perception module which processes data from cameras, and the lidar, and determines where the UGV can drive or tracks where specific objects are.

Trailblazer The motion planner which plans where to drive.

Trajectory tracker The trajectory tracker makes the UGV follow the planned trajectories by creating velocity commands.

Object path tracker Tracks an object over time to create a trajectory.

Themis_controller The low-level vehicle interface node described in Section 2.3.

For the UGV operator, the main way of interacting with the Tor is through *Norne* – the ground control station. *Norne* is a map-based control application and runs on a computer connected to the UGV through a radio link or a network connection. When connected to the UGV, it displays the UGV in the map and the operator can send it *missions*. A typical mission will consist of a mission type, for example *move*, and a route that defines where the UGV will drive. The operator can create the route by clicking in the map, or they can load an existing route from a file. They can then send the mission to the UGV where *HAL* – the *mission manager* handles it. The communication link between *Norne* and *HAL* is the only link in the stack that does not use ROS. That is because ROS communication is unreliable over radio. *HAL* is the mission manager, it keeps track of which mission the UGV is running, and breaks it into smaller tasks and passes them on in the control stack. For a typical *move* mission with a route, *HAL* sends the route to *Trailblazer* – the *motion planner* module that decides exactly where to drive. More details on *Norne* and *HAL* can be found in Section 3.2.

NavP and *NavParty* are the modules responsible for everything related to localization – where the UGV is, where it is pointing (together known as a pose) and what the velocities are (in ROS known as a twist). *NavP* is a GNSS-aided INS that estimates the pose and the twist of the UGV, using an IMU, a GNSS receiver, and a velocity radar (which measures true speed over ground). *NavP* concerns itself with the optimal global pose and twist of the UGV, meaning its best estimate of where the UGV is on the earth. *NavParty* uses the estimates from *NavP* to keep track of how the UGV moves in local coordinate systems over time. *NavParty* also maintains a small history of previous local poses. For instance, this provides us the option to look up the movement from where the UGV was 1.0 seconds ago to where it was 0.5 seconds ago. The advantage of using local coordinate systems is that they can be easier to work with when you have sensors that measure relative movement, such as wheel odometry or a velocity radar. You can also avoid problems where using a GNSS system causes jumps in the global pose estimate, particularly when the GNSS reception is poor. More on localization in Section 3.3.

The *Perception* system is the eyes of the autonomy stack. It consists of a lidar, cameras, and *Warpath*, the software for processing the lidar and image data to create situational awareness. The main task of *Warpath* is to create a local map that tells where the UGV can and cannot drive, an obstacle map. The obstacle map is created using only what *Warpath* can perceive with its sensors, and it uses the local coordinate systems from *NavParty* to reference all positions. The sensors have a limited range, and the map will therefore not be larger than approximately 100 x 100 m and centered at the UGV.

Warpath receives continuous sensor measurements, and updates the map as the UGV moves. It can also produce other types of maps, such as inclination maps that can improve the maneuvering on terrain. Warpath sends the local map to *Trailblazer*, the motion planner that decides where to drive based on the local map and the route from HAL.

Warpath can also use camera images to track an object tagged with an AprilTag. We can, for example, place an AprilTag on the back of a vehicle, and Warpath can then track the position of the vehicle. The control stack can then use the tracked position of the vehicle to make the UGV follow it. For more information on the perception system, see Section 3.4.

Trailblazer is the module that plans exactly where to drive by creating a trajectory that the UGV will follow. It uses three data sources for the planning:

1. The route from HAL that defines where its current goal is.
2. The obstacle map from Warpath that tells where it is possible to go.
3. Its own local pose, so it knows where it is in the local map.

The route it receives from HAL is given in global coordinates. Trailblazer converts the global route to the local NavParty coordinate system to relate it to the local map coordinates. Then, it uses the local map to plan a local trajectory towards the nearest waypoint that also avoids obstacles. To perform the actual planning, it uses one of several possible motion planning algorithms. As the local map rarely covers the entire commanded route, Trailblazer plans trajectories towards the next waypoint to the edge of the current obstacle map. Therefore, it replans the trajectory every few seconds using updated maps and vehicle pose. Then, Trailblazer sends the planned trajectory to the *Trajectory tracker*. More on Trailblazer can be found in Section 3.5.

An alternative module that creates trajectories is the *Object Path Tracker*. If the UGV is in *Follow* mode, this node receives a track from Warpath containing pose and twist of the tracked object. The Object Path Tracker uses the track to create a trajectory of where the object has moved. It then sends this trajectory to the *Trajectory tracker*, the same way as the motion planner sends a trajectory. However, the UGV will only use one of these two modules at the same time, it will not perform motion planning and object following simultaneously. More on *Follow* functionality can be found in Section 3.4 and Section 3.6.

The *Trajectory Tracker* takes the trajectories from either Trailblazer or the Object Path Tracker and tries to make the UGV follow them. It does so by continuously creating velocity commands for the velocity controller. It creates the commands based on where the UGV is in relation to the trajectory, and it will try to steer the UGV both along and towards the trajectory if it is not on it. The trajectory tracker supports different methods for achieving this, from using simple control laws to more advanced methods such as Model Predictive Control (MPC). More on trajectory tracking in Section 3.6. The velocity commands from the Trajectory Tracker are sent to *Themis_controller*, the low-level vehicle interface node described in Section 2.3, which makes the vehicle execute them.

3.2 Mission control – HAL and Norne

A UGV operator can command the UGV to perform tasks by creating a mission in the Ground Control Station *Norne*, and sending it to the UGV. On the UGV side, the mission manager *HAL* receives the mission, and is responsible for breaking it down into smaller tasks, passing them on in the autonomy stack, and monitoring the progress as the stack executes them.

3.2.1 HAL – The mission manager

HAL is the module that handles the intention of the operator in the form of *missions*. A mission consists of one or more high-level *tasks*, that the UGV can perform. Some examples of possible tasks are:

Move The UGV moves to a single specified location.

FollowRoute The UGV moves along a specified route.

Wait The UGV waits for a given time or until triggered by something.

A mission consists of either one such task or multiple sequential tasks. A task typically has some geographic features associated with it, such as a set of waypoints or a location to move to. The **Move** task requires a single waypoint that the UGV shall move to. The execution of the task is left to the autonomy of the UGV to handle, and requires the UGV to find its way to the target by itself. The **FollowRoute** task is similar, but features a more detailed route with multiple waypoints that the UGV should follow. While these tasks roughly achieve the same goal in getting to a destination, the way they achieve it can be different. For example, following a route can be more useful if there is a road that the UGV can follow, in which case it makes sense to try to follow the road as long as its location is known in advance. However, it requires a means of getting the route, such as a route planner or using routes from previously driven ones.

3.2.1.1 The HAL framework

HAL is a decisional autonomy framework developed at FFI, designed to handle mission execution and task management for autonomous systems. It provides a platform-independent solution for managing high-level tasks and breaking them down into smaller, executable components. Originally developed for FFI's HUGIN, an Autonomous Underwater Vehicle (AUV) [55], HAL has since been adapted for various platforms, including FFI's UAV swarm Valkyrie and Unmanned Surface Vehicles (USV) Frigg and Odin [20].

HAL can model missions and tasks in several ways, such as using hierarchical behavior trees or finite state machines, depending on the desired mission execution. HAL also supports running continuous processes simultaneously, allowing the system to handle ongoing activities such as monitoring sensor data or maintaining situational awareness. This flexibility allows HAL to support a wide range of mission requirements, from simple waypoint navigation to more complex multi-step operations.

HAL's extensibility is another key feature, allowing developers to integrate new capabilities and expand its functionality as the autonomy stack evolves. For example, as the UGV's basic autonomous driving capabilities improve, HAL can be extended to include more advanced decisional autonomy, such as dynamic route planning, obstacle negotiation, or collaborative missions with other vehicles.

Overall, the HAL framework provides a robust and scalable foundation for mission management, ensuring that the UGV can execute tasks efficiently while maintaining the flexibility to adapt to future requirements. For more on HAL in general, see [55, 20].

3.2.1.2 *Interface for ground control stations*

HAL uses an implementation of Battle Management Language (BML) that allows ground control stations (GCS), such as *Norne* to connect and send missions. BML is a message protocol for command and control for simulated and robotic military forces [44], which ensures a standardized way of interfacing the UGVs.

When a GCS connects to the UGV, HAL sends a **Capability Report**, which lists the tasks the UGV can perform and the parameters required for each task. For example, the **Patrol** task requires a list of waypoints, speed constraints, and whether the waypoints should be patrolled in a circular or back-and-forth manner. This report allows a GCS to dynamically create missions tailored to the UGV's capabilities. An operator can then define a valid mission and send an **Order** to HAL.

During mission execution, HAL sends periodic **Status Reports**. These reports include information such as the UGV's current location, active tasks, progress, planned trajectories, and other vehicle-specific data. This feedback allows the operator to monitor the mission's progress in real time.

3.2.1.3 *HAL capabilities*

The HAL implementation Tor is designed with the primary goal of enabling autonomous driving in terrain. This focus on terrain driving means that the **Move** and **FollowRoute** tasks are central to its current capabilities. However, until the autonomy system is capable of reliably driving in terrain, adding support for more advanced decisional autonomy is not a priority. The **Move** task allows the UGV to navigate to a single specified location autonomously, while the **FollowRoute** task enables it to traverse a predefined route consisting of multiple waypoints.

When executing one of these missions, the waypoints are sent to the *Trailblazer* motion planner, which is responsible for determining the best path for the UGV to follow. These tasks are handled slightly differently, even though we could create a **Move** mission with many consecutive waypoints. For **FollowRoute** we have higher confidence that the route is traversable, and the UGV should try to stay close to the route to maximize its chances of success in traversing the route. For the **Move** task, we assume that each waypoint could be placed arbitrarily, for example, by an operator clicking on a zoomed out map. We therefore have low confidence in that each waypoint is traversable, and we therefore rely more on the motion planner to find traversable paths while accepting larger deviations from the waypoints. An example from a GCS where Tor is executing a FollowRoute task is shown in Figure 3.2. The ground control station shown is FFI GCS, the GCS we primarily used before adopting *Norne*.

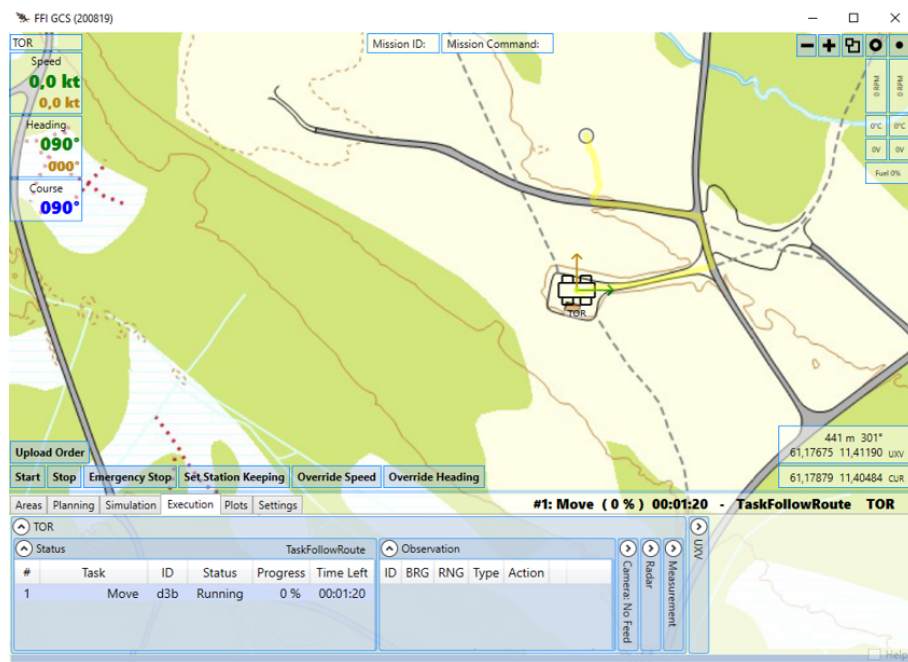


Figure 3.2 The previously used FFI GCS showing Tor during execution of a FollowRoute mission. The FFI GCS was primarily used for the USV Odin, and the keen observer can spot the maritime influences in some details of the GCS.

In addition to these core tasks, we have also implemented the task **Patrol**. This task allows the UGV to continuously loop through a set of waypoints, making it suitable for applications such as perimeter monitoring or repeated observation of specific areas. However, **Patrol** serves more as a way of showcasing something different from the basic **Move** and **FollowRoute** tasks, rather than capabilities required for the main goal of the UGV, namely driving in terrain. Looking ahead, future expansions aim to enhance the UGV's autonomy by incorporating more advanced decisional capabilities. For instance, integrating a route planner would enable the UGV to dynamically find alternative routes when it encounters obstacles or gets stuck, improving its ability to adapt to unforeseen challenges. These planned improvements will build on the existing foundation, ensuring that our autonomy stack continues to evolve and support increasingly complex mission requirements.

3.2.2 Norne – The ground control station

Norne is the ground control station that we use for controlling our the UGV. It is a web-based graphical user interface that we have developed for controlling multiple types of unmanned vehicles. It is based on the CesiumJS 3D geospatial engine, which makes it capable of visualizing map data and vehicle information in both 2D and 3D. Norne has been tested with multiple platforms, including USVs, UGVs, UAVs, the Spot robotic dog, and simulated units in the VBS simulation environment.

One of the primary motivations for developing Norne was to create a platform-independent GCS. We previously used a different control station, the FFI GCS[47], which can be seen in Figure 3.2.

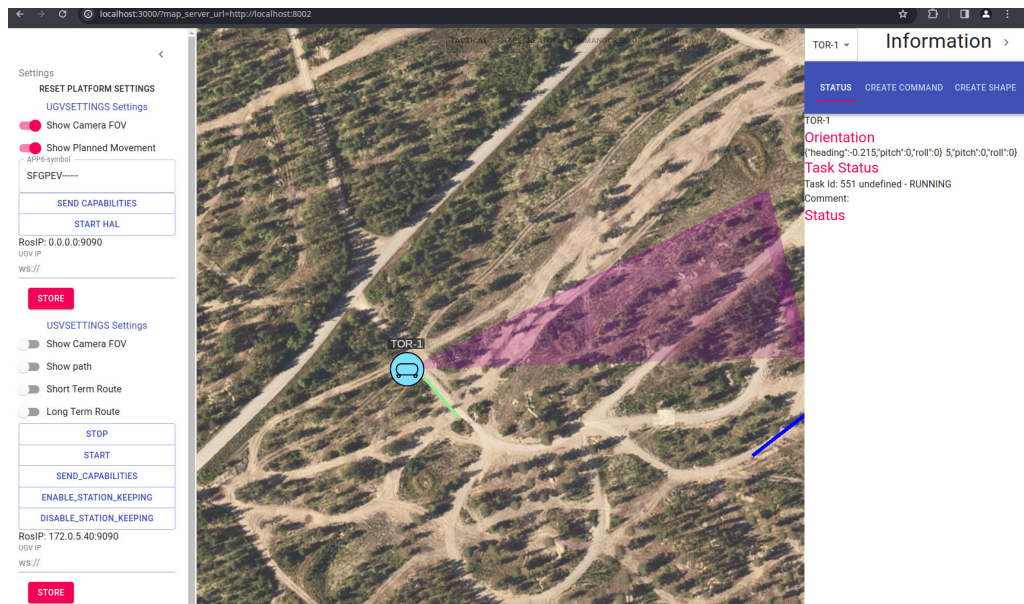


Figure 3.3 Norne during execution of a mission with Tor.

While FFI GCS was a functional and well-designed GCS, it was not as flexible as we needed, particularly for controlling multiple types of vehicles, and it was limited to the Windows operating system. Norne, in contrast, is built as a web application hosted in a Docker container, enabling it to run on any platform with a browser, including mobile devices. Norne's web-based framework also provides greater flexibility for development, enabling rapid testing of new features. While its graphical interface prioritizes functionality over aesthetics, this trade-off has proven valuable for our UGV autonomy development, where adaptability and extensibility are key. In Figure 3.3, we see a screenshot of Norne during the execution of a mission.

Norne uses the BML protocol to communicate with HAL on the UGV. Norne dynamically reads the **Capability Report** sent by the UGV, which lists the tasks the UGV can perform and their associated parameters. Based on this report, Norne can dynamically create missions tailored to the UGV's capabilities. For example, it can define tasks such as **Move**, **FollowRoute**, or **Patrol**, specifying parameters like waypoints and speed constraints.

Norne has both 2D and 3D map views, depending on the operator's preference. For 2D views, it can use generic tile map servers, which can be based on regular maps such as OpenStreetMap or maps from the Norwegian Mapping Authority, or aerial imagery as shown in Figure 3.3. For 3D maps, it needs a terrain model created from terrain height data, and textured with, for example, aerial imagery.

Norne also supports external route integration. Operators can load routes from files or connect to a route planning service to request a route based on the UGV's current position and target destination. This feature is particularly useful for missions requiring pre-planned paths or dynamic route adjustments. It can also save an operator from having to manually define a route, which can be time-consuming and error-prone.

The interface provides real-time visualization of the UGV's status and mission progress. Operators can monitor the UGV's location, active tasks, and planned trajectories, ensuring situational awareness

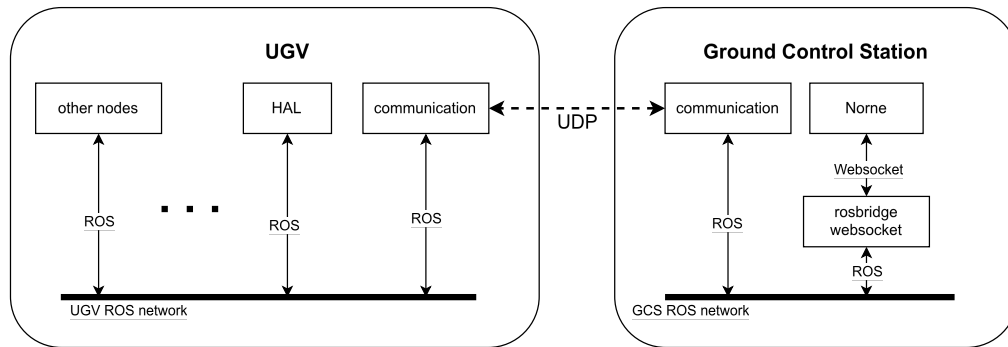


Figure 3.4 Communication between HAL and Norne. Both the UGV and the GCS have their separate ROS networks. Each of them runs a communication node that bridges the two networks using UDP communication. Norne itself communicates using websocket, and therefore has a Rosbridge-WebSocket to communicate with the GCS ROS network.

throughout the mission. Norne’s frontend, built with JavaScript and the Roslib library, interacts with a ROS bridge WebSocket server to translate messages between JSON formats and ROS topics. This setup ensures seamless communication and platform-independent operation.

To facilitate communication, Norne and the UGV use separate ROS networks bridged by a custom UDP-based communication solution with communication nodes on both sides. Unlike ROS’s default TCP-based communication, which is unsuitable for unreliable radio links, the communication solution ensures reliable and responsive data transmission. On both the UGV and the GCS sides, the communication node subscribes to a selection of ROS topics, serializes the data into a byte stream, and transmits it over UDP. Simultaneously, they also receive serialized byte streams over UDP, which they then deserialize and publish on the ROS network. This setup allows seamless communication between the two systems for a selection of the ROS topics.

Norne itself uses a standard Rosbridge-WebSocket server to connect to the ROS network. The server sets up a WebSocket interface, translating messages between JSON formats and ROS topics. Norne’s frontend, built with JavaScript, uses the Roslib library to interact with the WebSocket server, enabling platform-independent operation through a browser-based interface. This robust communication architecture ensures that the UGV and Norne can exchange commands and reports efficiently, even in environments with unreliable and low-bandwidth radio links. An illustration of the communication architecture is shown in Figure 3.4.

While Norne is not a finished product, it serves the need for an efficient control application for the UGV. As the primary goal of the UGV development is to try to achieve off-road driving, the main requirements for the GCS are to be easy to use and easy to develop for, which it does well. Its flexibility and rapid development cycle have enabled the team to test and refine new features quickly, making it a critical tool for advancing the autonomy stack.

3.3 Localization

The localization system in the autonomy stack consists of two systems: NavP that solves the global localization problem and NavParty that solves the local localization problem. NavP is a GNSS-aided INS developed by FFI. NavP is the real time solution of NavLab [12] and was originally developed for the AUV HUGIN. NavParty acts like a time database for global localization data from NavP and can transform coordinates between different coordinate frames (such as sensor frames for cameras and lidars or other frames like vehicle body center). Other processing modules in the stack, like Warpath and Trailblazer, query NavParty for the transformations between different coordinate frames or the same coordinate frame at different timestamps. Two examples:

1. Trailblazer transforms a global route into local coordinates. To do this, Trailblazer needs the transformation between latitude and longitude and a local origin.
2. Warpath transforms a point cloud in the lidar frame into a local plane under the UGV to create an obstacle map and then fuse the local obstacle map into a global obstacle map.

3.3.1 NavP – global localization

NavP estimates the optimal global localization solution by filtering measurements from the IMU and the aiding sensors, i.e. the ground speed radar and the GNSS. The localization solution consists of the global position, orientation, and velocity estimates, and angular velocity and linear acceleration measurements. NavP runs on the dedicated hardware DSU, and it sends the estimates out on the Ethernet network.

NavP does not use Tor's track odometry as an aiding sensor since the measurement frequency is too low on the Themis platform. However, when we measure zero track speed, we assume that the UGV is stationary, and we use that to command NavP to a so-called ZUPT mode ⁵ It is also possible to turn off each aiding sensor separately to emulate jamming or sensor failure.

On the main processing computer, a ROS node receives the localization estimates and redistributes them as standard ROS messages.

3.3.2 NavParty – Local transformation database

NavParty is a modular framework built up by a core module that does all the processing, subscribers for different data types on different middlewares that feeds the core module with data, and publishers that publish different data types on different middlewares. In the stack, NavParty is using a ROS subscriber that receives the global localization data from NavP. All the static transformations, i.e. all the body-fixed transformations, such as transformations between the sensor frames and body origin, are read from a configuration file. These static transformations will never change during operation.

NavParty uses a graph to keep track of all the different frames and their relations. The nodes represent coordinate frames, and the edges represent the transformations between connected frames.

⁵ZUPT can mean Zero Velocity Update or Zero Update Position and Timing. Here we mean Zero Velocity Update by setting linear and angular velocity relative to the inertial frame to zero.

We denote a transformation between two frames A and B as T_{AB} . A transformation T_{AB} is a four-by-four matrix consisting of a rotation matrix and a translation vector.

$$T_{AB} = \begin{bmatrix} R_{AB} & t_B^A \\ 0 & 1 \end{bmatrix} \in SE(3) \quad (3.1)$$

where R_{AB} is a 3×3 rotation matrix and t_{AB} is a 3×1 translation vector. For details on the mathematical background of transformations and details on notation, see [14]. Table 3.1 shows the most important coordinate frames we use on Tor. For more details on the different frames and their definitions, we refer to [11, 28].

Symbol	Description
E	Name: Earth-centered, Earth-fixed (ECEF) Position: Located in the center of the earth. Comments: Earth-fixed, i.e. rotates and moves with the Earth, see Table 2 in [11]. The ROS equivalent of the E frame is the <i>earth</i> frame [28].
N	Name: North-East-Down (NED) - local level Position: Center of the IMU, see Table 2 in [11]. Orientation: The x -axis points towards north, the y -axis points towards east (both are horizontal), and the z -axis is pointing down. Comments: When moving relative to the Earth, the frame rotates about its z -axis to allow the x -axis to always point towards the north.
B	Name: Body Position: Center of the IMU, see Table 2 in [11]. Orientation: The x -axis points forward, the y -axis to the right and the z -axis in the vehicle's down direction. Comments: The frame is fixed to the vehicle. The ROS equivalent of the B frame is the <i>base link</i> frame [28].
C	Name: Color Camera Position: Optical center of the color camera Orientation: Same orientation as the B . Comments: Fixed to the body of the vehicle, as the Body frame.
Q	Name: Local level, integration origin Comments: This frame is equal to the first NED frame registered in <i>NavParty</i> and acts as the integration origin. The ROS equivalent of the Q frame is the <i>odom</i> frame [28].
E_p	Name: Local level, local origin Comments: Somewhere on the earth's surface. Usually defined by latitude, longitude and height, or as the first NED frame registered in <i>NavParty</i> . The ROS equivalent of the E_p frame is the <i>map</i> frame [28].

Table 3.1 The main coordinate frame definitions used for localization on the UGV. The list does not contain all the coordinate frames for the sensors.

The edges of the frame graph can either be static or dynamic transformations. A static transformation is valid at all times, and if a new transformation is inserted, the old one is replaced. The dynamic

transformations consist of a time buffer with poses. To calculate the transformation between two frames at a given timestamp, NavParty first finds the shortest transformation chain between the two frames in the graph. Then, it computes the composed transformation by multiplying all the different transformations in the chain.

It uses the static transformations as-is independent of timestamp, whereas for dynamic transformations, it looks up the transformations in the respective time buffer. If a buffer contains a transformation with the exact same timestamp, that particular transformation is used. Otherwise, it interpolates the transformation using the existing transformations directly before and directly after in the buffer. The interpolation itself is calculated assuming constant linear and angular velocities between the two timestamps. This functionality is similar to the TF [9] in ROS, but a major difference is that NavParty also propagates uncertainties and computes velocities and accelerations when combining poses. Another difference is that NavParty also handles multiple paths through the graph by selecting the shortest path, unlike TF which assumes a tree structure.

To illustrate, we can show an example of a transformation with uncertainty, $T_{EC}(t)$, between the world frame E and camera frame C fixed on Tor:

$$T_{EC}(t) = T_{EN}(t)T_{NB}(t)T_{BC} \quad (3.2)$$

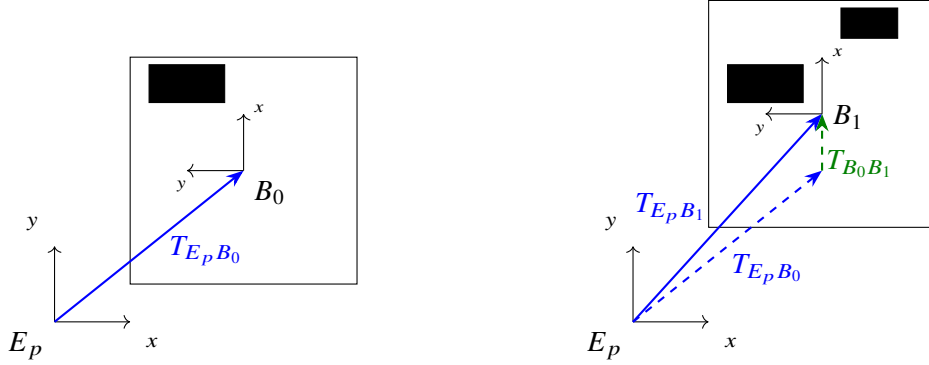
Here we have the transformations between the different frames:

- $T_{EN}(t)$ is the transformation to a local cartesian frame N centered at the INS system of Tor, and is estimated by NavP as latitude, longitude and height. It is a dynamic transformation.
- $T_{NB}(t)$ is the transformation between the local frame and the body oriented frame B , with the same origin. It is estimated by NavP as 3-2-1 Euler Angles, and is a dynamic transformation.
- T_{BC} is a static transformation between the body frame and the camera frame, and is measured during system calibration.

NavParty will do a lookup in each buffer to find an exact hit on timestamp or interpolate between the nearest poses. For more on the mathematical background on how NavParty computes and propagates uncertainties, velocities and accelerations, we refer to [14].

3.3.3 Absolute and relative localization

Consider the following task: We want to fuse sensor measurements over time into a local map. When the UGV moves, the body frame B that all the sensors measurements are relative to, also moves. To fuse these, we either have to transform the previous map into the current body frame, or transform all measurements into a fixed local frame E_p . For both approaches, we need to know how the body frame B moves in relation to the local frame E_p over time. This is illustrated in Figure 3.5, where we first see a UGV at time t_0 in Figure 3.5a with a local obstacle map centered at the body frame B_0 . At the next time step, the local obstacle map is translated together with the UGV based on the movement $T_{B_0B_1}$, and can combine existing features with new sensor measurements to get an updated obstacle map.



(a) A local obstacle map centered on the UGV body frame B at time t_0 . The transformation from E_p to B_0 is $T_{E_p B_0}$. (b) The UGV and the obstacle map have moved from frame B_0 to B_1 . The transformation from E_p to B_1 is $T_{E_p B_1}$. The movement is $T_{B_0 B_1}$.

Figure 3.5 Fusion of obstacle maps over time requires knowledge of how the UGV moves. The local frame E_p is fixed, whereas the body frame B moves in relation to E_p over time.

With the transformations from frame E_p to the frames B_0 and B_1 given by $T_{E_p B_0}$ and $T_{E_p B_1}$, we can compute the movement of the body frame between the two timestamps as:

$$T_{B_0 B_1} = T_{B E_p}(t_0) T_{E_p B}(t_1) \quad (3.3)$$

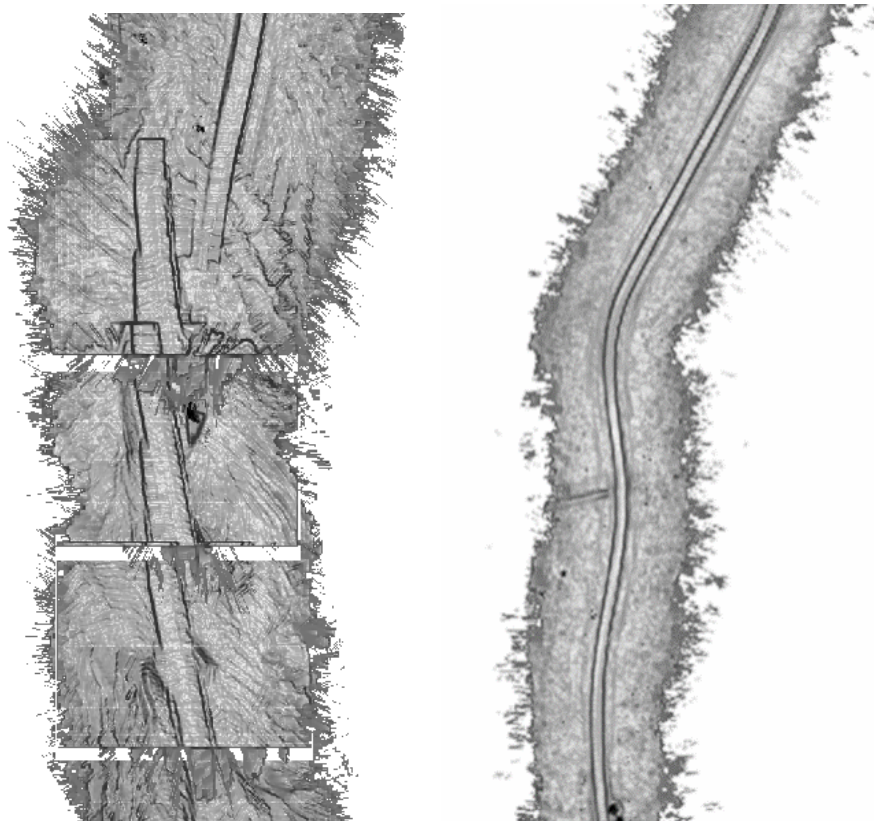
The issue is that we do not have perfect knowledge of the movement of the UGV. We can take two approaches to solve this:

Absolute Use the best global position estimates from NavP at each time step to compute the difference in position. This measurement will indirectly be based on GNSS and what NavP reports as the global position. With poor GNSS reception, the position estimates can be discontinuous, leading to jumps in the computed movement.

Relative Integrate velocities from NavP between time steps to compute the movement. This approach will give a smooth and continuous trajectory, but the trajectory will slowly drift away from the true global position over time, especially when GNSS is not available. However, the local consistency of the fused map will be much better than when using global positions directly. This approach is similar to using odometry such as wheel odometry, visual odometry, and lidar odometry to compute relative movements.

With our NavP and NavParty setup, we can compute both absolute and relative movements. Calculating absolute movement is straightforward because NavP already provides global positions and orientations at each time step.

For relative movement, we implemented an approach where NavParty integrates the linear and angular velocities reported by NavP over time to obtain relative transformations. These transformations are expressed in a new local coordinate frame, denoted as Q , which initially shares the same origin and orientation as the local localization frame E_p . However, as the integration progresses, the Q frame may drift away from the true global position over time due to small errors in the measured velocities,



(a) Fused map using absolute movement. (b) Fused map using relative movement.

Figure 3.6 Fused cost maps using absolute and relative movement. The cost maps are from a real-life experiment where the INS suffered from GNSS errors. The discontinuities in Figure 3.6a are due to jumps in the global position estimate.

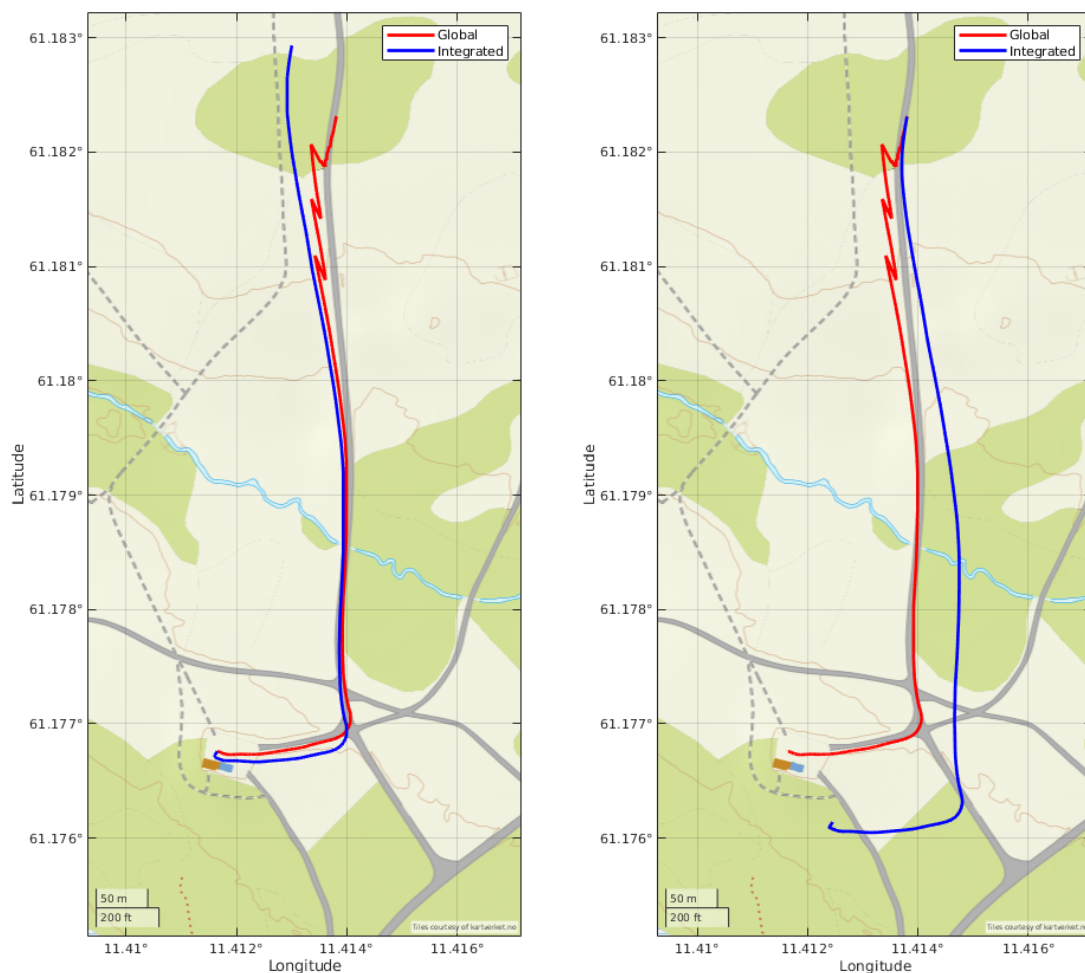
while E_p remains fixed globally. This provides us flexibility: for fusing local maps over time, we can use either E_p or Q . The key difference is that E_p maintains global consistency, whereas Q provides smooth local continuity at the cost of global drift.

In Figure 3.6, we see an example from a real-life experiment where the GNSS dropped out at the start of the data collection and then returned later. We see two fused cost maps where one is generated using absolute movement from NavP, and the other is generated using relative movements from the integrated velocities. In Figure 3.6a, we see how using the absolute movements caused discontinuities when the GNSS reception returned and from NavP, and we can see that the map is inconsistent due to jumps in the global position estimates. In Figure 3.6b, the relative movement cost map does not suffer from the same jumps and the resulting map is consistent and smooth.

The driven paths from this experiment calculated using both absolute and relative measurements are shown in Figure 3.7. In Figure 3.7a, we see the two paths using relative movements with the first timestamp as origin. The true driven path was along the road in the map, and we can see the relative/integrated path drifted away from the road over time. The absolute/global path initially followed the relative path, but when it regained GNSS reception it jumped towards the correct position on the road, and eventually converged to the true position. The relative path is smooth but continues to drift away from the true path over time.

In Figure 3.7a, the first timestamp is used as origin, i.e. the integrated relative positions drift away from the relative global positions. In Figure 3.7b, we have shifted the global position of the Q frame so that the paths are aligned at the newest timestamp, and we can see the integrated path drift further away as we go back in time.

Shifting the origin of the Q frame to align with the best available global position can be useful for obtaining improved coordinates in the current local frame and maintaining consistency with external references. This adjustment can be performed periodically, for example, at fixed intervals or when the detected drift has reached an error threshold. As long as we know how large the shift is, we can account for the jump in the existing fused maps and planned and driven paths, or we could discard the current system states entirely and do a soft reset of the local map before continuing operation. However, because we always know the transformation between Q and E_p , we can compensate for the accumulated drift when converting global coordinates into the local frame. This is particularly valuable when integrating external data such as global road coordinates into a locally drifting frame, ensuring that navigation remains aligned with the intended route.



(a) Absolute path and relative path with the first pose to the south as origin. (b) Absolute path and relative path with the last pose to the north as origin.

Figure 3.7 Absolute/global path and relative paths with different origins. Because we regained GNSS reception, the absolute path eventually jumps to the correct position on the road. These jumps are what caused the badly fused map in Figure 3.6a. Because the relative path is never at the true global position throughout, we can shift at which position the relative path is aligned with the true global position. Here, we see aligning the paths at both the first and the last position.

3.4 Warpath – Sensor fusion for situational awareness

The purpose of the perception system is to create *Situational Awareness* – to determine where it is possible to drive, and determine the position of objects of interest. As mentioned in Chapter 2, Tor is outfitted with a sensor platform with proprioceptive sensors (sensors measuring internal states) for localization and odometry measurements, and exteroceptive sensors for measuring the surroundings. By processing localization data coupled with exteroceptive measurements, Tor can generate an understanding of the surrounding terrain’s geometry and the motion of objects of interest in the scene. This information enables Tor to react to changes and plan its motion.

This section describes *Warpath* – our perception software and its sensor processing methods used in the autonomy stack. The sensor processing methods solve two general problems: estimating the pose of other objects in the scene, and the estimation of the terrain and obstacle maps. We begin by outlining the overall structure of the sensor processing framework and we introduce our method for estimating terrain structure and obstacles from lidar measurements. Next, we describe usage of the AprilTag method for image-based pose estimation. We end the section with a short overview of the visualization methods used and developed to ease and enhance experiments and demonstrations.

3.4.1 Overall structure of Warpath

The general structure of the sensor processing module in the autonomy stack is a pipeline. The pipeline receives data from the sensors and localization systems, processes them, and sends them to other modules. FFI uses this processing structure for several of our autonomous platforms, and we typically call the instances *warpath* with the platform name suffixed. Thus, the warpath instance on Tor is called *warpath_tor*. Common for each Warpath instance is that they use the *superflow* framework [48] described in the following subsection.

3.4.1.1 Superflow and proxels

Superflow is a framework for connecting smaller *processing elements* into a full process pipeline. Using the same naming scheme as pixel and voxel (picture and volume element), we call a processing element a *proxel*. A processing pipeline in superflow is a collection of connected proxels, where each proxel acts as a node in a *graph*. Superflow instantiates and configures its graph at startup by parsing a configuration file. This makes the processing pipeline flexible, as we can set up different configurations of the process pipeline with minimal effort, and without recompiling the *warpath_tor* executable. The main configuration of *warpath_tor* features two processing pipelines, one pipeline for terrain and obstacle estimation from lidar data, and a pipeline for doing image-based pose estimation of a target object. The two pipelines are described in Section 3.4.2 and Section 3.4.4.

3.4.2 Terrain mapping and classification from lidar measurements

For the UGV to determine where it can drive, it needs information about the structure and geometry of its surroundings. From this information, it can classify surrounding areas as drivable or obstructed. This section covers how warpath processes lidar measurements to generate 2.5D *height maps* and *obstacle maps* of the UGV’s surroundings. The height map pipeline is described in three steps:

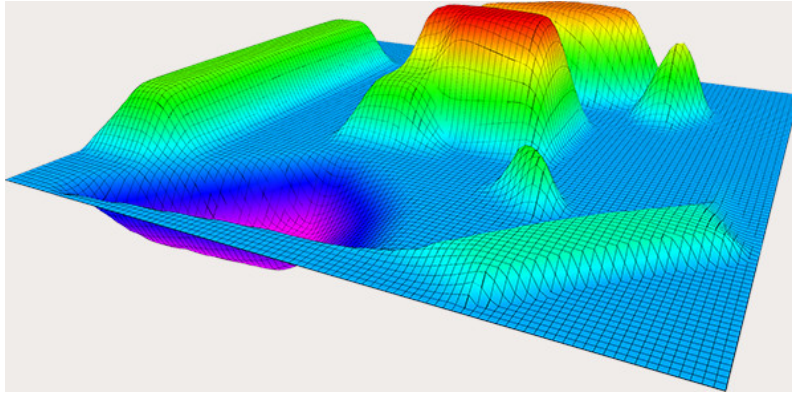


Figure 3.8 Render of a Grid Map[8] containing a 2.5D height map. Image from https://github.com/ANYbotics/grid_map.

- The height map data format.
- A review of our method for estimating height and obstacles from a single lidar swipe.
- An outline of how we fuse the height and obstacle maps over time.

3.4.2.1 The height map data format

A height map is a 2D discretization of 3D terrain geometry. We divide the 3D area into a horizontal *grid*, where each *grid cell* holds a floating-point value denoting the terrain height of that cell. In this way, 3D information is stored in a 2D data structure, which is why such grids are commonly called 2.5D maps. We can use the height map information to render full 3D images of the terrain topography, such as seen in Figure 3.8.

In principle, a height map is just a $n \times m$ float array. However, because of memory constraints, there are limits to how large these arrays can be before we replace old data with new measurements. Some logic is required to do this efficiently, which is why we use an existing tool, *grid_map*⁶[8], to handle this.

In this sensor processing pipeline, warpath calculates three maps from lidar measurements: a terrain height map, an inclination map, and an obstacle map. The methods for creating these maps will be described in the following sections.

3.4.2.2 Estimating terrain height from lidar measurements

A lidar measures the surroundings by emitting laser beams in known directions around itself and measuring the distances. We model the lidar as a group of emitters stacked in a column and rotated around the lidar's z-axis, as illustrated in Figure 3.9. The collection of all the 3D points from a single lidar revolution is called a *point cloud*.

⁶https://github.com/ANYbotics/grid_map

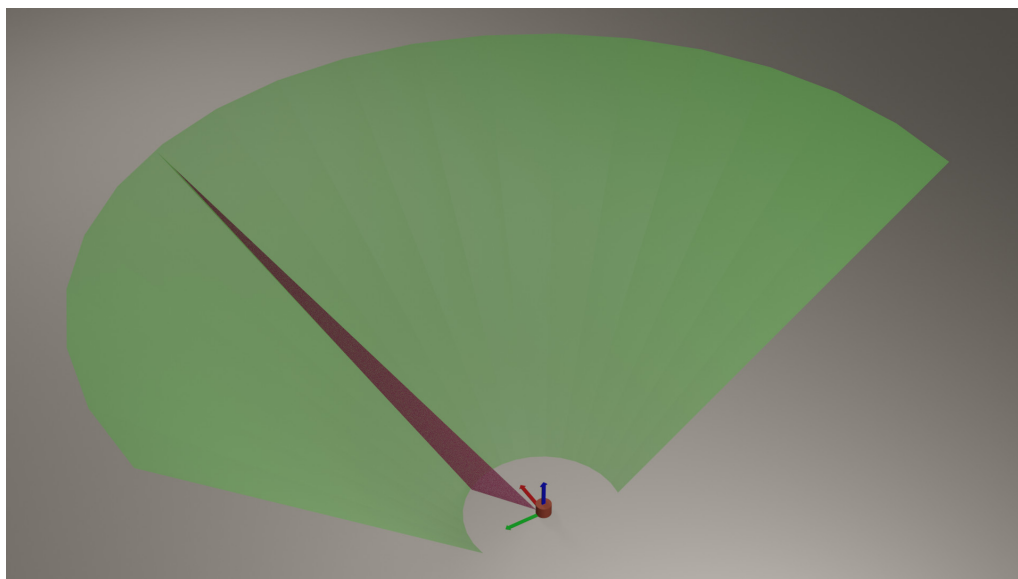


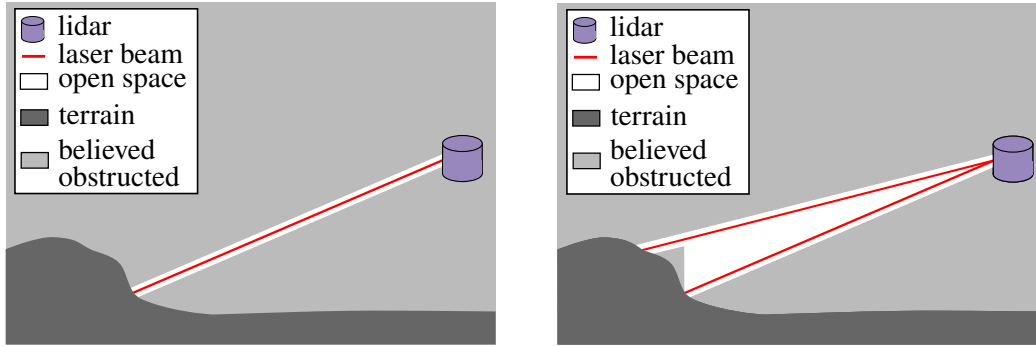
Figure 3.9 A lidar interacting with its surroundings. The lidar is illustrated by the red cylinder positioned in the coordinate system origin. The green half-cone is the terrain surrounding the lidar, while the red plane represents a single column of lidar lasers. A lidar swipe is a full revolution of the column around the lidar.

If a laser beam hits a surface, the light returns and the lidar makes a *range* measurement. Because the direction and range of the beam is known, we can calculate the 3D point of where the laser beam hit a surface. We can infer two interesting observations from such a point measurement:

- Because the light was reflected, the point measurement marks the position of something obstructive, e.g. ground, vegetation, objects, etc.
- The line between the emitter and the measurement is unobstructed.

Considering these observations, a *best guess* estimation of the surroundings would be that the UGV is embedded in a solid medium with the only free space defined by the measurement rays. We can reason to obtain an estimate of the surroundings: We assume the UGV is placed in a chunk of solid mass, and the only open space in this mass is the line between the lidar and the measurement points. Each measurement line has neighboring measurements, both in the same column and in the same row. We connect these open measurements to form an open space and a boundary surface mesh. We then make the guess that the ground is the space directly below our lowest measured open space. By doing a full lidar swipe, each laser measurement “carve” out observations of open and closed space around the UGV, as illustrated in Figure 3.10. Our method estimates and classifies the surroundings through three functional steps:

1. Generate a height grid of the terrain from a single lidar swipe. Details in Section 3.4.2.3
2. Use the terrain mesh together with the point cloud to classify each map cell as obstructed or not. Details in Section 3.4.2.4
3. Fuse the new map measurement with past measurements. Details in Section 3.4.2.5



(a) A single laser beam measurement from a lidar (b) The next laser above in the column maps open space that is mapping open space in the scene. space directly above previous measurement.

Figure 3.10 A side view illustration of open space mapping with lasers.

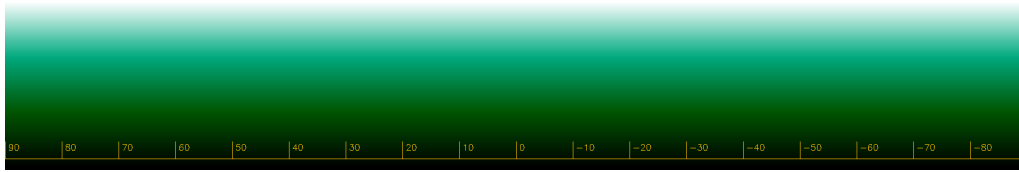


Figure 3.11 Depth image representing a 180 degrees lidar measurement swipe of the scene illustrated in Figure 3.9. The color of each cell shows estimated depth, with darker colors being closer than lighter colors.

3.4.2.3 Height grid generation

A full lidar swipe generates an array of range measurements which we can represent as a depth image, as shown in Figure 3.11. Each pixel in this image represents a range between the lidar origin and the surrounding scene. Because the elevation and azimuth angle of each beam is known, we can convert the depth image to a *point cloud image*.

The main contribution of our terrain height estimation method is to use this array structure of the lidar swipe to make a parallelizable process. We do this by creating a collection of meshes from the lidar measurements and render an image of the undersurface of the meshes with the view looking up along the z -axis (i.e., up-down axis). Because the undersurface represents the lowest measured open space of the surroundings, the pixel values of the rendered image will correspond to the estimated height values of the terrain. By structuring the computation this way, we can use a GPU to efficiently process a whole lidar swipe simultaneously.

The exact method for generating the height map is as follows:

1. For each row in the point cloud image, we construct triangles fan primitives⁷ by connecting two neighboring points with the lidar origin.
2. We connect all the triangle fan primitives on a row together to form a single triangle fan mesh, as illustrated in Figure 3.12a.

⁷https://www.khronos.org/opengl/wiki/Primitive#Triangle_primitives

-
-
3. All the meshes in the different rows together form a mesh collection, as seen in Figure 3.12b.
 4. We render the mesh collection to an image with the camera positioned below the meshes, looking up along the z-axis, as illustrated in Figure 3.12c.
 5. The resulting image is a height map of the terrain, where each pixel value corresponds to the height of the terrain at that pixel. The area between the center of the lidar and where the bottom lidar beam hits the ground is treated as unknown measurements. An illustration of the rendered height map can be seen in Figure 3.12.

This means we can generate an estimation of the terrain height around the lidar from the measurements of a single lidar swipe, and we call this a *single swipe height map*.

3.4.2.4 The obstacle map

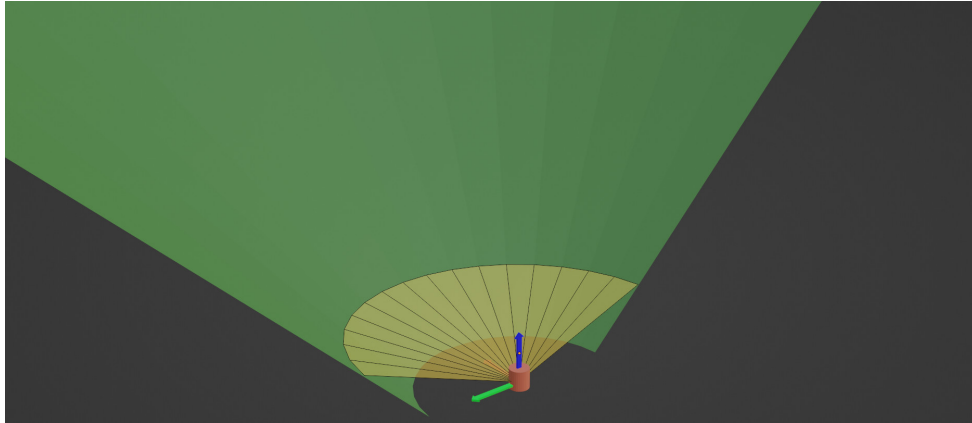
Once we have estimated the *single swipe height map*, we can use it together with the point cloud to classify the surroundings as obstructed or traversable. Objects above the terrain surface, such as rocks, debris, and branches *could* be obstructive, but they could also be small enough or hang high enough for the UGV to pass over or under them. In our method, we mark a cell as obstructed if there is a point within the cell area at a height above the manageable threshold height for the UGV to drive over, and below the passable height for the UGV to drive under as illustrated in Figure 3.14. By looping through every point in the point cloud, we create a *single swipe obstacle map*. This is a map where each cell is classified as either *not measured*, *obstructed* or *traversable*. An example is shown in Figure 3.15

3.4.2.5 Temporal fusion of terrain and obstacle maps

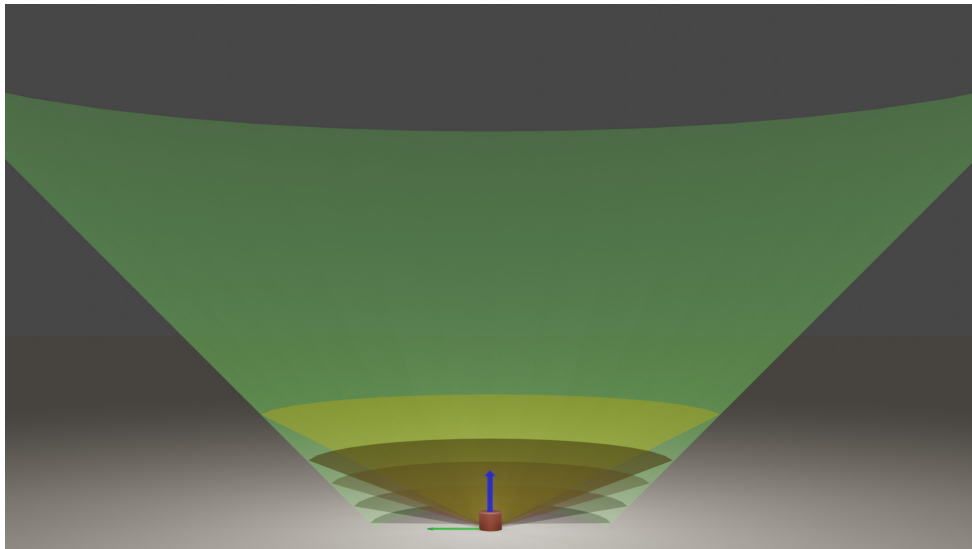
We have mounted the lidar high up in the front of Tor and tilted down 10 deg. The lowest lidar beam is angled down 22.5 deg from straight ahead. This means the lidar cannot see the ground below out to a certain distance, and effectively has a blind zone around itself. We can see the blind zone in Figure 3.15, around the center where Tor is placed. The lidar also has smaller blind zones due to occlusion from the chassis of Tor and its payload.

To remedy this, we fuse single swipe maps together over time to form one large map. In this way, “holes” in the map are removed as Tor moves across the terrain, getting measurements from areas previously covered by blind zones. When fusing maps, we can either use a static origin, e.g. a “world” point, or a dynamic origin, a “local” point. The latter method involves updating the map origin between short intervals. The main advantage of this is that points close to the origin have smaller uncertainties than points far away because the measurements are more recent. However, it comes with the disadvantage that *users* of the map (e.g., the motion planner) must have a way of knowing when the origin is updated.

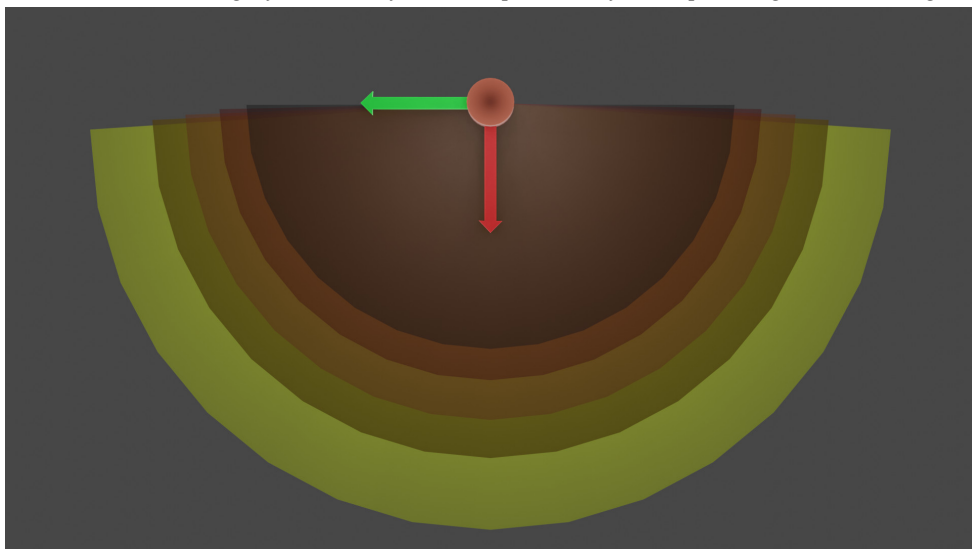
To avoid the origin synchronization, we fuse the UGV’s map using a static origin, and in the frame Q , described in Section 3.3. Because Q is a static frame that continuously integrates the UGV’s velocities, it prevents discrete jumps in the map, as illustrated in Figure 3.6. The drawback of this is that the map is prone to drifting away from the true global position. However, this would



(a) Triangle fan mesh generated from depth values of a single row of the depth image shown in Figure 3.11.



(b) Stacked triangle fan meshes from multiple rows of the depth image shown in Figure 3.11.



(c) The same meshes as in Figure 3.12b, but rendered with the camera positioned underneath looking up, i.e., positive z -values are into the image.

Figure 3.12 Using triangle fans to create a mesh

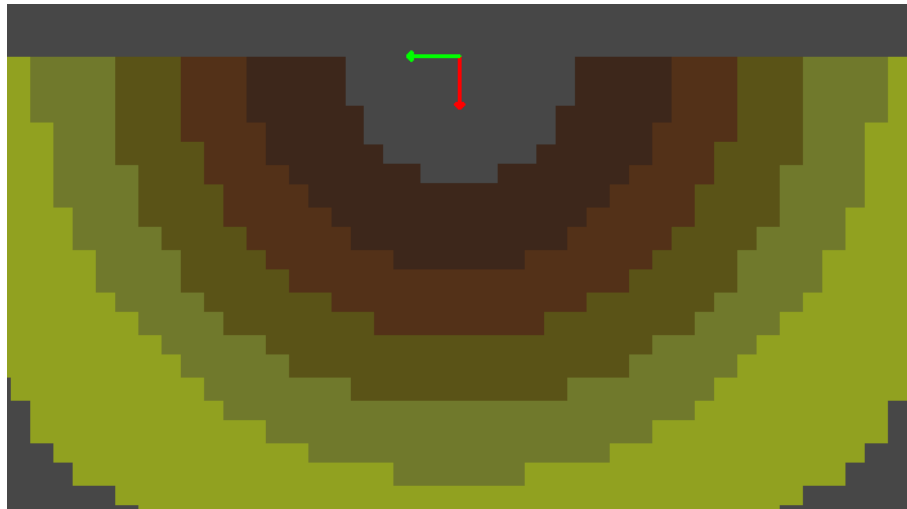


Figure 3.13 Terrain height grid created from rendering the meshes shown in Figure 3.12c. We have included the axis vectors for illustration purposes. Darker colors represent lower height values, dark gray is unknown/not measured.

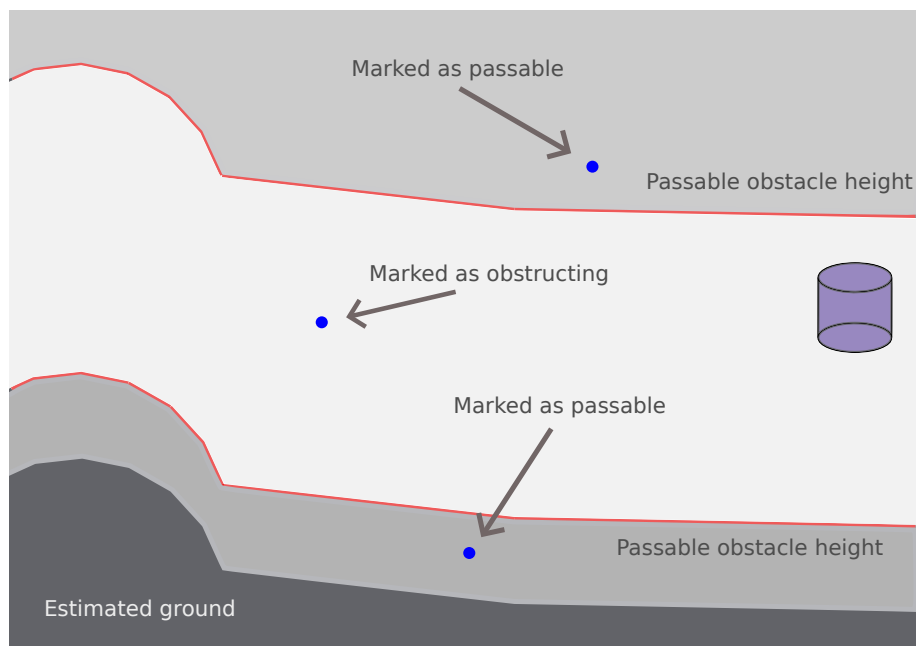


Figure 3.14 A side view illustration of the obstacle classification. If a point is measured in any of the light gray zones, the point is not classified as an obstacle because the UGV could drive over or below it. If a point is anywhere within the white zone, the point is classified as an obstacle, and the cell is marked as obstructed.

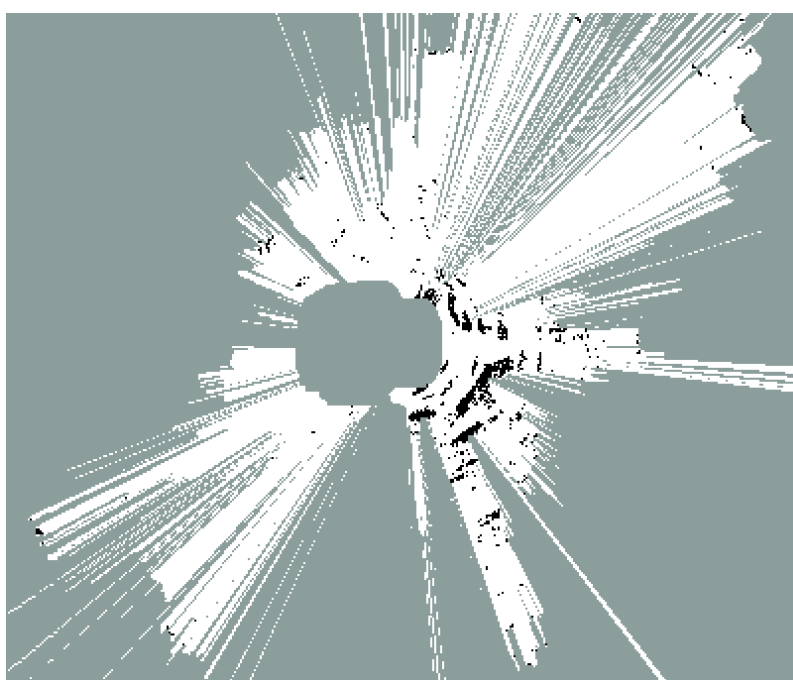


Figure 3.15 A full obstacle map from a single swipe height map. The data is from a recording captured with Tor. Black cells are classified as obstacles, white cells are traversable, and gray cells are not measured and treated as unknown in this swipe.

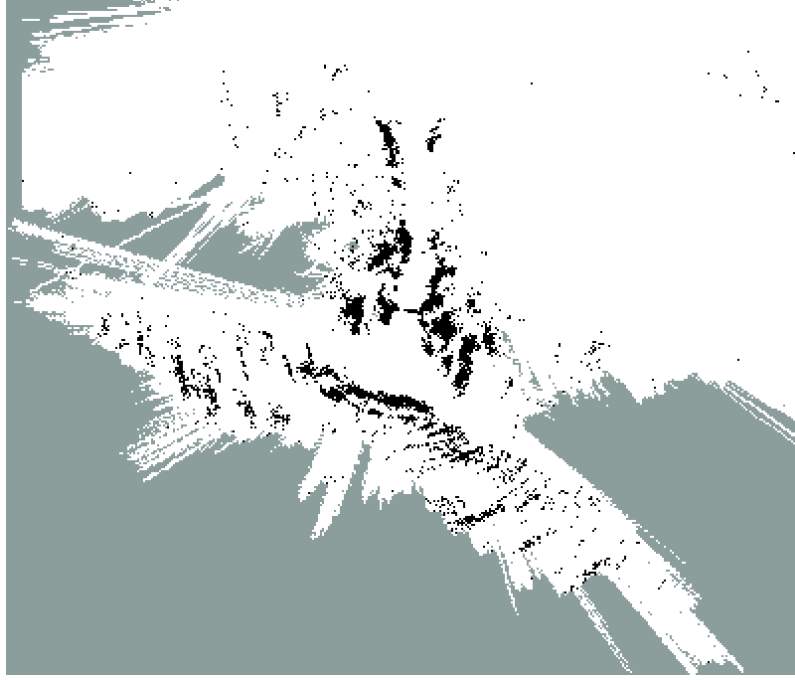


Figure 3.16 An obstacle map created by fusing multiple single swipe obstacle maps (as seen in Figure 3.14)

only be a problem if the UGV needed to return to a previously mapped location and use this map data. To drive autonomously in terrain, we require that the UGV is capable of mapping the terrain sufficiently for driving autonomously on its first run. Using previously mapped data is therefore out of scope for this project.

Because the obstacle map is a map where the cells only have three states (*obstructed*, *traversable* and *not measured*), we use a simple strategy for fusing obstacle maps: new measurements overwrite old measurements. If a cell from a new single swipe map has no measurement (e.g. because it is within a blind zone), we keep the previous measurements. This way, we immediately fuse new obstacles into the map, while we still remember obstacles that we no longer see. An example of a fused obstacle map can be seen in Figure 3.16.

Because a height map is a map where cell states are represented by a floating-point number, we need a more complex method for fusing the height data, and we have chosen the method from [7]. This method divides the map fusion in two steps:

1. A motion update step.
2. A measurement update step.

To achieve this, for every cell height value z_{ij} located in row i and column j , we create a three-dimensional position variance-covariance matrix $\Sigma_{x_{ij}}$, where

$$\Sigma_{x_{ij}} = \begin{bmatrix} \sigma_{x_{ij}}^2 & \sigma_{xy_{ij}}^2 & 0 \\ \sigma_{xy_{ij}}^2 & \sigma_{y_{ij}}^2 & 0 \\ 0 & 0 & \sigma_{z_{ij}}^2 \end{bmatrix} \quad (3.4)$$

where $\sigma_{x_{ij}}^2$ and $\sigma_{y_{ij}}^2$ are the variances of the position in the x and y directions, and $\sigma_{z_{ij}}^2$ is the variance of the height measurement.

In the motion update step, if the UGV has moved, we move it from its current position to a new position. We also propagate the variance from the UGV's motion through the variance-covariance values of every cell in the fused map. This *increases* the variance-covariance of every affected cell, as illustrated in Figure 3.17.

Every position and transformation is represented with respect to the Q frame, and we omit the row and column subscript (i, j) for readability. Then, the change in position $\mathbf{x}$ between step n and $n + 1$ can be represented by the transform $\mathbf{T}_n^{n+1}$:

$$\mathbf{x}^{n+1} = \mathbf{T}_n^{n+1} \mathbf{x}^n \quad (3.5)$$

We also propagate the variance-covariance between the two positions as described in [14]:

$$\Sigma_{\mathbf{x}_{n+1}} = \mathbf{J}_{\mathbf{T}_n^{n+1}}^{\mathbf{x}_{n+1}} \Sigma_{\mathbf{T}_n^{n+1}} \mathbf{J}_{\mathbf{T}_n^{n+1}}^{\mathbf{x}_{n+1} \top} + \mathbf{J}_{\mathbf{x}_n}^{\mathbf{x}_{n+1}} \Sigma_{\mathbf{x}_n} \mathbf{J}_{\mathbf{x}_n}^{\mathbf{x}_{n+1} \top} \quad (3.6)$$

where (3.8) transforms the old variance to the new pose, and (3.8) is the term for variance from motion.

$$\mathbf{J}_{\mathbf{T}_n^{n+1}}^{\mathbf{x}_{n+1}} \Sigma_{\mathbf{T}_n^{n+1}} \mathbf{J}_{\mathbf{T}_n^{n+1}}^{\mathbf{x}_{n+1} \top} \quad (3.7)$$

$$\mathbf{J}_{\mathbf{x}_n}^{\mathbf{x}_{n+1}} \Sigma_{\mathbf{x}_n} \mathbf{J}_{\mathbf{x}_n}^{\mathbf{x}_{n+1} \top} \quad (3.8)$$

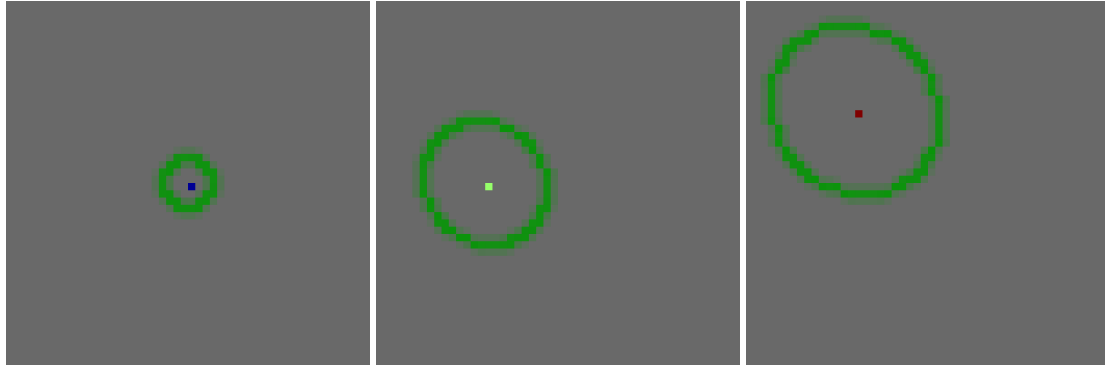
For methods for computing the Jacobians $\mathbf{J}$ we refer to [14].

In the measurement update step, we fuse height values from a single swipe terrain map into the fused map. This *decreases* the variance-covariance of every affected cell. Two values are fused: the height (z_{ij}) and its variance $\sigma_{z_{ij}}^2$. We set the position variance-covariance values ($\sigma_{x_{ij}}^2, \sigma_{y_{ij}}^2, \sigma_{xy_{ij}}^2$) to values dictated by the lidar sensor model. In this way, recently updated cells have the lowest variance-covariance values. We fuse new height and height variance values (z_{new} and $\sigma_{z_{new}}^2$) into the map using a one-dimensional Kalman filter (again omitting subscripts i, j):

$$z_{n+1} = \frac{\sigma_{z_n}^2 z_n + z_{new} \sigma_{z_{new}}^2}{\sigma_{z_n}^2 + \sigma_{z_{new}}^2} \quad (3.9)$$

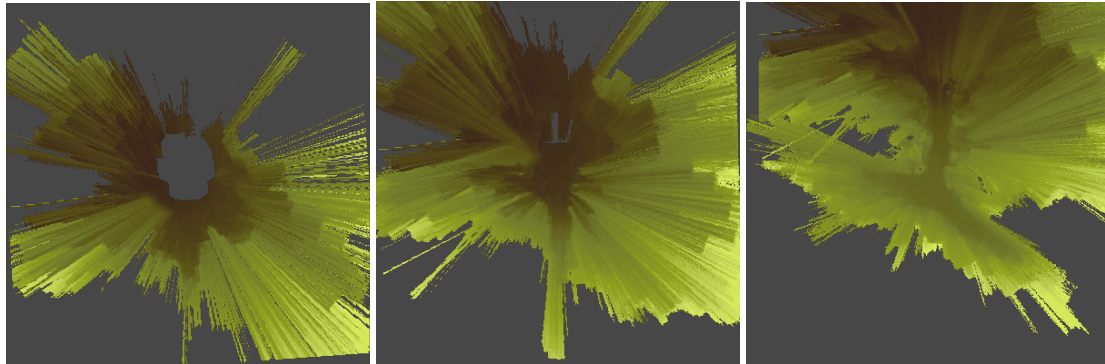
$$\sigma_{z_{n+1}}^2 = \frac{\sigma_{z_n}^2 \sigma_{z_{new}}^2}{\sigma_{z_n}^2 + \sigma_{z_{new}}^2} \quad (3.10)$$

An example of a fused terrain map can be seen in Figure 3.18.



- (a) An initial measurement in a cell. Dark blue color indicates low height variance of the measurement. The green ring shows position variance.
- (b) We have moved 10 cells to the right, which has moved the measured cell 10 cells to the left in the image. Notice the increase in position covariance and height variance.
- (c) We have moved 10 cells down, which has moved the measured cell 10 cells up in the image. The position covariance and height variance have both increased further.

Figure 3.17 Propagation of covariance. The green ring size shows position covariance, and the cell color shows height variance with blue being low variance, yellow medium and red high. Both position and height variance increase with motion.



- (a) Initial fused terrain map consisting of a few single swipe terrain maps.
- (b) The fused terrain map after adding more single swipe terrain maps to the map in Figure 3.18a. Notice that because we have moved downwards in the map, the area from Figure 3.18a is moved upwards.
- (c) The fused terrain map after adding more single swipe terrain maps to the map in Figure 3.18b. Notice that because we have moved downwards in the map, the area from Figure 3.18b is moved upwards.

Figure 3.18 Example of three stages of a fused terrain map.

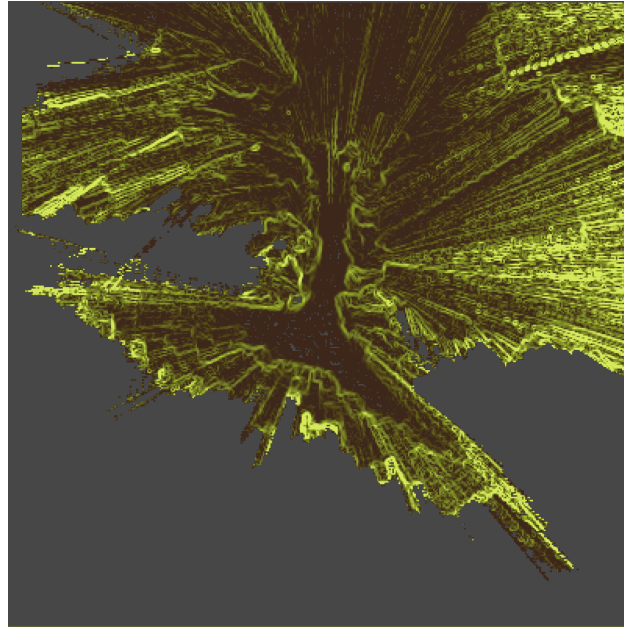


Figure 3.19 Inclination map created from the terrain map seen in Figure 3.18c. Brighter yellow indicates higher inclination. Notice how we can make out an area with low inclination in the middle of the map, which indicates that the terrain is flat here.

3.4.2.6 Generation of the inclination map

An inclination map marks the difference in height between cells, and will typically highlight steep slopes in the terrain. This is useful information for planning as it may be used to find flat areas, which can be beneficial to increase the likelihood that the given path is traversable. Areas with higher inclination can be avoided, as they are less likely to be traversable, even though they might not be marked as obstructed in the obstacle map. In addition, the inclination map acts as height curves in a normal map, making it easier for a human to interpret the data. Because an inclination map is simply marking the gradients in the terrain, the Sobel operator [49] can be used to extract the inclination map from the fused terrain map. An example of an inclination map can be seen in Figure 3.19.

This way of finding inclination provides the inclination of the observed surface of the surroundings, and not necessarily the true terrain. Therefore, the boundary to an obstacle like a bush will yield a high inclination, and therefore be intraversable and will be marked as obstacles in the obstacle map. Some of the yellow high-inclination measurements at the edges of the observable space are not true high-inclination areas, but rather areas where we have few measurements which causes ray-like height measurement artifacts. As the UGV gets closer to these areas it will get more measurements and the artifacts will disappear, leaving the map based on the true height differences of the surroundings.

3.4.3 Visualization of situational awareness

Warpath provides tools for visualizing processed data and monitoring the UGV's performance while running. For visualization purposes, we combine obstacle maps and inclination maps to a single cost map that we can view in, for example, the ROS visualization tool RViz. An example of a combined inclination and obstacle map can be seen in Figure 3.20a.

Tor has a camera mounted on its sensor rack, and we use this for an alternative visualization by projecting the combined map into the camera image as an overlay. The way we achieve this is as follows:

1. We create a surface mesh from the height map.
2. We render the cost map onto the surface mesh, with a customizable color map.
3. We create a virtual camera in the same position as the real camera. We then create a virtual image of the surface mesh at the same timestamp as the camera image.
4. We overlay the virtual image of the surface mesh with the camera image and publish it.

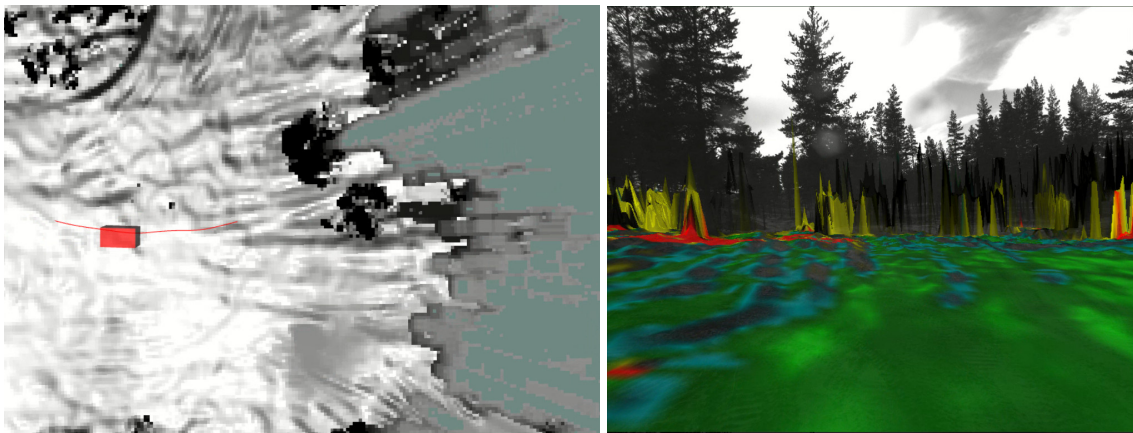
The result is a camera image with a colored overlay of the cost map projected into the image.

We can see an example of this in Figure 3.20, where we have the cost map on the left, and the same cost map projected into the camera image on the right.

To achieve this, we have a couple of requirements for the system:

- We need to know the camera's intrinsic and extrinsic parameters, which we can get from a calibration process such as [10].
- We need to know the camera position relative to the lidar with high accuracy. This can be achieved by methods that do camera-lidar calibration, such as the one described in [56]. This particular method can also be used to calibrate the intrinsic camera parameters at the same time.
- Lastly, because we are dealing with sensors that do not capture the environment at exactly the same time, we need some way to synchronize the data streams. We achieve this by having a common time reference, which originates from the time sync server. As both the lidar and camera timestamp their sensor data, we can look up their positioning at the time of capture, and make sure that the virtual camera lines up with the physical camera at the time of capture.

The projected cost maps can be a useful tool for an operator that is monitoring the autonomous driving of Tor. Using only camera images, it can be difficult to get a good understanding of the geometry in the scene, and the top-down view is often not the best when driving forwards. The projected cost map can therefore provide an enhanced view, that helps with building situational awareness. In Figure 3.21, we can see more examples of how the projected cost maps provide a better understanding of the geometry in the scene with the edges of the road standing out.

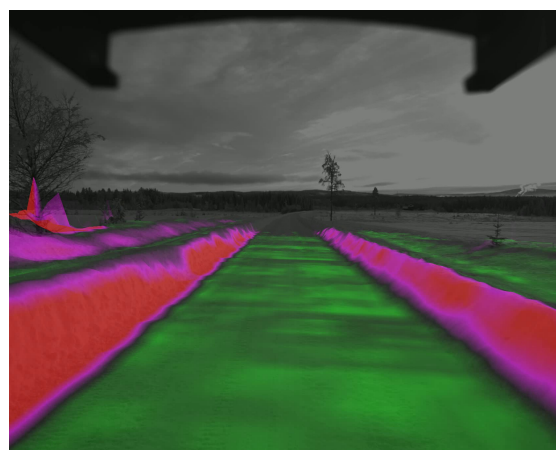


(a) Top-down view of cost map. The UGV is the red box, (b) The cost map projected into the camera image. facing to the right. Black is obstacles, the gray scale Red is obstacles, while the scale from green to yellow indicates increasing inclination. represents inclination values, with white being flat and therefore preferable for driving.

Figure 3.20 Both the top-down view of the cost map and the same cost map projected into the camera image.



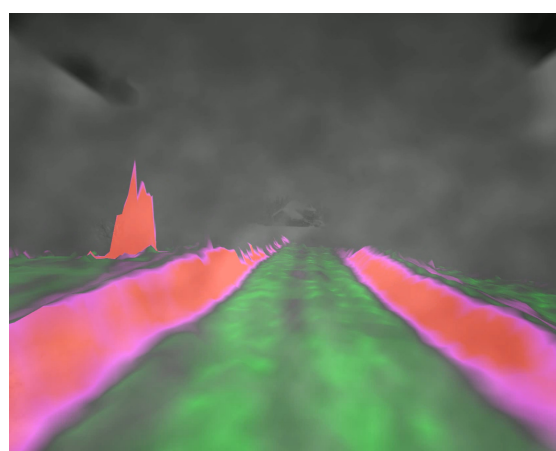
(a) Camera view from a winter road with plow edges to the sides.



(b) Cost map projected into the same camera image. Plow edges are easily distinguishable from the road.



(c) Camera lens covered by snow, with practically no visibility.



(d) The cost map is based on lidar which still works. We can make out the road, despite not seeing anything in the camera image.

Figure 3.21 Examples of how cost map image projection can give improved situational awareness for an operator. Only camera images to the left, and the same images with cost map projection on to the right.

3.4.4 Image-based object detection with AprilTags

The other of the two processing pipelines is the pipeline for image-based object detection. This pipeline is used to estimate the pose and velocity of a target object, such as another vehicle. The purpose was to have a simple, yet robust method for making the UGV follow a lead vehicle autonomously. The implementation detects and tracks AprilTags, which are fiducial⁸ markers designed for robust image-based pose estimation.

An example of an AprilTag marker can be seen in Figure 3.22. The AprilTag method is widely used due to its reliability and the availability of pre-generated markers and open-source implementations⁹ [33]. AprilTags have several benefits:

1. The markers have a geometry rarely found in nature, which means the probability of finding a marker pattern in natural clutter is extremely low.
2. The markers are encoded with an ID payload that makes them distinguishable from one another.
3. It is possible to decode the ID payload and perform the pose estimation even though the environment and lighting conditions are demanding. The decoding is invariant to the rotation of the marker and the design facilitates error correction.

This image-based object detection is not as sophisticated as the work that has been done on the terrain mapping. The field of image-based object detection is a highly active field of research, and there are many different methods available that do not rely on artificial markers. For example, the YOLO [40] and SSD [22] methods from 2015 are both capable of detecting objects in images without the need for markers. We have also experimented with these methods, such as the example seen in Figure 3.23 from 2018, but never integrated them into Warpath.

Using AprilTags was a good first step to get a full object detection and vehicle following system up and running. As vehicle following was conducted as a feasibility experiment showing the capabilities of autonomous vehicle following only, using AprilTags served for a proof-of-concept. However, we do not believe relying on markers is the way forward. If we want to get serious about

⁸Fiducial marker: A reference marker used to provide a known position or orientation in an image or physical space, enabling accurate measurement, alignment, or localization.

⁹<https://github.com/AprilRobotics/apriltag>

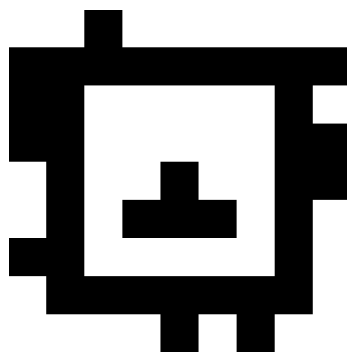


Figure 3.22 Example of an AprilTag marker.



Figure 3.23 Example of image-based detection using Yolo from 2018. The detected vehicle is our old UGV Olav, a Polaris Ranger.

vehicle following in the future, we will replace the AprilTag method with a more sophisticated method that does not require markers.

3.4.4.1 The AprilTag detector

The goal of this process pipeline is to estimate the pose and velocity of a target object, such as another vehicle. To achieve this, we attach a marker to the object. A camera captures images of the scene, and the AprilTag method processes these images to determine the marker's pose. By analyzing the pose differences over time, the system calculates the marker's velocity. Warpath then publishes the estimated pose on the ROS network so the other autonomy modules can use them.

The AprilTag detection process has two parts: the detector and the decoder. The detector identifies markers in an image and estimates their pose, while the decoder extracts the marker's ID from its encoded payload. The detector has a high false positive rate to ensure that it detects all tags in an image. Then, the decoder will verify the detections and filter out the false ones.

The detector looks for four-sided regions, called *quads*, that have darker interior than exterior. It finds these by looking at the gradients of the image, clustering the gradients with same directions and magnitude together, and from these clusters extract *line segments*. The quads are detected from the line segments by searching for lines that form a 4-sided shape. Once it has detected all quads, the detector computes the 3x3 homogeneous projection from 2D points in a marker's coordinate system to the camera 2D image coordinate system using Direct Linear Transform (DLT). It then uses the projection together with the known camera parameters and marker sizes to estimate the pose of the marker. When it has calculated the pose of a marker, it orders the detected quads to create the structure of a marker, and decodes the payload and ID.

3.4.4.2 Hardware and software for AprilTag detection

To perform the image-based pose estimation, we only need a camera and a localization system. In addition, we need to attach a marker to the object that we want to track. Then, we can run the pose estimator: It receives the captured image and applies the AprilTag detection method to get the marker's pose. The velocity of the marker is calculated from the pose difference over time. Warpath then publishes the pose and velocity on ROS topics for use by the other modules. In Figure 4.1, we see the system from an experiment where we tested following, and we see the marker attached to the lead vehicle.

3.5 Trailblazer – Deciding where to drive

Trailblazer is the software module that determines exactly where Tor drives. It creates plans for where to drive so that Tor reaches the waypoint from HAL while avoiding obstacles and getting stuck. Trailblazer uses two inputs – the route from HAL and the local map from Warpath – to plan a path from Tor’s current position to the next waypoint, ensuring it avoids obstacles and areas that cannot be traversed. This planning process is referred to as *motion planning*.

3.5.1 Motion control strategies

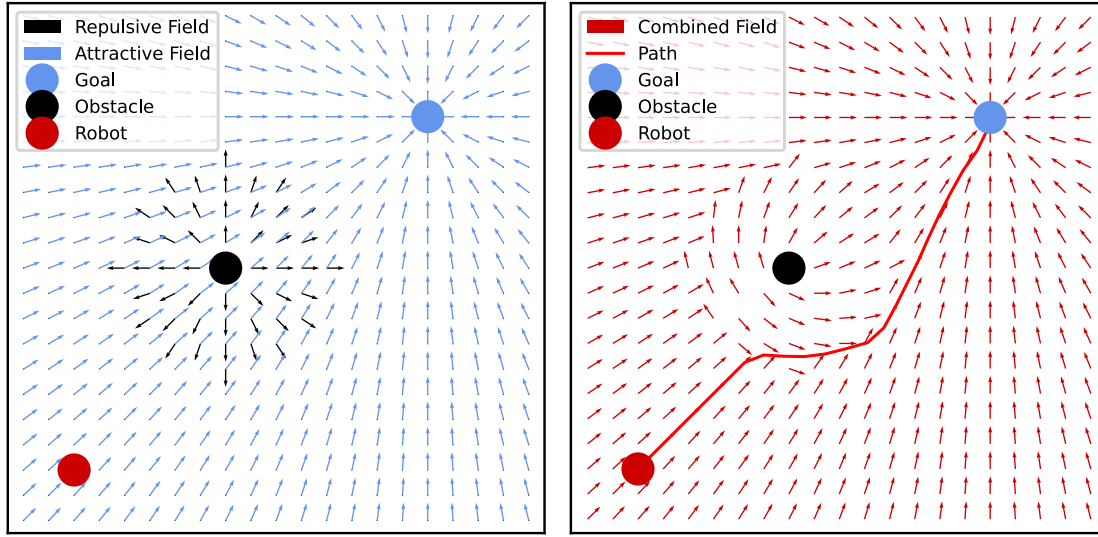
To guide a UGV towards a waypoint while at the same time considering the environment, there are two main strategies that can be employed: *Reactive* planning, which sets steering inputs directly based on the environment and *deliberate* planning, which performs a more rigorous planning that considers how the UGV will behave in the future.

3.5.1.1 Reactive planning – Collision avoidance

Reactive planning is a control strategy where the UGV calculates a steering action based on the current state of the UGV itself, the goal and the environment. The UGV reacts to changes in the state, and selects the best immediate action, without regard for if that action is optimal in the long term. Consider a UGV that drives towards a goal and encounters an obstacle slightly to the right. By steering to the left and steering more as it gets closer to the obstacle, the obstacle can be circumvented and the UGV can resume driving towards the goal. This is a reactive strategy, the UGV reacts to the obstacle by setting the control action or steering input directly. The strategy described is also referred to as *collision avoidance*, a reactive strategy that seeks to avoid collisions.

A way to implement a reactive strategy is the artificial potential fields method [46, pp. 546–554]. This method creates virtual potentials or forces in the environment; the goal generates an attractive force that steers the UGV towards it, and obstacles generate repulsive forces that steer the UGV away from them. The combination of the two creates a direction that the UGV can follow. An example of the artificial potential fields method is shown in Figure 3.24. Here, the goal in red has attractive forces shown as vectors in red, and the obstacle in black creates repulsive forces shown as black vectors. By taking the sum of the vectors, we obtain the blue vector and the UGV will attempt to steer in this direction. By repeating this process, the UGV will follow the path in blue towards the goal.

Pure collision avoidance is easy to implement and computationally inexpensive, which can be an advantage to react fast to changes in the environment. However, there are drawbacks of only using collision avoidance. If the environment is dense with obstacles, collision avoidance will struggle to successfully guide the UGV through. Collision avoidance can get the UGV stuck because evasive maneuvers due to some obstacles can lead it on a collision course with other obstacles. It also runs the risk of becoming stuck in dead-ends because it only considers the immediate obstacles and ignores obstacles further down the line. In other words, collision avoidance needs sufficiently open spaces to complete the avoidance maneuvers. In more complex environments, it is therefore necessary to employ deliberate planning strategies that consider more than the immediate obstacles.



(a) The goal has an attractive vector field and the obstacle has a repulsive vector field in its vicinity. (b) The combined vector field. Following the vector field from the robot yields a path to the goal.

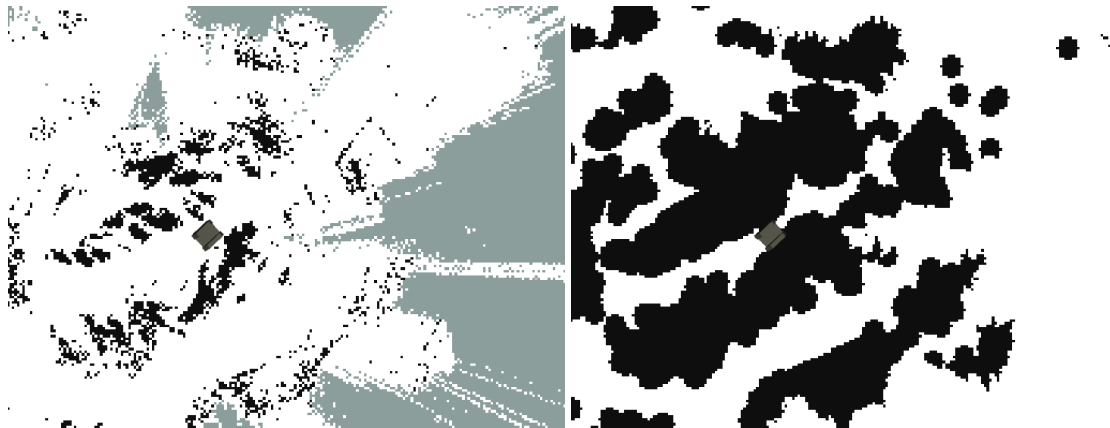
Figure 3.24 An example of artificial potential fields. The robot moves towards the goal until it gets close to the obstacle, and will then make an evasive maneuver before continuing.

In Chapter 1 we introduced a set of benchmark routes, and we have hypothesized the autonomy requirements for successfully following them in Appendix A. For *Route A – Open and flat terrain*, collision avoidance would be sufficient as the environment is open and sparsely populated with obstacles, and with enough space to navigate around them. For the other route types, however, pure collision avoidance would most likely not be a successful strategy as the environments are too complex; the environments can be expected to feature both dead-ends and have obstacle density so high that avoidance of one obstacle will interfere with avoiding the next ones. Due to the limited usefulness of obstacle avoidance in the benchmark routes of Chapter 1, we have chosen to not implement collision avoidance in *Trailblazer* and instead use a deliberate motion control strategy.

3.5.1.2 Deliberate planning – Motion planning

Deliberate planning or *motion planning* involves creating a plan for how the UGV should move in the future, rather than simply reacting to the current situation as in collision avoidance. This approach enables the UGV to perform more complex maneuvers that are not possible with a purely reactive strategy. For example, the issue of performing evasive maneuvers from one obstacle that puts the UGV on collision course with another can be solved by making a plan that takes both obstacles into account. The issue of getting stuck in dead-ends can be avoided because a motion planning algorithm can detect that there is not an open passage, and can instead make a plan that takes the UGV around the area with the dead-end.

The goal of motion planning is to find collision-free trajectories or paths from the current UGV pose to the given goal position or pose. A path is a sequence of poses that are connected such that the



(a) Original obstacle map with Tor in the center. (b) The same obstacle map with dilated obstacles.

Figure 3.25 An obstacle map that has been dilated. Tor was driving up and to the right along a vehicle trail, but the trail has been blocked off in all directions due to the dilation.

UGV can follow them continuously. A trajectory is the same as a path but also contains information about the velocity along the path. For example, a trajectory can say that the UGV should drive fast along straights and slow down before turns, whereas a path only has information about where the straights and turns are. By the most common definition, planning of paths is referred to as path planning and planning of trajectories is referred to as motion planning. Trailblazer supports both path planning and motion planning, and we will use the term motion planning from here on as it is the wider term.

Basic motion planning needs three inputs to work: an obstacle map, its own pose relative to the map and a goal position. Its objective is to find a trajectory from the starting pose to the goal without collisions with the environment as given in the map. The simplest map is a binary map, typically given as a grid map where every cell is either obstacle or free space. For many environments, an obstacle map is sufficient for motion planning. For example, we hypothesized that for *Route C – Wide and open vehicle trail*, motion planning with obstacle maps would be sufficient. That is because we assume the vehicle trail is drivable, and that motion planning would make sure that the UGV avoids obstacles and stays on the trail.

When planning trajectories for a UGV, the size of the UGV needs to be accounted for. One simple way of achieving that is by increasing the size or dilating the obstacles in the obstacle map by half the size of the UGV. After dilating, motion planning can be done by assuming that the UGV is a point mass with no size. Then, as long as the planned trajectories are collision-free in the dilated obstacle map, the UGV will not collide with obstacles when following the trajectory. An example of how the dilation looks for an obstacle map can be seen in Figure 3.25, where all obstacles have been dilated. We treat the unknown space in the map as free space when performing the dilation to avoid limit potentially valid paths. An effect of the dilation is that small and close obstacles get grouped together as larger obstacles. This reflects the fact that the UGV cannot pass through small gaps, and with the dilated obstacles, it's easier to spot where there are valid passages and where it is blocked.

Tor and many other UGVs are capable of performing neutral turns where it rotates on the spot. The neutral turn capability makes it theoretically possible for such UGVs to follow any kind of path;

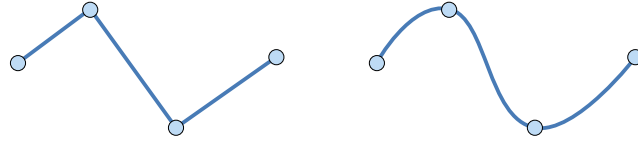


Figure 3.26 Tor could follow the piece-wise linear path on the left using neutral turns at each waypoint, but it would be more efficient to follow the smooth path on the right.

even a piece-wise linear path, a non-smooth path with sharp turning points. In practice, however, performing neutral turns requires Tor to come to a complete standstill, and its turn accuracy is not great unless performed slowly. Therefore, following paths using neutral turns is slow, inefficient and also less accurate than path following while moving forwards. Hence, the motion planner should plan paths that are smooth with no sharper curvatures than Tor is capable of following while driving forward. The difference is illustrated in Figure 3.26: Tor could in theory follow the path on the left with straight segments using neutral turns at each waypoint, but it would be more efficient to follow the path on the right. This constraint for Tor is called a *non-holonomic* constraint, which means that Tor cannot move freely in all directions [21].

Motion planning algorithms often aim to find the shortest path. This type of planning is also called *optimal motion planning* where the algorithm aims to find the path with the lowest “cost value” for some “cost function”. For finding the shortest path, the path length or the distance covered is the cost function, and the planning algorithm tries to find the path with the lowest path length or distance, i.e. lowest cost value. When planning for a non-holonomic vehicle, finding the exact path that is shortest is challenging, and in many cases, an approximated shortest path is sufficient. There are other types of cost functions as well; the environment map can, for example, be a cost map where every cell contains a cost value, which says what the cost of visiting each cell is. A cell that represents road could be assigned a lower cost than a cell that represents grass, and the planning algorithm could take advantage of the difference in cost to prefer paths on roads over grass. Cost maps are useful for motion planning in terrain where you could assign low cost values to areas you believe are more traversable or preferable, and high cost to areas with low traversability. That way you could make the planner prefer areas with vehicle trails, flat areas, and areas without vegetation, whereas rougher terrain and areas with vegetation would be avoided if possible.

An example of how cost maps can be used for motion planning can be seen in Figure 3.27, which is from experiments we performed in 2019[51] with our former Polaris Ranger UGV. Here we can see a cost map, where white is the highest traversable areas with the lowest cost, and the darker the shade, the higher the cost, and black are obstacles. In those experiments, we did the same run first using only a binary obstacle map (run 1–3), and then switched to using a cost map for run 4 and 5. With the cost map runs, the motion planning algorithm preferred lower cost areas, which resulted in the UGV taking a different path and avoiding a high-cost area where the UGV got stuck in run 1–3.

3.5.2 Motion planning algorithms

There are a wide variety of motion planning algorithms that exist. For a general overview of motion planning theory and algorithms, we refer to [21]. For more recent work for motion planning for

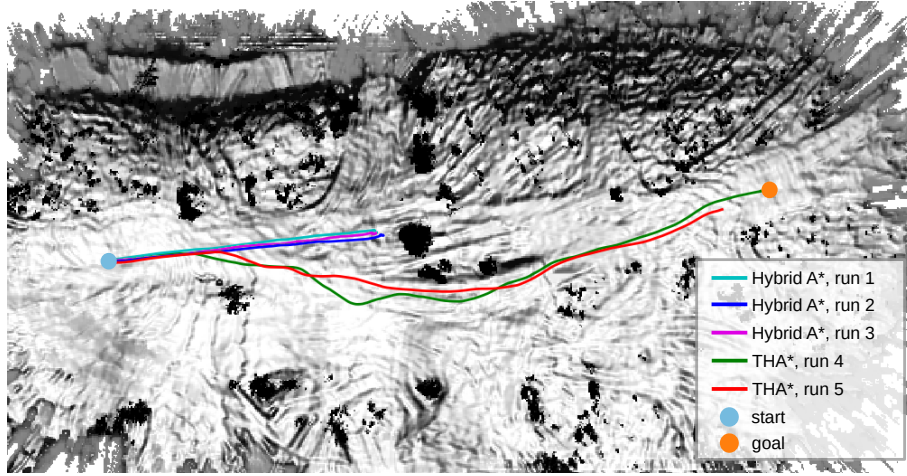


Figure 3.27 Real-life experiments with motion planning using a cost map[51]. Darker shade represents higher cost, and black are obstacles. Run 1–3 only used a binary obstacle map, whereas run 4 and 5 used a cost map and actively tried to drive in lower cost areas resulting in different driven paths. More on the methods Hybrid A* and Traversability Hybrid A* in Section 3.5.2.3 and Section 3.5.2.2.

car-like vehicles, [37] and [13] are good options. In this section, we present the algorithms that we’ve implemented and run with our stack.

3.5.2.1 Planning algorithm – the Fast Marching Method

In Trailblazer, one of the motion planning methods we have implemented is based on the Fast Marching Method (FMM)[45]. The FMM is in many ways similar to the artificial potential fields, where an attractive potential guides the path towards a goal. FMM itself is an algorithm that can be applied to a grid map from a source, and then it will expand outwards like a wavefront and find the distance from every cell to the source. The wavefront will not move through any obstacles, but instead move around like a real wave would. An example can be seen in Figure 3.28, where FMM is applied from a source or goal point, and we can see the contour lines for the distance to the goal which resembles waves that spread from the goal.

In the Fast Marching field, its reverse gradient has the quality that it always points in direction of the shortest way to the source. Therefore, by following the direction of the reverse gradient incrementally, the shortest path from any point to the goal can be obtained. This is similar to the artificial potential fields, except that the attractive field will always point towards the boundary of the obstacles instead of directly towards the goal. In Figure 3.28, we can see the shortest path from an arbitrary start point to the goal point, which is obtained using this strategy. It must be noted, the Fast Marching Method only provides an approximation of the true distance to the goal, and any path found with this method is an approximation to the shortest path, albeit a good one.

The Fast Marching Method also works with cost maps and not only binary obstacle maps. The cost map can be treated as a speed map of how fast or slow the wave spreads. By assigning a high speed for low-cost values and a low speed for high-cost values, the wavefront will spread fastest in

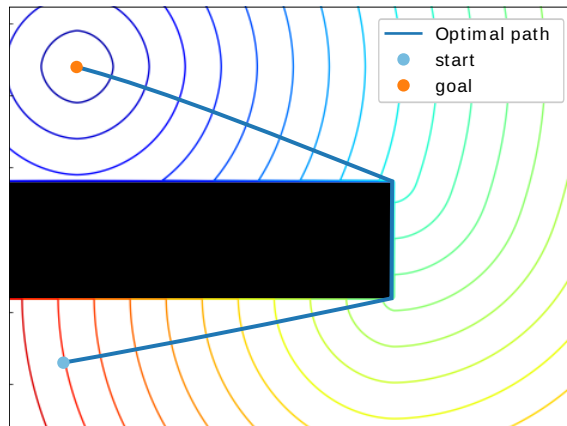


Figure 3.28 The Fast Marching Method creates an attractive field from a source, that follows obstacles. The shortest path is found by following the gradient towards the goal.

areas with low cost. An example of this effect can be seen in Figure 3.29, where we can see the Fast Marching Method applied to a cost map from the source in orange. The wavefront spreads faster in the lighter low-cost areas, which means that the optimal path is not a direct line towards the goal, but rather a curve moving through lower-cost areas.

In Trailblazer, we have implemented a motion planner based on the Fast Marching Method, which can run using both binary obstacle maps and cost maps. We apply the Fast Marching Method from the current goal point to obtain a Fast Marching (FM) field. Then, if the UGV is facing in the direction of the FM field, we can obtain the optimal path by gradient descent (seen in Figure 3.28 and Figure 3.29), and use this path directly.

If the UGV is not aligned with the FM gradient field direction, we created a path that aligns the UGV with the Fast Marching field. We do this by applying the maximum steering input for the UGV, making the path have as tight a turn as possible. From the point where this path is aligned with the field, the optimal Fast Marching path is used towards the goal.

This strategy is not without issues, as it could cause collisions if there is an obstacle in the path that aligns the UGV with the Fast Marching Field. In Figure 3.30 we see such a situation, where the UGV is not aligned with the FM field. The optimal Fast Marching path in blue is collision-free, but the path to align the UGV with the field in red causes a collision. In these cases, the algorithm would not produce a path, and therefore fail.

In practice, using this Fast Marching motion planning method performs well in many situations. As long as the UGV is heading in a direction reasonably close to the FM field, failing to find a path due to obstacles is rarely an issue. In most situations, the UGV will be heading in the general direction of the goal, in which case it typically is reasonably aligned with the field. The benefit of the Fast Marching strategy is that it is good at finding the optimal path, and the path will always point straight towards the edges of obstacles, which means few unnecessary maneuvers and predictable movement. In relatively sparse environments where few aggressive turning maneuvers are required, the Fast Marching Method is a good choice for Motion Planning. In Figure 3.31, we see the Fast Marching Method being used for motion planning in terrain using dilated obstacle maps. The example is from the experiments described in Section 5.4.

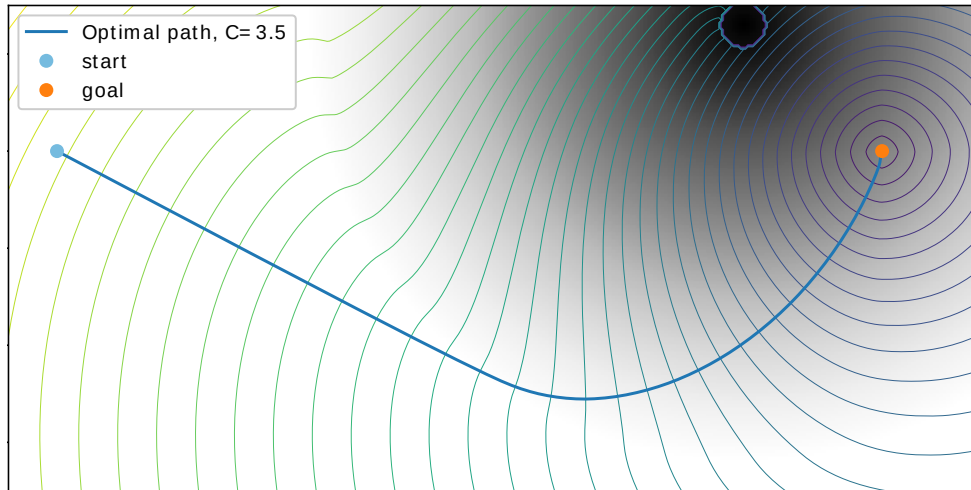


Figure 3.29 *Fast Marching applied to a cost map with source shown in orange. The gray levels indicate cost, with white being 1.0 and black being 3.5. The blue path is the optimal path with this cost map.*

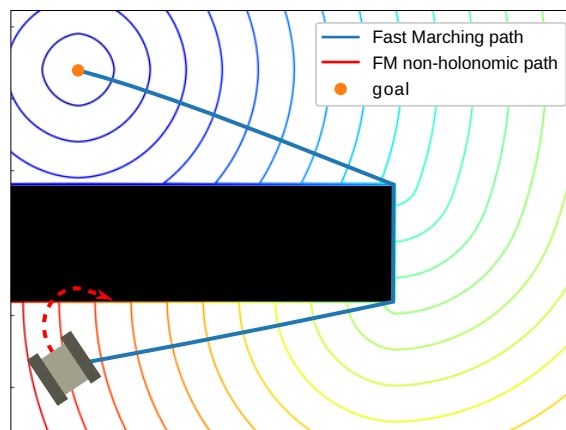


Figure 3.30 *The Fast Marching Method when the UGV is not aligned with the field. Trying to turn the UGV right towards the FM field would have caused a collision, while turning the other way would have been fine.*

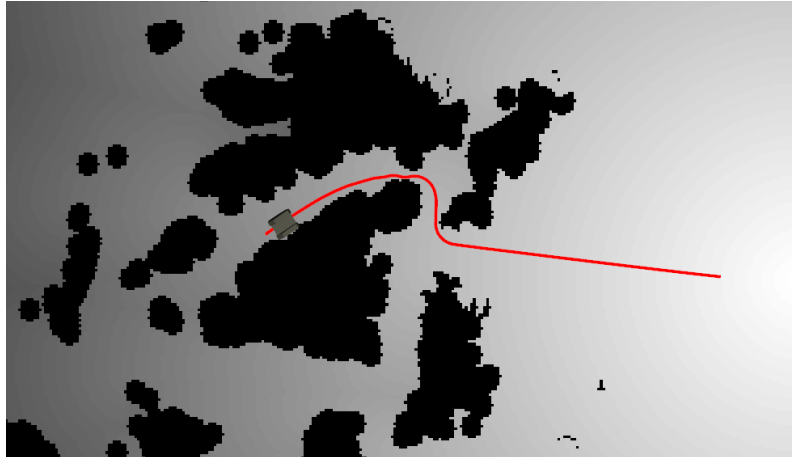


Figure 3.31 The Fast Marching Method being used for motion planning on a dilated obstacle map. Black are obstacles, the grayscale represents the Fast Marching Field with white being closest to the goal.

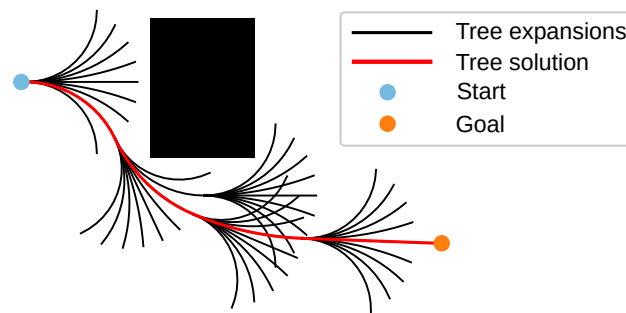


Figure 3.32 The tree search of Hybrid A*, using motion primitives from the start point. For each step, the best motion primitive is selected for further expansion of the tree until the goal is reached.

3.5.2.2 Planning algorithm – Hybrid A*

Hybrid A* [6] is a motion planning algorithm that uses a different take on motion planning than the Fast Marching Method. Whereas the Fast Marching planning is based around expansion from the goal, Hybrid A* is instead based on what movements are possible to perform starting from the UGV.

Hybrid A* is based on a set of *motion primitives*, short driving maneuvers based on different steering inputs. The set of motion primitives are then used to incrementally build a tree that expands from the starting pose of the UGV. For each step, each of the motion primitives is evaluated to see which is most promising by looking at the distance from the start to that point combined with the distance left to the goal. The overall most promising motion primitive is selected to expand the search tree further.

For a simple case with one obstacle, the search process can look like what we see in Figure 3.32. At the start, the set of 9 different motion primitives forms the basis of the tree. The best one is selected,

and a new set of motion primitives is added to the tree. The process repeats, and for each step, the best of all the primitives in the search tree is evaluated. Eventually, the tree gets close enough to the goal for a full solution to be created.

The tree search of Hybrid A* can grow exponentially, as for each new expansion, N different motion primitives are added. In our example in Figure 3.32, the branching factor $N = 9$. The key to avoiding exponential growth and long runtime is the measure of the “best” motion primitive. If the estimate for the remaining drive distance from any point to the goal is accurate, the algorithm will converge to a solution fast. In the original Hybrid A* method, they use a combination of optimal non-holonomic path that ignores obstacles and a grid based distance measure that accounts for obstacles but not non-holonomic constraints. There are a few more details that goes into the Hybrid A* algorithm, so for a full overview we refer to the original article[6].

Hybrid A* is a reasonably efficient algorithm in practice. It does not find the optimal path, but it finds a near-optimal path that is guaranteed to be drivable and collision-free, unlike our Fast Marching planner. The standard Hybrid A* path planner does not have support for using cost maps, which the Fast Marching Method does. We have previously implemented a pure Hybrid A* path planner for our UGVs, and we have also added support for cost maps to obtain the Traversability Hybrid A* method.

3.5.2.3 Planning Algorithm – Traversability Hybrid A*

The Traversability Hybrid A*(THA*)[51] is a method we have implemented that is capable of using traversability maps or cost maps. Instead of optimizing for the shortest path like Hybrid A* does, we instead want to minimize the cost along the paths in a cost map. The result is a path planning method that finds the most traversable paths instead of the shortest one.

We achieve this by switching out the grid distance measurement used in Hybrid A*, to a grid-based measurement that estimates the remaining cost. We use the Fast Marching Method for the cost calculation, as FMM when used with a cost-map creates a cost-weighted distance map to the selected goal position. We substitute the path length measure from the start point to any point in the search tree, by a measure of accumulated cost along those paths. With these changes, Traversability Hybrid A* can find near-optimal paths in a cost map.

We have tested Traversability Hybrid A* in real-life experiments with a UGV[51]. With THA*, we were able to make our UGV choose different paths than the ones selected using the regular Hybrid A* algorithm. We can see the difference in Figure 3.27 where Hybrid A* ignored a high-cost area, whereas THA* preferred lower cost areas.

THA* does come with a calculation time penalty compared to using only the Fast Marching Method, as FMM is run as the first step for both path planning methods, and the tree-search will add to the calculation time for THA*. For this reason, FMM can work as well as the THA* algorithm in practice in some situations, for example when fast responses are more important than guaranteed collision free paths.

3.5.3 Motion planning in dynamic and partially known terrain

Traditional motion planning algorithms typically operate under two key assumptions: that the environment is fully known, and that the goal lies within this known space. However, when applied to UGVs navigating real-world terrain, these assumptions often do not hold.

UGV sensors have limited range, which means the world map generated through perception is inherently constrained. Terrain features, obstacles, and occlusions can obscure parts of the environment, making it more likely that new elements will be detected as the UGV approaches them. Consequently, the world map is continuously updated as the UGV moves.

Moreover, the environment itself may change. Dynamic objects can alter the perceived scene, and perception is not perfect – object classifications may shift over time as more data is gathered. This is especially true for areas near the UGV, where repeated observations improve accuracy. These factors contribute to a constantly evolving map, requiring the motion planner to adapt in real time.

Another challenge is that the goal location may lie outside the current perceptual range. Even when the goal is within view, the reliability of the map decreases with distance from the UGV. As a result, it is unrealistic to expect a single trajectory to remain valid throughout the entire journey. Instead, planning should prioritize the immediate segment of the path, with the understanding that the trajectory will need to be revised as new information becomes available.

3.5.3.1 *Receding horizon motion planning*

To handle dynamic environments and goals located beyond the current perceptual range, the motion planning process must be periodically repeated. Rather than committing to a single trajectory all the way to the goal, the UGV follows only the most recent plan, updating it as new information becomes available. This approach ensures that outdated or invalid trajectory segments are discarded before they can lead to poor decisions.

Importantly, the motion planner does not need to compute a full path to the goal in each iteration. Instead, it plans a trajectory up to a fixed distance ahead – referred to as the planning horizon. As the UGV moves forward, the horizon advances with it, gradually bringing the goal within range. This iterative process is known as Receding Horizon Planning, illustrated in Figure 3.33.

In Figure 3.33a, the UGV plans a path to the horizon, unaware of an obstacle beyond it. After moving forward along the initial path (Figure 3.33b), the obstacle enters the perceptual range, and the motion planner incorporates it into the next trajectory. This cycle continues until the goal is reached.

When the goal lies outside the observable map, a temporary goal can be selected within the perceptual range – typically at the edge of the map and in the direction of the true goal. As the UGV progresses, this temporary goal is updated, gradually guiding the UGV toward the final destination.

However, reaching the goal is not guaranteed. Untraversable terrain beyond the perceptual range may block progress, and such limitations stem from the fundamental challenge of incomplete information. Motion planning alone cannot resolve this; instead, care must be taken to ensure that the chosen route minimizes the risk of the UGV becoming trapped or unable to proceed.

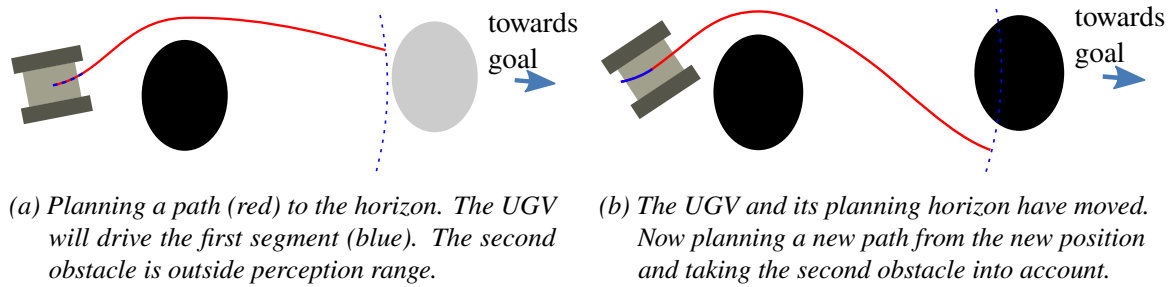


Figure 3.33 Receding horizon motion planning. The UGV repeatedly plans paths to a planning horizon. The UGV will drive the first segment of the path, discard the rest, and plan a new path from its new position and using updated map information.

3.5.3.2 Route following

A route consists of a sequence of waypoints that the UGV is expected to follow. These waypoints may originate from various sources: manually placed by an operator in a map interface, generated by an automated route planner, or recorded from a previously driven path.

At its simplest, route following can be implemented by sequentially setting each waypoint as a goal and navigating toward it. However, to ensure robust and efficient navigation, several additional considerations must be taken into account.

The strategy for following a route should ideally depend on the origin and accuracy of the waypoints:

Low-accuracy or manually created routes Manually placed waypoints, such as those created in a map interface, are often imprecise. In these cases, the motion planner should treat the waypoints as general guidance rather than exact targets. Strictly following such waypoints could lead to inefficient or even unsafe paths, especially if the map contains offsets or GNSS inaccuracies.

High-Accuracy Routes If the route is derived from precise map data (e.g., road networks or previously recorded runs), the UGV should aim to follow the waypoints closely. In such cases, tight adherence to the path may be necessary to ensure repeatability or compliance with known constraints. However, map inaccuracies or GNSS errors may still exist, so the motion planner should maintain some flexibility. Even without such errors, the motion planner should retain flexibility to navigate around unforeseen obstacles or terrain features.

In practice, our motion planner adopts a flexible approach: it does not place heavy emphasis on the exact placement of waypoints, regardless of their origin. We currently do not differentiate between route types or waypoint sources; we operate under the assumption that waypoints may be imprecisely placed. This conservative assumption allows the planner to treat waypoints as general guidance rather than strict targets, which is particularly useful in environments with low localization accuracy or where map data may be offset. By not relying heavily on exact waypoint placement, the system becomes more robust to errors and better suited for a wide range of operational scenarios.

To implement this flexible approach, we use a long look-ahead strategy when selecting the current waypoint to track. Instead of targeting the nearest waypoint, the planner identifies a waypoint further

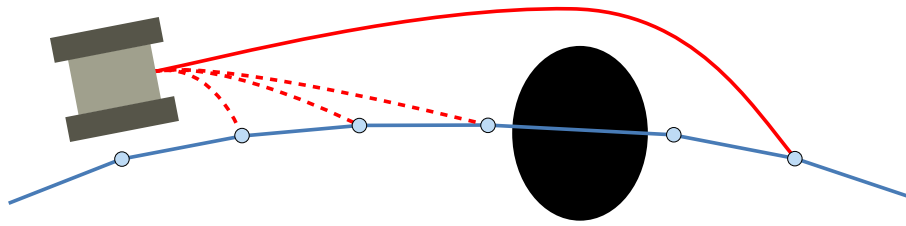


Figure 3.34 Route following with far look-ahead. By focusing on a waypoint far ahead, the motion planner can find smoother paths instead of following the closest waypoints, which would cause larger obstacle avoidance maneuvers later down the line.

ahead, typically near the edge of the perceptual map, and uses it as the active goal. All intermediate waypoints between the UGV and the selected target are effectively ignored. This method ensures that the UGV maintains forward momentum and avoids unnecessary corrections that could result from closely spaced or poorly placed waypoints. It also allows the planner to focus on navigating toward meaningful progress along the route, rather than micromanaging each step. The concept is illustrated in Figure 3.34, where the UGV targets a distant waypoint, enabling better obstacle avoidance than if the closer waypoints were targeted.

As a result of this strategy, the motion planner naturally gains the freedom to react to the environment. It can dynamically adjust its path to avoid obstacles, reroute around untraversable terrain, and respond to changes in perception without being constrained by rigid waypoint adherence. The long look-ahead approach provides the necessary buffer for safe and adaptive planning, ensuring that the UGV can continue progressing toward the goal even in uncertain or changing conditions. Another example can be seen in Figure 3.35, where the UGV is following a vehicle trail with waypoints that are not perfectly aligned with the trail. By using a long look-ahead, the motion planner can prioritize following the perceived trail while still roughly following along the route.

The potential downside to this approach is in the cases where the waypoints are accurately placed and represent an optimal path. This could, for example, be the case when following a route driven before, or where the route is created from precise road location data from accurate maps. In such scenarios, the most efficient strategy would be to follow the waypoints closely instead of searching for a feasible path. For example, if the route is placed at the center of a vehicle trail, the optimal path would be to follow the trail directly, whereas if the UGV only detects obstacles, it might deviate unnecessarily from the trail. However, we believe that the increased robustness and flexibility provided by the long look-ahead strategy outweighs the potential efficiency loss in these cases.

3.5.4 Other work on motion planning

We have also experimented with other motion planning algorithms and techniques in the past, where the work primarily was conducted by master students. These works have not been tested live on Tor, but have instead been run on recorded terrain data. These works are:

- Robust motion planning with Rapidly-exploring random trees (RRT) based on Funnel Libraries. The method uses pre-computed motion primitives (funnels) that are guaranteed to

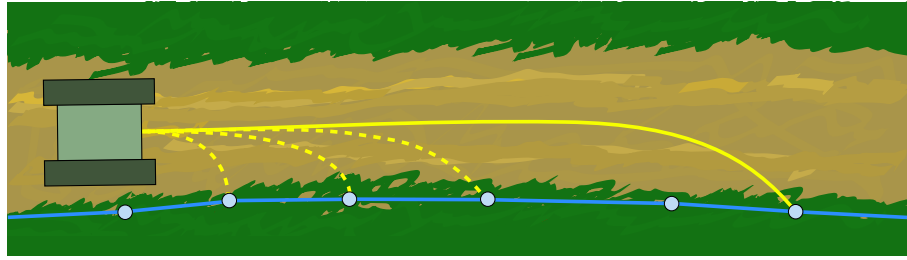


Figure 3.35 Route following along a vehicle trail, with misplaced waypoints. In this case, it is better to follow the perceived trail instead of strictly following the route. By using a large look-ahead, the motion planner has enough space to follow the trail for a while before turning towards the waypoint.

be safe, and then use RRT to connect these funnels to create a full path. This means that the motion planner can create paths that are guaranteed to be safe in the presence of uncertainties, even in complex environments. This work is described in detail in 2019[34] and 2022[35].

- Path planning with radial constrained A*. The method employs an A* grid search with a turn-restriction that makes it more suitable for use with non-holonomic vehicles. This method can help with the type of problems illustrated in Figure 3.30 where a grid-search yields an invalid solution due to obstacles. The work can be found in [5].
- Work on using variants of RRT for motion planning in rough terrain, and comparing and combining RRT with the Fast Marching method. The work can be found in [3].
- Extending the Fast Marching method and the FM² methods to work better as a continuous process local motion planner. This includes incorporating techniques for integrating non-holonomic constraints as well as using previous planning results in the next planning steps to increase the consistency of the planned paths over time. The work is found in [54].

3.6 Trajectory tracker

The *Trajectory Tracker* is the autonomy module that takes a planned trajectory and guides the UGV along it by generating drive commands. For our UGV, this means calculating linear velocity v and rotational velocity ω that are executed by the UGV (see Equation 2.1). The trajectory tracker problem is illustrated in Figure 3.36 where we see a UGV that should follow the given trajectory. The objective is then to find appropriate velocities that will guide the UGV along the trajectory, for example, as illustrated by the green paths.

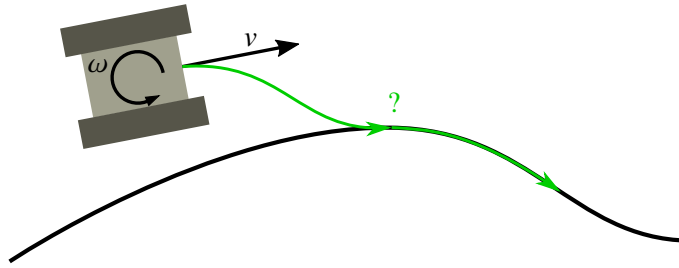


Figure 3.36 The trajectory tracking problem: Find linear velocity v and angular velocity ω to guide the UGV onto and along a trajectory.

There are two related terms that describe trajectory following that is often used interchangeably, but have slightly different meanings: *path following* and *trajectory tracking*. A path is a geometric description of a route, for example, represented by a sequence of points or poses, or by a continuous function. A trajectory is a path with a time component, which means that the UGV should be at a certain point on the path at a certain time. A trajectory can therefore have information about what speed the UGV should have at different points on the path. *Path following* is about following a geometric path as closely as possible, while *trajectory tracking* is similar but also cares about the UGV being at specific points on the trajectory at given times. Path following can be seen as a special case of trajectory tracking where the time component is ignored, and is an easier problem to solve.

Trajectory tracking is necessary when following a moving target, such as another vehicle, to keep a fixed distance to the target. It can also be beneficial to reduce speed at sharp turns, and increase speed on straight segments. This, of course, requires that the motion planner creates trajectories with a speed profile. Our trajectory tracker has support for following both paths and trajectories. We have implemented several different methods for trajectory tracking, with different strengths and weaknesses, and we have the option of switching between them at run-time. However, some of the methods do not take the time component into account, and are therefore more accurately described as path following methods. In many cases, we do not require following specific speeds along the path, for example, when navigating in terrain where we want a slow and steady speed. In these cases, we configure the trajectory tracker to ignore the time component and follow the geometric path at a constant speed. For consistency, we refer to the module as a trajectory tracker, and call the process trajectory tracking and the software module *Trajectory Tracker*.

3.6.1 Trajectory tracking principles

The trajectory tracking methods work by looking at where the UGV is relative to the trajectory, and then by balancing mainly two things:

- To move the UGV forward along the trajectory and track its movement.
- To move the UGV towards the trajectory if the UGV deviates from it.

There are many different ways to implement trajectory tracking algorithms, and they have different strengths and weaknesses. Fast reactions to bring the UGV back to the trajectory will make it follow the trajectory closely, but it may also lead to twitchy and aggressive steering. Focusing more on following the trajectory will give a smoother steering, but it may also lead to larger deviations from the trajectory.

Perhaps the easiest way of achieving trajectory following is with the simple path following algorithm pure pursuit (illustrated in Figure 3.37). It works by looking at a point some distance ahead on the trajectory, and then steering the UGV towards that point. As the UGV moves, the target point on the trajectory also moves forward. It is a simple and effective algorithm, but it has some limitations. For example, it will never follow a trajectory exactly and will cut corners. This is because it always steers towards a point ahead on the trajectory, but there is no mechanism that makes the UGV converge to the trajectory. Thus, when the trajectory curves, the UGV will steer away from the trajectory. How far ahead to look is a tunable parameter. Setting it too short will make the UGV react quickly, but the steering will be jerky and can oscillate. Setting it too long will make the UGV follow the trajectory more smoothly, but it will also lead to larger deviations from the trajectory. We have previously experimented with using the pure pursuit method for path following as a baseline method for experiments we conducted with following a vehicle [50].

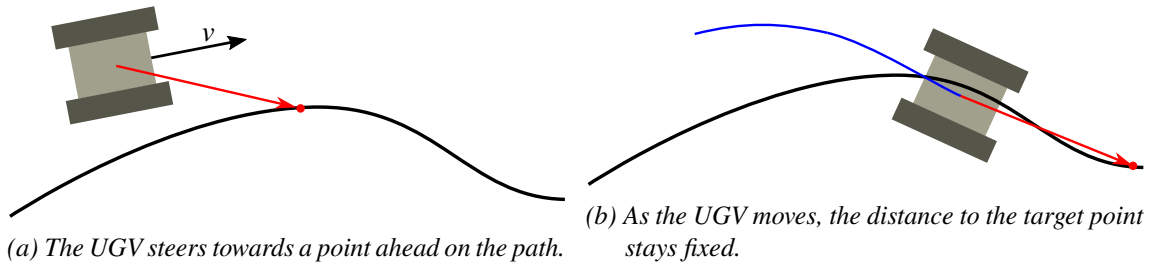


Figure 3.37 The pure pursuit method for path following.

3.6.2 Improved trajectory tracking: the Morin controller

To improve path following, we can add mechanisms that make the UGV converge to the trajectory, and maintain it once reached. To do so, we define a reference vehicle that moves along the trajectory, as shown in Figure 3.38. The reference vehicle uses a kinematic unicycle model, which means its dynamics are described by the equations:

$$\dot{x} = v \cos(\theta) \quad (3.11)$$

$$\dot{y} = v \sin(\theta) \quad (3.12)$$

$$\dot{\theta} = \omega \quad (3.13)$$

Using the reference vehicle, we can define errors (longitudinal, lateral and heading) between the UGV and the reference vehicle, as seen in Figure 3.38. Then, we can design a controller that uses

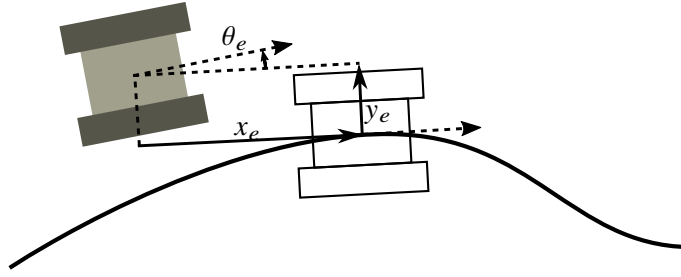


Figure 3.38 The Morin controller for trajectory tracking. The UGV has a longitudinal error x_e , lateral error y_e and a heading error θ_e relative to a reference on the trajectory. The controller uses these errors to calculate control inputs v and ω .

these errors to calculate the control inputs v and ω for the UGV that will make it converge to the reference vehicle and follow it. We can have the reference vehicle move with a velocity profile defined by the trajectory, creating a true trajectory tracker and not only a path follower. This is the principle behind the Morin controller [32], which we have implemented in our trajectory tracker.

The Morin controller works with continuous trajectories instead of the pose sequences that the motion planner creates. The Morin controller's first step is therefore to create a continuous trajectory from the waypoints using third-order polynomials. After the trajectory is obtained, the next step is to control the UGV to follow the trajectory. This is divided into two separate problems: one for controlling the UGV onto the trajectory and one for controlling the forward velocity. For controlling the UGV onto the trajectory, the Morin controller uses lateral error and heading error in combination. For controlling the forward velocity, the controller uses the longitudinal error. If we do not have the need to follow a velocity profile along the trajectory, we can set the forward velocity to a constant value to effectively make it a path follower.

The Morin controller is based on a few assumptions [32]:

- The UGV can be modeled as a unicycle.
- The UGV has perfect friction with the ground, meaning no slippage occurs.
- The UGV operates on a flat, even surface.

The Themis UGV is a tracked vehicle that behaves similar to a unicycle model with an Instantaneous Center of Rotation (ICR) located in the center of the vehicle. Moving on uneven terrain and with non-constant friction means that the ICR can shift, and the vehicle will not behave exactly as the unicycle model describes. We therefore cannot expect perfect performance of the controller, especially in a terrain environment. Even though it is based on an ideal model, the Morin controller has shown itself to be robust and reliable in most off-road conditions as long as shifts in the vehicle's rotation center is minimal [26, 53]. For this reason, the Morin controller is our preferred trajectory tracking method.

3.6.3 Other trajectory tracking methods

Following a planned trajectory is significantly more challenging in rough terrain compared to following a planned path on an even flat surface, like a road. In rough terrain, the trajectory tracker

must handle the slopes, roughness, and slippage of the tracks that the UGV experiences. We have implemented the trajectory tracker such that it is possible to switch between different trajectory tracking methods and configurations at run-time. We use the ability to switch between different methods, for instance, when we switch between path following and tracking a moving target.

Currently, we have implemented and tested three trajectory tracking algorithms in addition to the Morin Controller. These can all be categorized as so-called kinematic models, which has the advantage that they reasonably accurately describe vehicle movement without being subjected to terrain-dependent parameters. The three additional algorithms we have implemented are:

HBZ controller The Huskić-Buck-Zell (HBZ) controller is based on an Instantaneous Center of Rotation (ICR) kinetic model, and the name HBZ is an acronym of the original authors' surnames [17]. It uses an Extended Kalman Filter (EKF) [38] to estimate ICR locations in real-time, and this makes it able to better handle off-road effects that cause shifts in ICR locations [52, 53].

ADRC The Active Disturbance Rejection Controller (ADRC), which is based on an extended unicycle model [43], where disturbances due to slippage are modeled in the vehicle kinematics. These disturbances are estimated using an EKF and a simple Proportional Derivative (PD) controller is used for path following. Similar to the HBZ controller, the ADRC controller has been shown to have better performance than the Morin controller in off-road conditions where the performance of Morin controller is degraded [52, 53].

MPC The Model Predictive Controller (MPC) is an optimization based controller, which utilizes a mathematical model of the UGV to predict future states over a predefined prediction horizon. The predictions are generated at discrete time steps by a numerical solver, which finds the control inputs at each time step that will result in the future states best corresponding with a desired control goal and behavior. Any model that adequately describes the UGV can be utilized, such as the ideal unicycle model. But by implementing a highly accurate model such as one that includes slippages due to slopes and terrain with low frictions, the MPC may account for disturbances due to such factors when generating its control outputs, so that more accurate trajectory tracking is achieved during difficult driving conditions [15]. This controller is currently only tested in simulations, not on the physical UGV.

3.6.4 Trajectory tracking for following a moving target

The trajectory tracking method for following a lead vehicle builds upon the Morin controller described above. It consists mainly of two parts: First, it creates a continuous path from the following UGV to the lead vehicle. Second, it implements a velocity controller for holding the appropriate distance to the lead vehicle. The latter relies on measuring the position and heading of the lead vehicle and estimating the relative velocity between the vehicles. The implementation details are given in Appendix B, and the controller is briefly described here.

The path from the following UGV to the lead vehicle is created using third-order polynomials, the same method as in the Morin controller. This means that the path between the vehicles is smooth, and that the relative heading difference between the vehicles is taken into account. This causes the following UGV to not cut corners, as it will not drive directly towards the lead vehicle, but will drive in an arc to match the lead vehicle's heading in the end. To ensure that the following UGV does not

make too sharp turns at high speeds, we have modified the calculation of the arc curvature. At lower speeds, the behavior is the same as in the Morin controller, but at higher speeds, the maximum curvature is limited based on the UGV's speed. This modification means that the following UGV will take wider arcs at higher speeds, which improves stability and reduces the risk of slipping. It does, however, mean that the following UGV will not be able to exactly match the lead vehicle's path in sharp turns at high speeds. In practice, we have found that the modified arcs do not cause any major issues and the overall ability to track a lead vehicle is satisfactory, even at higher speeds.

For controlling the velocity of the UGV, we modified the forward velocity controller part of the Morin controller. The operator sets a desired distance to the lead vehicle, which means the controller can calculate a distance error from the actual distance to the lead vehicle. Instead of using a constant forward velocity, the controller calculates the forward velocity based on a combination of the estimated velocity of the lead vehicle and using proportional control to minimize the distance error. In addition, we implemented a safety mechanism so that the UGV speed was scaled down when close to the desired distance to the lead vehicle, and that the UGV stopped if the following vehicle was closer to the lead vehicle than the desired distance. We demonstrated the platooning controller at LandX21, which will be presented in Section 4.1, and we have it on a tractor as well [1].

4 The LandX demonstrations

The UGV group conducted several larger experiments from 2020 to 2022. The primary goal has been to demonstrate autonomous driving capabilities in military scenarios in terrain. At the same time, we also had a secondary goal of demonstrating the utility that UGVs can provide, for example, as a mobile observation platform.

Starting in the fall of 2020, FFI has hosted LandX, a yearly collaborative experiment and demonstration where different research projects at FFI participate to experiment with the technology they research [24, 25, 29, 4]. The UGV project at FFI has participated in all the LandX events, demonstrating how UGVs provide operational benefits on the battlefield. Not all of these, however, involved autonomy for UGVs and have instead explored other aspects of UGV operation involving teleoperation. At LandX21, we participated in initial testing of parts of the autonomy described in this report. At LandX22, we experimented with the full autonomy stack described for the first time at LandX, and details about these experiments are presented in the following sections. After LandX22, the autonomy was also tested more rigorously in a separate experiment in more diverse terrain, presented in Chapter 5.

4.1 LandX21

At LandX21, we used Tor to demonstrate a partially unmanned logistics concept. Tor had the capability to follow a manned lead vehicle automatically. It would start by following the leader, and after receiving a signal it would then return to the starting point by on its own along the path it already had driven. During the demonstration, Tor was loaded with cargo and followed a lead vehicle to an offloading point, where soldiers from the lead vehicle would unload the cargo. After unloading, Tor was able to return to the starting point.

LandX21 was a step forward for the autonomy development on Tor, but Tor could not truly be called autonomous. To run the logistics scenario, Tor detected and tracked the lead vehicle using a front-mounted camera. The lead vehicle had a 50×50 cm AprilTag plate mounted at the back, which made the tracking software able to determine the distance and orientation of the lead vehicle at any time as long as the AprilTag was visible. The method used was the one presented in Section 3.4.4. With a known position of the lead vehicle, Tor could use the rest of its autonomy stack to follow it. In Figure 4.1, Tor can be seen following the lead vehicle during testing. The vehicle following was based on technology FFI used in collaboration with Yeti Move AS in 2021 in experiments with an unmanned logistics column [1].

Returning to the start location was achieved by following the recorded GNSS track directly back to start. The vehicle following method was the one presented in Section 3.6.4. Except for the lead vehicle tracking, the driving was performed without using perception. Thus, Tor was unable to react to the environment itself, for example, by detecting obstacles and stopping or avoiding them. The driving was therefore automatic instead of autonomous at LandX21. Nevertheless, this basic perception capability was an important stepping stone in developing true autonomy. In the future, the lead vehicle tracking could be replaced by a more general object tracking system, such as the one we experimented with in [50]. More details about LandX21 in general can be found in [25].



Figure 4.1 LandX21: Tor following a lead vehicle as it did during the logistics demonstration. The AprilTag on the back of the lead vehicle is what makes Tor able to track the vehicle using a camera.

4.2 LandX22

For LandX22, both of the UGVs Tor and Siv were used in a tactical scenario together. The overall mission was to advance through a field, detect and defeat any enemies before manned vehicles would enter the area. Tor was, for the first time at LandX, autonomous and ran the autonomy stack described in this report in Chapter 3. Siv was teleoperated and was equipped with a remote weapon station (RWS), and the same operator could control both Siv and the RWS.

The mission that the UGV participated in can be seen in Figure 4.2. Both Tor and Siv started at the same place out of sight. Tor was given a route to follow and was set in action. Tor moved autonomously and followed the route to the specified goal point. The Tor operator raised the mast with the observation payload, and started scanning for enemies (Figure 4.2a). The operator detected an enemy, and signaled to the Siv operator to drive Siv forward and to take out the threat. While Tor maintained observation of the enemy, Siv was being driven forward teleoperated (Figure 4.2b).

The Siv operator eliminated the enemy threat using the Remote Weapon Station on Siv. Tor was given a new route to follow, along a vehicle trail and through a forest section. Tor drove off autonomously, and the operator could use the sensor payload to observe while Tor was driving. The UGV part of the scenario ended when Tor entered the forest section and moved out of sight.

4.2.1 Mission execution

Tor was driving autonomously for the whole demonstration at LandX22. The overall mission was divided into two separate tasks that Tor executed: first, drive to the first observation point and then drive to the second observation point. We defined the tasks using waypoints that we manually placed in the ground control station Norne, presented in Section 3.2.2. Figure 4.3a shows the approximate location of the waypoints the operator selected. They used two waypoints for the first



(a) Tor moves into an observation position. The operator raises its mast and starts observing the sector. (b) Tor operator gets a positive identification on an enemy. Siv is being driven into a firing position. (c) Siv operator eliminates enemy with its RWS. Tor advances to a new observation position.

Figure 4.2 LandX22: Tor, driving autonomously and equipped with an observation sensor, and Siv, being teleoperated and equipped with a remote weapon station working together in a tactical scenario.

task or leg of the mission, and three waypoints for the second leg. The rest of the navigation was left to the autonomy stack on Tor.

The environment for the experiment is a typical example of *Route C – Wide and open vehicle trail* as we defined in Appendix A. There were wide open spaces and few obstacles, making the route easily traversable as long as the task waypoints are placed along the drivable vehicle trail.

We selected the waypoints manually, and we unintentionally placed a few of the waypoints off to the side of the vehicle trail we intended to follow. This occurred along the stretch seen in Figure 4.3b, where some waypoints were off to the left side of the trail. During execution, we observed that instead of driving directly towards these waypoints, Tor actively avoided the bushes, which were classified as obstacles and thereby stayed on the vehicle trail.

The perception system on Tor had issues with time synchronization of the sensors, which caused some problems. The issues manifested as “ghost obstacles” that Tor was trying to avoid, but in most of the runs these avoidance actions did not have a major effect on the overall driving. The route that Tor completed is shown in Figure 4.3. The demonstration was a success, and we showed that Tor was capable of executing route following missions in the Type C environment, despite the sensor issues. For more details about LandX22, we refer to [29].



(a) Location of the selected waypoints(b) The long stretch towards the second observation point. The picture and Tor's driven path.

Figure 4.3 Tor's mission at LandX22.

5 Long-range autonomous drive experiment

At the LandX22 experiment, we demonstrated basic autonomous driving in a relatively simple environment (see Section 4.2). After LandX22, we wanted to do more rigorous testing of the autonomy in more diverse environments. We were also able to improve the issues with the time synchronization that affected the performance in the LandX22 experiment.

Our overall goal for this experiment was to run autonomously over longer distances in varied terrain to evaluate performance of the autonomy in diverse terrain types. We aimed to evaluate the limits of the autonomy and see where our system would fail. Based on our experiences with the autonomy stack, we knew that the more challenging routes from Section 1.1.1 would be the least likely to succeed. We instead started with the easier routes, and wanted to progress as far as the autonomy allowed, and to see for which routes our autonomy would be sufficient, and in which situations it would fail.

We ended up doing four experiments, which matched well with the predefined routes.

Experiment 1 – Open and flat terrain An open field with a few large trees in one end and some areas with low vegetation.

Experiment 2 – Wide and open vehicle trail A mix of open areas and wide vehicle trails with a sparse forest and vegetation nearby.

Experiment 3 – Dirt road A dirt road in a flat area. The surrounding area has sparse vegetation, but it is drivable.

Experiment 4 – Varied terrain An experiment in varied terrain types, but mostly consisting of vehicle trails of various sizes and open forest.

Tor ran the autonomy stack with the following configuration:

Mission input Route from manually placed waypoints in Norne GCS (Section 3.2.2).

Perception Producing obstacle maps based on lidar (Section 3.4).

Motion planner Trajectory planning based on the Fast Marching method, with obstacle maps and the route from Norne as inputs (Section 3.5.2.1).

Trajectory tracker Path following based on the Morin method (Section 3.6.2).

The UGV operator created missions for Tor in Norne by manually placing waypoints. The waypoints were placed relatively long distances away from each other, up to 300 m apart for the longest segments. Tor used the autonomy stack to navigate, trying to drive to the mission waypoints, but with no other prior information about the environment, map or surroundings. We deliberately avoided being overly precise when placing waypoints so they would not be at the optimal locations. This is because we believed we would learn more about the true performance of the autonomy system if exposed to a few potentially suboptimally placed waypoints. If our only goal was to make the system drive as much by itself without interventions, we could most likely achieve that by being as careful with placing waypoints as possible. Such careful waypoint placement could be placing waypoints through open areas or along visible vehicle trails in aerial photos, and that the waypoints were placed close enough to follow the rough shape of the trails.

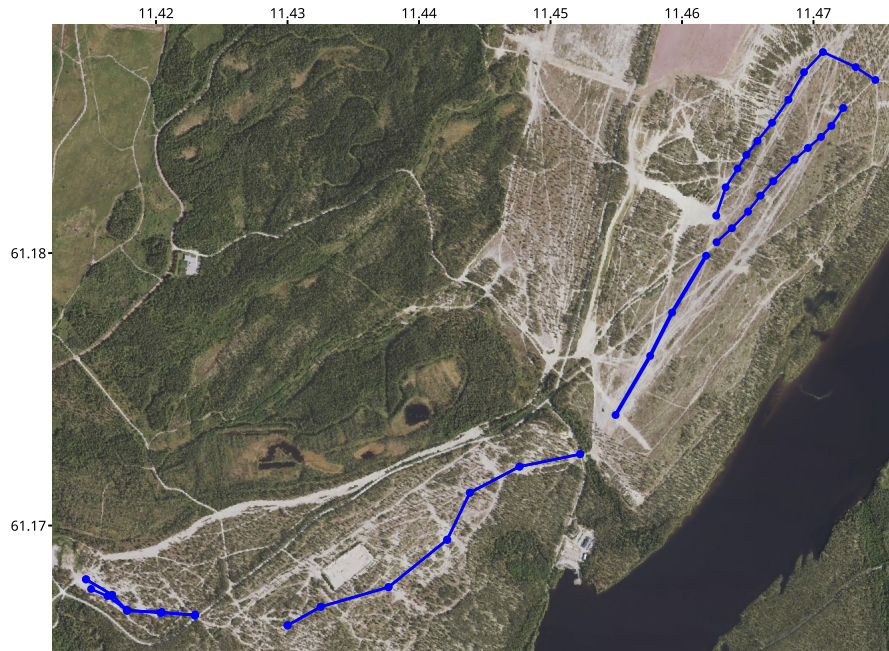


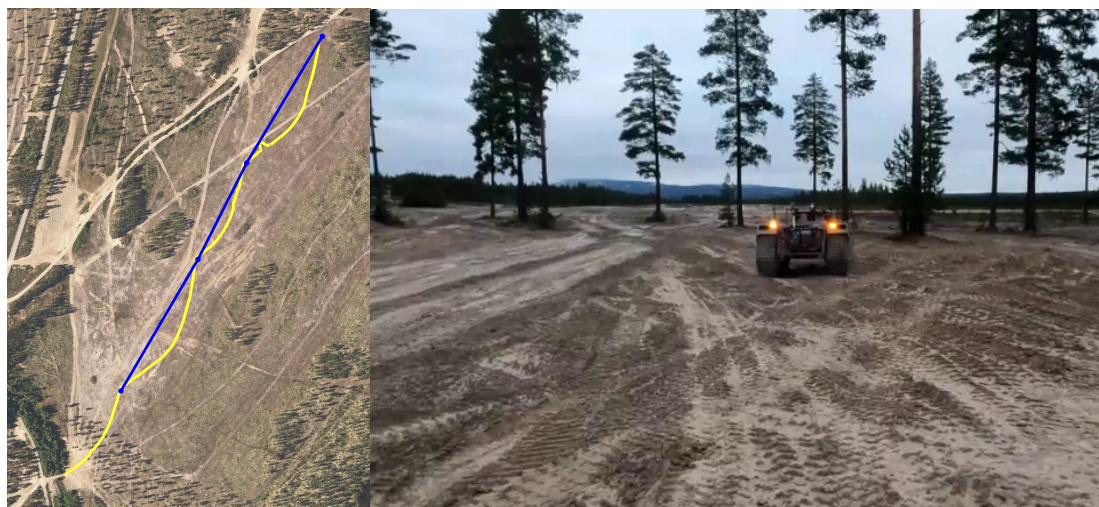
Figure 5.1 The four experiments of the long-range experiment. The routes are the mission waypoints that we manually selected as input to the autonomy stack.

The location of the experiments we did can be seen in Figure 5.1. The routes shown are the actual routes we used as inputs to the Autonomy stack. The start position of Tor is not shown here, the routes start at the first mission waypoint, and Tor would drive to the first waypoint from its start position.

The experiments were carried out at walking speeds, as the operators followed Tor by walking. In most of the experiments, we ran the system at 2 m/s. In experiment 4, we reduced the speed to 1.5 m/s in order for the operator to better keep up while monitoring Tor, as well as to reduce the effect of time synchronization issues that we experienced.

5.1 Experiment 1 – Open and flat terrain

The goal of the first experiment was to test how the autonomy of Tor performs in the simplest of the defined routes, namely an open field with large and few obstacles. The field we selected for the experiment was a near perfect example of such an environment, being flat, mostly without vegetation, and the trees that were there were large and easily detectable by lidar. However, near the end there were some patches of vegetation, which means that this experiment did not match the idealized scenario of route A, defined in Appendix A.



(a) Input route (blue) and driven path (yellow), starting from the south.

(b) Tor at the beginning of experiment 1.

Figure 5.2 Experiment 1 – results from Tor traversing the open field with sparse obstacles.

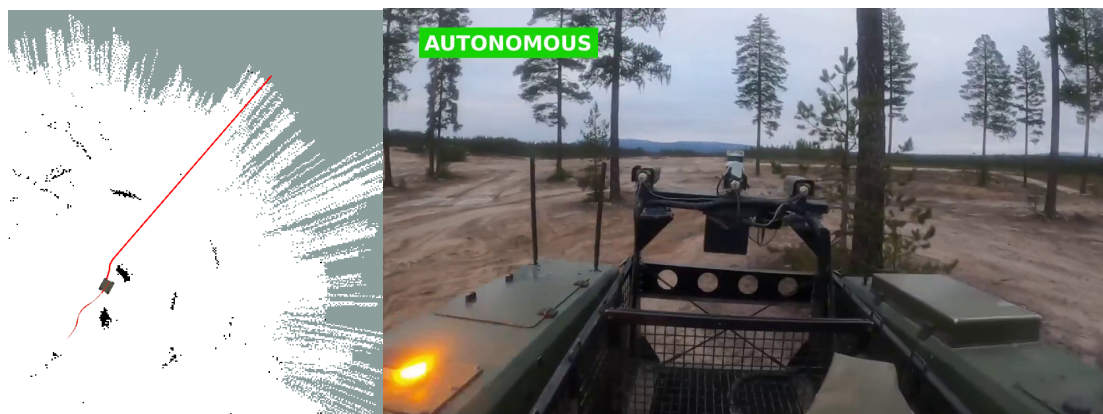
5.1.1 Experiment 1 – Results

In Figure 5.2, we see results from the run, with Figure 5.2a showing both the input route consisting of four waypoints in blue, and the driven path in yellow. The spacing between the input waypoints is approximately 250 m. The first thing to note is that the driven path repeatedly curves away from the input route, before converging back to the waypoints. This behavior is the result of how the waypoints that were beyond the range of local obstacle map were handled at the time. The motion planner would select a temporary goal within the local obstacle map based on the current waypoint, and the logic in this process had a logical error. In Experiment 1, this peculiar behavior did not negatively affect the overall performance of the system, besides causing Tor to drive slightly longer than necessary.

The obstacle detection and motion planning for avoiding the trees worked without issue in Experiment 1, and Tor ended up actively avoiding a couple of trees. In Figure 5.3, Tor can be seen right before passing a tree. Towards the end of the run, Tor ended up getting stuck, and the operator had to take manual control. This happened because there was a patch of low vegetation in the path towards the final waypoint. The obstacle detection was initially not classifying the vegetation as obstacles, and Tor kept driving towards it. When Tor got closer, the obstacle detection was able to detect the vegetation, and thus causing Tor to become stuck in perceived obstacles, without maneuvering space for escaping. After manually driving Tor out of the vegetation patch, it was able to continue to the goal.

5.1.2 Experiment 1 – Conclusion

Tor drove a total distance of 1006 m and had one manual intervention in Experiment 1. The autonomy on Tor performed well as long as the environment consisted of an open field with large



(a) Obstacle map with UGV position, planned trajectory in red and the driven vehicle trail.

(b) Simultaneous view from Tor.

Figure 5.3 Snapshot from Experiment 1 while Tor drives past a tree.

obstacles as specified in the defined Route A. Experiment 1 served as a good verification that the basic autonomous capabilities worked well in a simple environment. That said, sparse and low vegetation proved to be a challenge for the obstacle detection perception system. In reality, Tor would not have had any problems with driving straight through the vegetation, but the challenge is to discern between the light vegetation that can be driven through and the heavier vegetation that should be avoided.

5.2 Experiment 2 – Wide and open vehicle trail

Experiment 2 was a more challenging experiment than Experiment 1, and consisted of different types of terrain:

- It started in an open field, but with rougher terrain than for Experiment 1.
- It continued on wide vehicle trails, with sparse forest next to it.
- It ended with a section through a medium density forest, but with flat terrain and enough space for Tor to maneuver.

The input route and the resulting driven path for Experiment 2 can be seen in Figure 5.4. The terrain featured small inclines and was a little rougher than the flat field from Experiment 1, but not to a level that would affect the performance of Tor. The main difference from Experiment 1 was that this experiment had more vegetation, both large pine trees and smaller bushes. The vegetation grew next to the vehicle trails, so if Tor had stayed on the trail, the vegetation would not have affected the run much. The final section is the exception, as this was included specifically to test the performance of driving through a denser forest that would challenge Tor's maneuvering.



(a) Input route (blue) and driven path (yellow), starting from the south. (b) Tor during experiment 2 on the vehicle trail and hugging the tree line.

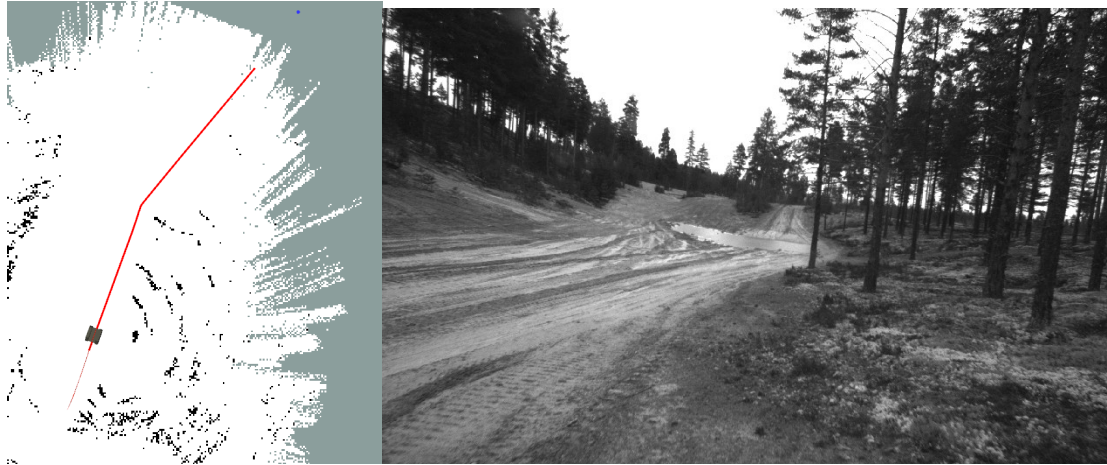
Figure 5.4 Experiment 2 – Tor traversing a course consisting of an open field, wide vehicle trails and a small forest section.

5.2.1 Experiment 2 – Results

The first section through the open terrain went without issues. This section had the roughest terrain of the experiment and caused Tor to shake, but not enough for it to affect the autonomy performance. The first section also did not feature obstacles, which meant Tor could drive straight towards the waypoints across the field.

After the open field, Tor entered an area with more vegetation between the vehicle trails themselves. The input route had waypoints placed in a manner that made Tor drive straight into areas with low vegetation. This caused a situation similar to the one from Experiment 1, where Tor got stuck in a perceived dead end and the operator had to drive Tor manually out from the vegetation. The behavior was repeated two more times with manual interventions before arriving at a larger vehicle trail where the low vegetation ended. This vehicle trail can be seen in Figure 5.4b.

Following the vehicle trail went mostly without issues. Tor ended up following the tree line tightly on the right side of the trail while driving. This behavior was caused by the waypoints being consistently offset to the right side of the vehicle trail. Instead of steering directly towards the waypoints and into the forest, Tor instead avoided the obstacles and stayed at the boundary of the vehicle trail. An example can be seen in Figure 5.5 where we see Tor navigating towards the blue waypoint, and actively plans a trajectory along the tree line to avoid the trees instead of directly towards the goal. The behavior works as intended, and demonstrates that despite inaccuracies in the route, Tor can still navigate based on its perceived surroundings. We also observed that when Tor approached a waypoint that was placed inside a patch of vegetated area, which Tor could not reach. Tor drove past the waypoint, and because it was configured with a large acceptance radius for waypoints, it was deemed close enough. Tor then switched to tracking the next waypoint on the route. The situation is shown in Figure 5.6. This flexibility is important in situations like this when the exact positions of the waypoints is not perfect, or the accuracy of the localization cannot be trusted, but instead are there to guide Tor along in a rough direction.



(a) Obstacle map with UGV position, planned trajectory and goal point in blue.

(b) Simultaneous view from Tor.

Figure 5.5 Snapshot from Experiment 2. The goal is slightly to the right, but Tor drives straight ahead to avoid hitting the trees.

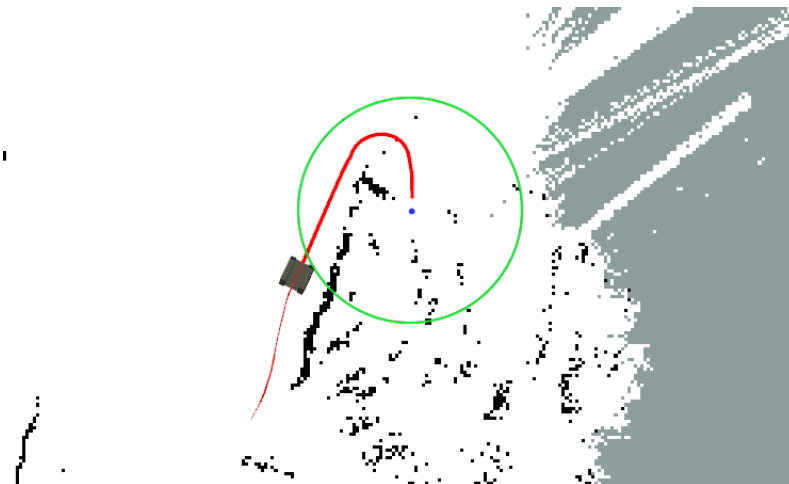


Figure 5.6 Tor right before reaching acceptance radius for a difficult to reach waypoint. Entering the radius means the Tor switches to tracking the next waypoint in the route.

The reason for off-placement of the waypoints was most likely because localization estimate had drifted a few meters away from the true location. In Figure 5.4a, we can see that the driven yellow path is correctly placed to the right of the vehicle trail, while the input waypoints are placed in the middle of the trail, which indicates an error in the localization. This is a consequence of using a local integrated coordinate frame as the main navigation frame (Section 3.3.2). If the system had updated the local coordinate positions used as navigation targets with the most recent global estimate, the position of the goals in the local map would have been improved.

Another issue that affected the performance was the time synchronization issue that we experienced at LandX22 (see Section 4.2). There were still “ghost obstacles” that occurred when Tor made sudden turns and the true obstacles would briefly be placed in the wrong location in the obstacle map. For smooth driving without sudden turns, the ghost obstacles did not affect the performance much. However, when a planned trajectory deviated significantly from the previous, for example, switching from going on one side of a tree to the other, Tor would make a sharp turn to follow the new trajectory, which would cause obstacle ghosting.

During Experiment 2, Tor had at least one incident where a new trajectory caused a sudden maneuver which in turn caused ghost obstacles. The ghost obstacles rendered the new planned trajectory invalid and the motion planner, and the new trajectory would go straight through the ghost obstacles, causing the new trajectory to be invalid. The motion planner would in turn plan a new trajectory where the original trajectory was, causing Tor to turn back to where it first was heading. The result was back-and-forth oscillations, before Tor eventually got back on track.

The last part of the route turned away from the vehicle trail, and into an area with more vegetation. As before, Tor struggled with the low vegetation, and several manual takeovers were needed. The forest section at the end went had larger trees, and not as much of the low vegetation. Thus, Tor performed well in the forest environment with fewer but larger obstacles.

5.2.2 Experiment 2 – Conclusion

Tor drove a total distance of 1409 m and had 7 manual interventions in Experiment 2. Experiment 2 proved to be an interesting experiment that put many of the autonomy features to the test. The intention when placing the input waypoints was to give Tor a not-too-difficult case to emulate Route C. Because the waypoints ended at the side of the vehicle trail, the experiment ended up more challenging as Tor had to interact with more vegetation and forest than planned.

With better-placed waypoints, it is clear that the performance of Tor would have been improved, and a few small detours into the forest would have been avoided. At the same time, we consider this a good example of realistic conditions where we have to assume that all waypoints could be badly placed and the localization is not accurate enough to trust. The real lesson here is that the methods for following vehicle trails need to be even more robust to bad waypoint placement, and have capabilities to follow the trails based on their appearance and not presumed location.

The situations with areas with low vegetation turned out to be a weakness of the autonomy, and while Tor managed to get out from a couple of these on its own, it was effectively random whether it escaped or became stuck.

As for the defined *Route A – Open and flat terrain* from Section 1.1.1, our assumption that an obstacle map is sufficient turned out to be false. We learned that areas with vegetation can cause

problems, and obstacle maps alone might not provide enough information to avoid such areas. We also learned that being able to detect the vehicle trail directly instead of relying on obstacles at the side of the trail and obstacle avoidance would be of great benefit for effectively following vehicle trails.

5.3 Experiment 3 – Dirt Road

Experiment 3 featured a long and fairly straight dirt road, and was, in theory, a straightforward experiment. The road curved slightly back and forth, but not enough to really be considered turns. The road went through mostly flat terrain. It did not have ditches to the sides, but instead typically smooth transitions into open space or vegetation or forest patches. Experiment 3 was a prime example of a *Route B – Dirt Road* from Section 1.1.1, and actually covers half of the sample route that can be seen in Figure 1.1. The beginning of the Experiment 3 can be seen in Figure 5.7b.

Despite appearing to be a simple experiment, we did not have high expectations for the dirt road following experiment. In Appendix A, we have hypothesized that for reliably being able to follow a road, the autonomy should have a perception system that can recognize the road. Tor did not have such a system for actively detecting and following the road, but instead relied on avoiding obstacles. Thus, the performance would rely on the road being surrounded by obstacles to successfully achieve road following.

5.3.1 Experiment 3 – Results

We initially created an input route with 5 waypoints for the whole road section, that we sent to Tor. The waypoints were placed on the road, but the overall route did not follow the road exactly. Tor started driving, and it quickly became clear that Tor did not follow the road to a satisfactory degree. The first waypoint ended up slightly to the side of the road, which caused Tor to drive along the side of the road towards the waypoint. Unlike in Experiment 2, where there were obstacles to guide Tor along the vehicle trail, the first section did not feature any and Tor ended up driving along the direct path to the waypoint.

After a manual override to drive Tor back onto the road, a software error caused the current route to be deleted from the autonomy system. We wanted to see if Tor could perform better with an input route of higher accuracy. We therefore created a new route that followed the road closer, and where we placed the waypoints accurately. The new route as well as the driven path of Tor in Experiment 2 can be seen in Figure Figure 5.7a.

Tor was able to follow the road better with the new route as expected as Tor tries to steer directly towards waypoints. However, the new route did not track the road perfectly, and as a result, Tor still ended up to the side of the road several times. On example can be seen in Figure 5.8. We also had an issue with the same behavior that caused the curved paths across the open field of Experiment 1 (seen in Figure 5.2a). The same software error caused Tor to steer not directly towards the waypoints but at a slight angle on some occasions.

On two more occasions did Tor end up too much on the side of the road, and a manual override back to the road was needed.



(a) Results from following a dirt road. Input waypoints in blue and driven path in yellow.

(b) Tor on the dirt road at the beginning of Experiment 3

Figure 5.7 Experiment 3 – Tor following a dirt road.



(a) Obstacle map with UGV position, planned trajectory and goal point in blue.

(b) Simultaneous view from Tor, which is almost off the road.

Figure 5.8 Snapshot from Experiment 3.

5.3.2 Experiment 3 – Conclusion

Tor drove a total of 906 m along the dirt road in Experiment 3, and had 3 manual overrides.

Experiment 3 was a simple experiment, and is a case that is highly relevant for a UGV in any military operations. The relatively poor performance highlights the importance of methods that somehow can detect roads. The performance of Tor in this experiment is clearly something that needs to be improved. In the end, Tor was not able to reliably navigate along the road, and still had manual interventions. This is despite that a detailed route that followed the road closely was used.

A UGV operator should also not be required to place waypoints that follow the road. Ideally, only the goal waypoint should be needed, and the autonomy should handle the rest. Only needing a single waypoint could be achieved in at least two ways:

- Using a route planner that uses prior map information to plan a detailed route.
- Using a road detection and following system, that makes the UGV able to follow the road autonomously.

Ideally, both of these systems should be combined to increase the robustness of the autonomy system.

5.4 Experiment 4 – Varied terrain through forest

In Experiment 4, we wanted to challenge the autonomy system with more diverse terrain. The area we selected for testing featured a mix of wide and narrow vehicle trails and sparse forest. The terrain was rougher and not as flat as the terrain in the previous experiments.

We started the experiment in an area with wide vehicle trails, and this was the least challenging part of Experiment 4. After that, the terrain consisted of denser forest with narrow vehicle trails running through it. Following that section was a longer stretch of sparse forest with different types of vehicle trails running through it. The experiment ended with a long decline down to the starting point for Experiment 1. In Figure 5.9 we can see four examples of the terrain types in Experiment 4.

In Experiment 4, we also wanted to see how well the autonomy would handle waypoints being placed in a manner that did not follow particular vehicle trails, but instead more or less at random. This would mean that driving straight towards waypoints with minor obstacle avoidance would most likely not work satisfactory. Instead, the autonomy system would have to explore the perceived terrain more and plan trajectories that would go where it is possible to drive. This increased challenge also meant that the likelihood of Tor getting stuck or not finding a viable route forward would be higher.

5.4.1 Experiment 4 – Results

The execution of Experiment 4 was split into two parts, first the wide vehicle trail section that ended at a road crossing, and then we manually drove Tor across the road before we started the second part through the forest.



Figure 5.9 Experiment 4 – Tor driving through different terrain types.



Figure 5.10 Experiment 4 – Input routes in blue and driven path in yellow. The start position was to the left. A closer look at the area within the rectangular marking can be seen in Figure 5.11.



Figure 5.11 Experiment 4 – A closer look at the driven path through a narrow vehicle trail section. The numbers indicate issues or manual interventions.

For the first section with wide vehicle trails, we created a route that followed the trail. The location of the first waypoint ended up being placed in a patch that was too rough for Tor to pass. Here, there were deep ruts and low friction that left Tor without traction, but there were no positive obstacles that Tor could detect. The perception therefore did not classify this area as obstacles, and we had to manually reposition Tor and upload a new route. With the new route, Tor did not have any issues during execution, and followed the planned route and avoided obstacles at the sides of the trail smoothly.

For the second section, we created a route by placing waypoints randomly through the forest without considering existing vehicle trails. The route covered 1800 m and had 7 waypoints, with distances between waypoints of 200–400 m. As expected, this section proved to be the most challenging of all the experiments, and the autonomy struggled and required numerous manual interventions. At the same time, the experiment served well to highlight issues and weaknesses of the autonomy stack.

Experiment 4 after the first wide vehicle trail section was a long experiment, and it took nearly an hour to complete. Instead of trying to analyze the entire run in detail, we have chosen to take a deeper look at one of the most challenging sections of Experiment 4. The section features a narrow vehicle trail through a forest patch that proved to be difficult to traverse. We discovered a few new issues in this section that we had not experienced during the other experiments. The issues experienced here together with the issues described during Experiments 1–3 are representative for the issues experienced during the rest of Experiment 4.

5.4.2 Case study – Issues along a narrow vehicle trail

In Figure 5.11, we can see a more detailed view of the driven path through an area with narrow vehicle trails at the beginning of the experiment. The location of this section is marked in Figure 5.10. This short stretch offered many challenges, and also exposed some weaknesses that we had not seen in the previous experiments. We have marked and numbered every issue that we experienced during this run in Figure 5.11.

Issue 1 – Planner cannot decide between right and left Tor arrives at junction, and right and left are equally good. The motion planner therefore switches between planning paths to left and right. Tor cannot respond quickly enough and drives straight ahead until getting stuck in front of an obstacle in the middle.

Issue 2 – Planner plans path into forest The motion planner planned a trajectory away from the trail and into the forest. The new trajectory is heading more directly towards the goal, and the perception does not separate between vehicle trail and forest, which makes the new one a better trajectory for the motion planner. In reality, the optimal choice for Tor would have been to continue driving straight ahead on the vehicle trail as a trail has a better chance of leading forward compared to an unstructured forest. The behavior displayed is expected with the current autonomy, but highlights that being able to prioritize some types of terrain, such as vehicle trails over others like open space in a forest could improve overall performance.

This could be achieved both by semantic segmentation as discussed for road detection in Experiment 3, and by creating a cost map based on, for example, the smoothness of the terrain.

Issue 3 – Tor follows path into forest, gets stuck too close to obstacle Tor started to follow the planned trajectory that heads off the trail. The trajectory featured a sharp turn to the right, and when turning, Tor got too close to the obstacle. Tor did not collide with the obstacle, but got too close for the motion planner to plan a collision-free trajectory forward. We therefore had to take manual control to get Tor back to the trail.

Issue 4 – Vehicle trail perceived as too narrow After getting back to the trail, Tor was unable to find a valid trajectory forward that would follow the trail forward. This was caused by Tor perceiving the vehicle trail as too narrow. The trail could be seen in the map, but when the obstacles were dilated to account for the width of Tor, there was no space left for Tor to pass. In reality, the vehicle trail was wide enough to pass through, even when the safety margin added in the motion planner was accounted for. This could mean that there was some error in the perception, for example, caused by a discretization effect of the perceived obstacles or some error in the fusion of the obstacle map over time. Another possibility could be that the perception picked up overhanging branches and marked them as obstacles in the map.

We tried to intervene by performing a manual reset of the obstacle map. This deleted all obstacles from the map, and the perception started building a new obstacle map with new measurements. The reset worked, and Tor was able to start following the vehicle trail.

Issue 5 – Vehicle trail switches between too narrow and wide enough After the first perception reset, there was a stretch forward where the perception continuously switched from marking the vehicle trail as passable and too narrow and back again. When marked as too narrow, the motion planner was unable to plan a trajectory forward. When marked as wide enough, the motion planner would plan a trajectory and Tor would be able to move forwards until the next time the switch occurred. This process made it possible for Tor to continue advancing, but with stops occurring when the motion planning failed. A couple of times, the motion planner did not find a path through, but was able to plan paths that took the vehicle a little forward, before turning away from the trail. The net result was that Tor could advance, and the motion planner was able to plane new paths on the trail before Tor could drive into the forest.

Tor managed to get past a good portion of the narrow section on its own, but that was not deterministic and not a robust motion planning system that was able to deal with the changing environment in a consistent manner. If we repeated the same section several times, it is unlikely that the run would play out the same way. The situation with the narrow environment requires a more sophisticated motion planner to avoid randomness playing such a significant role in the execution. The strategy to plan trajectories at fixed intervals based on the most current map means that too much relies on the map being consistent. Instead, previous planned trajectories should be included when planning new trajectories to increase the consistency of the planning.

Issue 6 – Perception resets causing Tor to drive off the trail Eventually, Tor got to a point on the trail where the perception was unable to detect an open space forward along the trail. We therefore intervened with a manual perception reset. After the reset, the trail was still not passable, but a few obstacles close to Tor that previously were marked as obstacles were not detected. This led to the motion planner planning a trajectory through the obstacles and into the forest. We therefore had to take control manually and get Tor back to the trail. The situation demonstrates the potential risks of resetting the obstacle map that has been built up over time. The resets were a temporary solution to keep Tor moving along, but it could be a symptom of errors in how the fused obstacle map is being created and maintained over time as the resets helped.

Issue 7 – Tor starts driving into forest, but recovers by itself Towards the end of the narrow vehicle trail, Tor ended up planning a trajectory away from the trail and into the forest. Tor followed the trajectory for a short distance, before receiving a new trajectory that followed the trail. Tor followed the new trajectory, and was able to complete the vehicle trail without intervention. The situation once again demonstrates how the autonomy system can be inconsistent with the choices it makes. There should be a larger penalty for switching to a different trajectory once a choice has been made. This could make the system more predictable, and therefore easier for an operator to trust.

Issue 8 – Perception detecting gate as too narrow After exiting the vehicle trail and entering a dirt road, Tor approached an open gate. The opening was perceived as too narrow, even though there clearly was enough space for Tor to pass through. Tor first turned away from the gate, before finding a trajectory through and turning back. Right before passing through, the gate opening was perceived as too narrow and Tor got stuck without finding any trajectories forward. The situation is shown in Figure 5.12.



(a) Obstacle map with UGV in front of the gate. The gate is too narrow to pass in the obstacle map
 (b) Simultaneous view from Tor. The gate is clearly wide enough to pass.

Figure 5.12 Tor in front of a gate. The gate is wide enough to pass, but the perception detects it as too narrow, making the motion planning fail.

The situation was similar to the ones on the narrow vehicle trail, but this time there was clearly enough room to pass by a good margin, and there was less clutter on the sides that could influence the measurements. This points to there being some systematic error in the perception system that causes obstacles to appear wider than they are.

5.4.3 Experiment 4 – Forest with vehicle trails

The rest of Experiment 4 went smoother than the narrow section. The terrain was easier to drive in, the forest was sparser and the vehicle trails were wider. It was still necessary to take control over the autonomy on several occasions, but Tor was able to drive longer between each intervention. The autonomy also showed the ability to successfully handle some situations that we did not expect it to do. One such example was being able to find a drivable path through trees and successfully navigating through. The situation can be seen in Figure 5.13 where Tor plans a path past a section with trees and low vegetation and is able to navigate through. Other examples were areas where the terrain was rough, where the perception still was able to correctly separate between open space and obstacles, which can be challenging in such situation.

The final decline towards the goal point was a section where Tor again struggled. The decline made it more difficult for the perception to separate between ground and obstacles, and the result was that approximately half of the decline had to be driven manually.

5.4.4 Experiment 4 – Conclusions

The total driven distance of Experiment 4 was 2886 meters, and we had 26 manual interruptions where we either had to take control and drive manually or reset the perception.

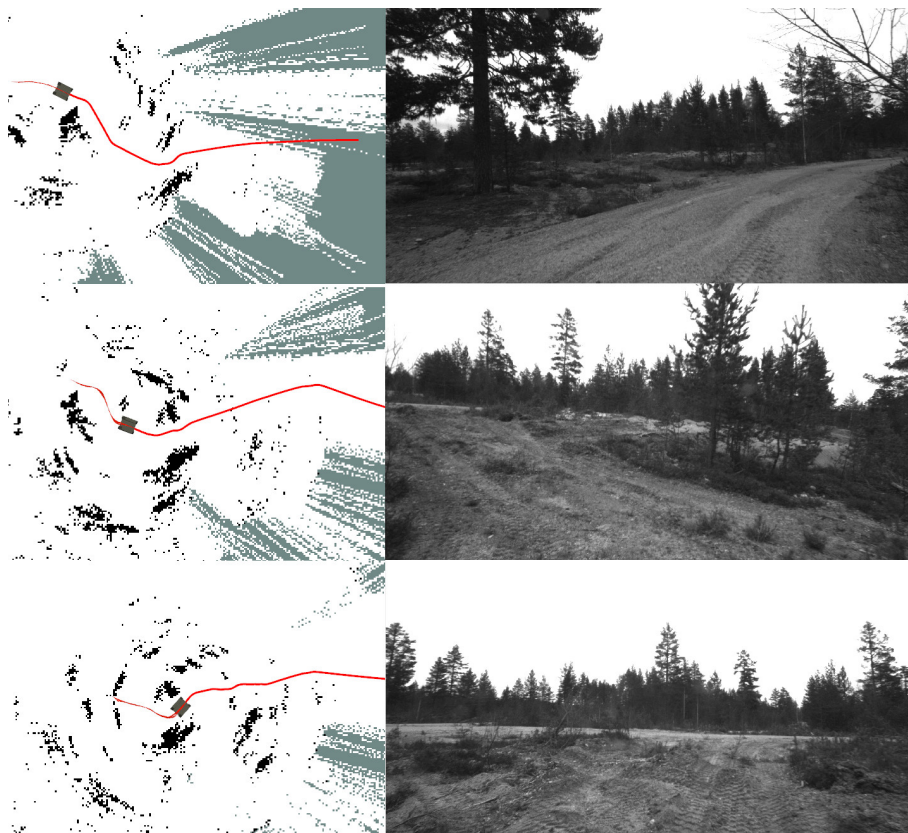


Figure 5.13 Tor navigating past a complex section with trees and low vegetation.

The routes went through varied and challenging terrain. In terms of the routes from Section 1.1.1, the experiment contained elements from *Route C – Wide and open vehicle trail*, *Route E – Narrow vehicle trail without obstacles* and *Route G – Sparse and flat pine forest*. Of these, the *Route C* part was the one that went smoothest, and the run from Experiment 2 proved to be a more difficult challenge than the segment that featured in this experiment. The parts with the narrow vehicle trails were the most troublesome section in the experiment. The autonomy system struggled with both finding drivable paths forward, and there were also issues with lack of consistency in the motion planning. There were also several shorter sections throughout Experiment 4 that were on medium-width vehicle trails. These also were right in between the narrow and the wide trails in regard to how well the autonomy handled them.

The areas of Experiment 4 with typical *Route G* environment – Sparse and flat pine forest – went mostly without interruptions. Nevertheless, there were still situations where this occurred partly by chance and Tor could easily have gotten stuck on several occasions. The section at the end with steep declines was not as steep as the declines of the benchmark *Route D – Vehicle trails in steep inclines*. The autonomy also struggled during this section, so there is no reason to expect more success in even steeper environments.

Overall, Experiment 4 served both to demonstrate the capabilities of the autonomy system and the limitations. In the end, Tor did drive autonomously through a large portion of a forest with no prior information about where to go, besides the coarse input route. The experiment also highlighted that the autonomy has a long way to go to increase the consistency of the decisions it makes. Experiment 4 did put the autonomy of Tor to the limit in a number of ways, and will serve as a great benchmark for future experiments.

5.5 Long-range driving experiment conclusions

During the long-range driving experiment, Tor ended up driving 6.2 km where most of it was by itself. Figure 5.14 shows the full driven path during the long-range driving experiment, with the input waypoints marked as yellow circles, the sections where Tor was in autonomous mode marked in green, and the sections where Tor was driven manually marked in red. The experiment was a comprehensive trial for the autonomy stack, and served both to demonstrate the capabilities and the limitations in the system. We learned valuable lessons about what areas need to be improved, and the tests themselves have been great inspiration for future developments.

Some of the most important findings are:

- Not relying on accurate localization and waypoint placement meant the system could handle challenging situations such as free navigation in unstructured terrain reasonably well.
- Letting the motion planner handle the planning with enough room to make real decisions works. There were several complex situations where the UGV was able to find its way through obstacles.
- A route planning with accurate placement of waypoints could help to guide the UGV in the correct direction. This would be especially important where there are known vehicle trails that could be followed.

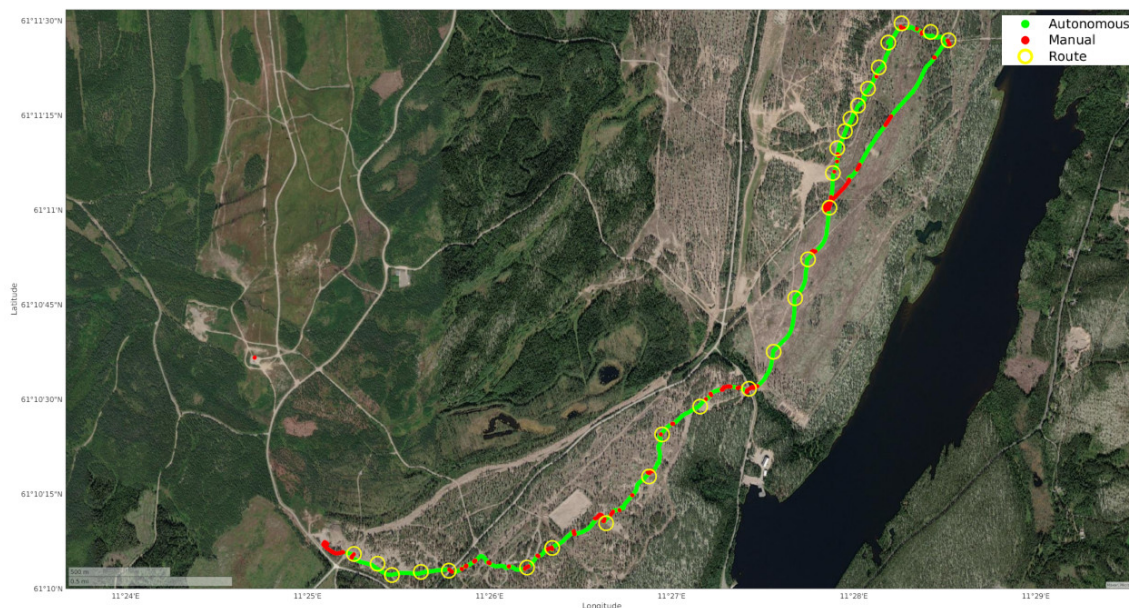


Figure 5.14 Driven path of Tor during the long-range driving experiment. Green is in autonomous mode, red is in manual mode. The yellow circles marks the waypoints from the input routes.

- The system as a whole is too unpredictable, it is hard to know from the outside what the system would do next, and some decisions came down to luck.
- We do not have a reliable way of handling low and sparse vegetation that can easily be driven through. Low vegetation can occur almost anywhere, and needs to be accounted for in the perception and the planning to avoid autonomy failures.
- Dirt road or trail segmentation or tracking is needed – it would have greatly improved performance in trail and road following.
- Using only an obstacle map (instead of a cost map) provides insufficient information in many situations. It could often be preferable to drive where it is smoothest, flattest and in recognized vehicle trails or roads.
- Using localization based on integrated odometry worked well, and it will be interesting to test how well the autonomy could perform without GNSS at all.

Some of these lessons can be applied to the autonomy stack relatively quickly, while others will require more time and research to be remedied.

We have tried to run our autonomy in environments representing many of the canonical routes that we defined Section 1.1.1. We can not say that our autonomy definitively can handle any of these routes, as we did have issues when running in all of them. Our system did, however, perform better in some environment types:

Route A – Open and flat terrain without vegetation Our system performed well, did get stuck due to low vegetation. If the vegetation was taller, it would likely not have had any issues. In practice, the definition reads no vegetation, but some vegetation must always be expected in terrain.

-
-
- Route B – Dirt road** Our system did not hit obstacles, but was not able to follow the road. As expected, some method for tracking the road is necessary.
- Route C – Wide and open vehicle trails** Our system performed well, and managed to stick to the vehicle trail despite bad waypoint placement. This result was mainly because there were trees to the side that the motion planner could avoid. As with the dirt roads, we did not have a system for actively following the trails, and without obstacles to the sides, the performance would likely be worse.
- Route D – Vehicle trail in steep inclines** We attempted to run the system in moderate decline, and the obstacle detection did not handle it well. Improved perception is needed if we were to attempt even steeper terrain.
- Route E – Narrow vehicle trail without small obstacles** Our system struggled with the narrow vehicle trails, the obstacle detection often caused the trail to be blocked off, and the motion planner had no way of handling those situations. A trail tracking mode would be needed.
- Route F – Narrow vehicle trail with small obstacles** Not attempted, but given the results from both narrow vehicle trails and areas with low vegetation, we would not expect this case to go well.
- Route G – Sparse and flat pine forest** The system did fairly well in these environments. Detecting and avoiding large obstacles is one of the main strengths of our system, even in complex environments. Still, there were many situations in such environments where our system got stuck, and the reliability needs to increase.
- Route H – Dense forest** Not attempted, and would be far beyond the capabilities of our current autonomy system.

6 Conclusion

Autonomy for unmanned ground vehicles remains one of the most challenging problems in robotics, particularly in unstructured and off-road environments. Our work set out to explore this challenge by developing an autonomy stack and evaluating it against a set of benchmark routes designed to represent diverse terrain types. These routes have proven to be a useful framework for structured testing and for identifying the strengths and weaknesses of our approach.

The autonomy stack we developed is modular and capable of basic autonomous navigation. It demonstrated the ability to traverse open terrain and sparse forest without prior map information, relying solely on onboard perception and planning. At the same time, the system is still in an early stage of development. It lacks advanced capabilities such as semantic classification of the enviro, map-based route planning, and the ability to recover from errors. Predictability and robustness remain key challenges, and many decisions still are not deterministic rather than formed from consistent reasoning. All these factors are essential for reliable performance in complex environments like narrow trails or dense forests, and improving them will be pursued in the future.

Despite these limitations, the experiments confirm that our approach, which focuses on autonomy without dependence on precise localization or pre-defined paths, has merit. It offers flexibility and demonstrates potential in handling certain scenarios that arise during execution, even if overall performance remains limited. The lessons learned provide a way forward for improvement: improved perception for vegetation and terrain classification, more sophisticated motion planning, and integration of route-following strategies.

Looking ahead, we see clear value in applying established autonomy standards such as the ALFUS[16] and SAE J3016[18] standards to assess the maturity of our system during execution in these benchmark scenarios. While our system is not yet mature enough for formal classification, these frameworks can guide future development and help quantify progress. Ultimately, achieving higher autonomy levels will require not only technical enhancements but also rigorous evaluation against diverse and realistic scenarios.

In summary, this work represents a step toward autonomous UGVs capable of operating in challenging terrain. The system provides a foundation for continued research, and the benchmark routes offer a repeatable and scalable method for assessing progress. With sustained development, we can move closer to the vision outlined in the introduction: UGVs that reduce operator burden, enhance mission flexibility, and operate effectively in demanding environments.

References

- [1] Magnus Baksaas, David Kolden, Eilert Mentzoni, Tønnes Nygaard, Anders Fuglested Nilsen, Erik Hexeberg Henriksen and Tord Fauskanger. *Ubemannet logistikkolonne – utvikling og testing av en demonstrator*. FFI-report 21/01791. FFI, 2021.
- [2] Magnus Baksaas and Niels Hygum Nielsen. *Milrem THeMIS 4.5 - grensesnitt og lavnivå kontroll*. FFI-Report 21/00520. Unntatt Offentlighet. Norwegian Defence Research Establishment, 2021.
- [3] Lars Kristian Bårdevik. ‘Motion Planning for UGVs in Terrain Scenarios Based on RRT, RRT* and the Fast Marching Method’. <http://hdl.handle.net/10852/105394>. MA thesis. University of Oslo, Department of Technology Systems, 2023.
- [4] Dan Helge Bentsen et al. *LandX23 - target acquisition and engagement*. FFI Report 24/00357. Norwegian Defence Research Establishment, 2024.
- [5] Sondre Bø. ‘Motion planning for terrain vehicles: Path generation with radial-constrained A* and trajectory optimization’. <https://hdl.handle.net/11250/2780886>. MA thesis. Norwegian University of Technology and Science, Department of Engineering Cybernetics, 2020.
- [6] Dmitri Dolgov, Sebastian Thrun, Michael Montemerlo and James Diebel. ‘Practical Search Techniques in Path Planning for Autonomous Driving”. In: *Proc. First Int. Symp. Search Techniques in Artificial Intelligence and Robotics (STAIR-08)*. Chicago, USA, 2008.
- [7] Péter Fankhauser, Michael Bloesch and Marco Hutter. ‘Probabilistic Terrain Mapping for Mobile Robots with Uncertain Localization’. In: *IEEE Robotics and Automation Letters (RA-L)* 3.4 (2018), pp. 3019–3026.
- [8] Péter Fankhauser and Marco Hutter. ‘A Universal Grid Map Library: Implementation and Use Case for Rough Terrain Navigation’. In: *Robot Operating System (ROS) – The Complete Reference (Volume 1)*. Ed. by Anis Koubaa. Springer, 2016. Chap. 5.
- [9] Tully Foote. ‘tf: The transform library’. In: *Technologies for Practical Robot Applications (TePRA), 2013 IEEE International Conference on*. Open-Source Software workshop. Apr. 2013, pp. 1–6.
- [10] Paul Furgale, Joern Rehder and Roland Siegwart. ‘Unified temporal and spatial calibration for multi-sensor systems’. In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. 2013.
- [11] Kenneth Gade. ‘A Non-singular Horizontal Position Representation’. In: *The Journal of Navigation* 63.3 (2010), pp. 395–417. doi: 10.1017/S0373463309990415.
- [12] Kenneth Gade. ‘NavLab, a Generic Simulation and Post-processing Tool for Navigation’. In: *Modeling, Identification and Control: A Norwegian Research Bulletin* 26.3 (2005), pp. 135–150.
- [13] David González, Joshué Pérez, Vicente Milanés and Fawzi Nashashibi. ‘A Review of Motion Planning Techniques for Automated Vehicles’. In: *IEEE Transactions on Intelligent Transportation Systems* 17.4 (2016), pp. 1135–1145.

-
-
- [14] Trym Vegard Haavardsholm. *A handbook in Visual SLAM*. Tech. rep. 2021. URL: <https://github.com/tussedrotten/vslam-handbook/releases/tag/v0.1.0>.
- [15] Morten Hanevold. 'Path Following Model Predictive Control of a Differential Drive UGV in Off-Road Terrain'. <http://hdl.handle.net/10852/95375>. MA thesis. University of Oslo, Department of Technology systems, 2022.
- [16] Hui-Min Huang, Kerry Pavek, Brian Novak and Elena Messina. 'A Framework For Autonomy Levels For Unmanned Systems (ALFUS)'. In: *Proceedings of the AUVSI's Unmanned Systems North America 2005*. June 2005.
- [17] Goran Huskić, Sebastian Buck, Matthieu Herrb, Simon Lacroix and Andreas Zell. 'High-speed path following control of skid-steered vehicles'. In: *The International Journal of Robotics Research* 38.9 (July 2019), pp. 1124–1148.
- [18] *J3016 Surface Vehicle Recommended Practice - Taxonomy and Definitions for Terms Related to Driving Automation Systems for On-Road Motor Vehicles*. J3016_202106. SAE International. June 2021. URL: https://www.sae.org/standards/content/j3016_202104/.
- [19] Kendra Diezel, Jeff McGough, Chris Smith, Andrew Stelter, Riley Sutton, Samuel Williams. *Veranda*. <https://github.com/roboscienceorg/veranda>. Version 1.0. 1st Jan. 2019.
- [20] Thomas Røbekk Krogstad et al. *HAL - A decisional autonomy module for unmanned systems*. FFI Report 20/00896. Unntatt offentligheten. Norwegian Defence Research Establishment, 2020.
- [21] Steven M. Lavalle. *Planning Algorithms*. Cambridge University Press, 2006. ISBN: 0521862051.
- [22] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu and Alexander C. Berg. 'SSD: Single Shot MultiBox Detector'. In: *European Conference on Computer Vision (ECCV)* (2016). URL: <http://arxiv.org/abs/1512.02325>.
- [23] Kim Mathiassen et al. 'Demonstrating interoperability between unmanned ground systems and command and control systems'. In: *International Journal of Intelligent Defence Support Systems* 6.2 (2021), pp. 100–129.
- [24] Kim Mathiassen et al. *LandX20 Experiment report - Public version*. FFI Report 22/00274. <http://hdl.handle.net/20.500.12242/3013>. Norwegian Defence Research Establishment, 2022.
- [25] Kim Mathiassen et al. *LandX21 - teknologi i framtidens stridsfelt*. FFI Report 24/00481. Norwegian Defence Research Establishment, 2024.
- [26] Kim Mathiassen et al. 'Making the Milrem Themis UGV ready for autonomous operations'. In: *Unmanned Systems Technology XXIII*. Vol. 11758. International Society for Optics and Photonics. SPIE, 2021, 117580S.
- [27] Kim Mathiassen, Magnus Baksaas, Lars Erik Olsen, Marius Thoresen and Bjørn Tveit. 'Development of an Autonomous Off-Road Vehicle for Surveillance Missions'. In: *STO-MP-IST-127: Intelligence and Autonomy in Robotics*. NATO STO, 2016.
- [28] Wim Meeussen. *REP-105: Coordinate Frames for Mobile Platforms*. 27th Oct. 2010. URL: <https://www.ros.org/repos/rep-0105.html#id12>.

-
-
- [29] Eilert Mentzoni et al. *LandX22 - Experiment report*. FFI Report 24/00481. Norwegian Defence Research Establishment, 2024.
- [30] Eilert Mentzoni and Magnus Baksaas. *Milrem THeMIS 4.5 - Ombygging av kjøretøy og hardware-modifikasjoner*. FFI-Notat 21/00519. Unntatt Offentlighet. Norwegian Defence Research Establishment, 2021.
- [31] Bohdan Miroshnychenko. *The Ukrainian military is set to receive 15,000 combat robots this year. What roles will these ground drones take on?* 3rd Apr. 2025. URL: <https://www.pravda.com.ua/eng/articles/2025/04/03/7505903/> (visited on 20/11/2025).
- [32] Pascal Morin and Claude Samson. ‘Motion Control of Wheeled Mobile Robots’. In: *Springer Handbook of Robotics*. Ed. by Bruno Siciliano and Oussama Khatib. Springer-Verlag Berlin Heidelberg, 2008. Chap. 34, pp. 799–826.
- [33] Edwin Olson. ‘AprilTag: A robust and flexible visual fiducial system’. In: *2011 IEEE International Conference on Robotics and Automation*. 2011, pp. 3400–3407.
- [34] Ole Petter Orhagen. ‘The development of the RRT-Funnel motion planning algorithm’. <http://hdl.handle.net/10852/69325>. MA thesis. University of Oslo, Department of Technology Systems, 2019.
- [35] Ole Petter Orhagen, Marius Thoresen and Kim Mathiassen. ‘The Rapidly exploring Random Tree Funnel algorithm’. In: *2022 8th International Conference on Mechatronics and Robotics Engineering (ICMRE)*. Institute of Electrical and Electronics Engineers (IEEE), 2022.
- [36] Markus Otsus. *Unmanned Ground Vehicles Autonomy Trials - 28.06-29.06.2023, Läsna, Estonia*. Tech. rep. Estonian Military Academy, 2023.
- [37] Brian Paden, Michal Čáp, Sze Zheng Yong, Dmitry Yershov and Emilio Frazzoli. ‘A survey of motion planning and control techniques for self-driving urban vehicles’. In: *IEEE Transactions on Intelligent Vehicles* 1.1 (2016), pp. 33–55. issn: 2379-8858.
- [38] Jesse Pentzer, Sean Brennan and Karl Reichard. ‘Model-based Prediction of Skid-steer Robot Kinematics Using Online Estimation of Track Instantaneous Centers of Rotation’. In: *Journal of Field Robotics* 31.3 (Mar. 2014), pp. 455–476.
- [39] Jouni Rantakokko and Jonas Nygård. *Obemannade farkoster för markstriden – Erfarenheter från Ukraina*. FOI Rapport FOI-R–5723–SE. Totalfrsvarets forskningsinstitut (FOI), July 2025.
- [40] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick and Ali Farhadi. ‘You Only Look Once: Unified, Real-Time Object Detection’. In: *CoRR* abs/1506.02640 (2015). arXiv: 1506.02640. URL: <http://arxiv.org/abs/1506.02640>.
- [41] Milrem Robotics. URL: <https://milremrobotics.com/defence/> (visited on 10/01/2022).
- [42] Frank Schneider, Kim Mathiassen, Paul Bounker, Jakub Glówka, Alexander Tiderko, Magnus Baksaas, Rafal Kozik, Juha Röning and Geert De Cubber. *Capability Concept Demonstrator for Interoperability within Unmanned Ground Systems and C2*. NATO STO Technical Report STO-TR-IST-149. 2021.

-
-
- [43] Bijo Sebastian and Pinhas Ben-Tzvi. ‘Active Disturbance Rejection Control for Handling Slip in Tracked Vehicle Locomotion’. In: *Journal of Mechanisms and Robotics* 11.2 (Feb. 2019).
- [44] Rikke Amilde Seehuus et al. ‘Battle management language for robotic systems - Experiences from applications on an UGV and an USV’. In: *International Conference on Modelling and Simulation for Autonomous Systems (MESAS 2018)*. 2018.
- [45] James A. Sethian. ‘A fast marching level set method for monotonically advancing fronts.’ In: *Proceedings of the National Academy of Sciences* 93.4 (1996), pp. 1591–1595.
- [46] Bruno Siciliano, Lorenzo Sciavicco, Luigi Villani and Giuseppe Oriolo. *Robotics - Modeling, planning and control*. Springer London, 2009.
- [47] Espen Skjervold. *Design og implementasjon av FFI GCS - kontrollstasjon for ubemannede systemer*. FFI Report 15/01620. Norwegian Defence Research Establishment, 2016.
- [48] Ragnar Smestad, Martin Vonheim Larsen and Trym Vegard Haavardsholm. *Superflow*. FFI-report 19/00776. FFI, 2019.
- [49] Irwin Sobel. ‘An Isotropic 3x3 Image Gradient Operator’. In: *Presentation at Stanford A.I. Project 1968* (Feb. 2014).
- [50] Marius Thoresen and Morten Hanevold. *Tracking and following a lead vehicle – A Distributed Cooperative Demonstration of Technology in AVT-408*. FFI-report 26/xxxxx. FFI, 2026.
- [51] Marius Thoresen, Niels Hygum Nielsen, Kim Mathiassen and Kristin Y. Pettersen. ‘Path Planning for UGVs based on Traversability Hybrid A*’. In: *IEEE Robotics and Automation Letters* (2021), pp. 1216–1223.
- [52] Andreas Tisland. ‘Path Following in Off-Road Conditions using Kinematic Control for Differential Drive Vehicles’. <http://hdl.handle.net/10852/87255>. MA thesis. University of Oslo, Department of Technology systems, 2021.
- [53] Andreas Tisland, Magnus Baksaas and Kim Mathiassen. ‘How extending the kinematic model affects path following in off-road terrain for differential drive UGVs’. In: *Unmanned Systems Technology XXIV*. SPIE, May 2022.
- [54] Bryan Walther. ‘Local Planning using Fast Marching Square’. <http://hdl.handle.net/10852/112195>. MA thesis. University of Oslo, Department of Technology Systems, 2024.
- [55] Martin S. Wiig, Rikke A. Løvliid, Kim Mathiassen and Thomas R. Krogstad. ‘Decision Autonomy for Unmanned Vehicles’. In: *Proceedings of IST-127/RSM-003 Specialists’ Meeting on Intelligence & Autonomy in Robotics*. NATO UNCLASSIFIED. NATO Science and Technology Organization. Bonn, Germany, Oct. 2016.
- [56] Lipu Zhou, Zimo Li and Michael Kaess. ‘Automatic Extrinsic Calibration of a Camera and a 3D LiDAR Using Line and Plane Correspondences’. In: *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. 2018.

A Benchmark routes and autonomy requirements

To evaluate progress in developing autonomy for UGVs, we defined a set of benchmark routes in Section 1.1.1. These routes represent different types of terrain, ranging from simple and open environments to dense forests with complex obstacles. The purpose of these routes is to provide a structured way to assess the autonomy system, identify weaknesses, and guide further development. The routes we selected are shown in Figure A.1, and representative examples of the type of terrain in each route are shown in Figure A.2 and Figure A.3.

The requirements we outline for each route are hypotheses based on our current understanding of autonomy capabilities. We have broken them down into three core autonomy components: *perception*, *motion planning*, and *trajectory following*. The requirements represent what we believe to be the minimum necessary for successful traversal. Actual requirements may differ as we gain experience and improve the system. Reliability of methods is not considered in this breakdown—for example, a simpler but more reliable perception method may outperform a complex but less accurate one.

All routes are located in the Rødsmoen military exercise area in Norway, which means they share similar vegetation and soil types. While this limits diversity, the selected routes still challenge the autonomy system in different ways. Future work may include adding routes from other areas to increase variability.

The benchmark routes are not intended as an exhaustive set but as a practical starting point. They allow us to measure progress, avoid bias toward easy scenarios, and stress-test the autonomy system. The following sections describe each route and the autonomy requirements we believe are necessary to complete them.

A.1 Route A – Open and flat terrain

In this case, we assume that the area is mostly drivable and that there are only occasional obstacles that must be circumvented. An example of such an environment can be seen in Figure A.2a.

Perception The perception system needs to differentiate between obstacles and free space and create an obstacle map.

Motion planning The UGV must perform collision avoidance. The environment has few obstacles, and it should be possible to navigate around these without large deviations.

Trajectory following With a pure collision avoidance strategy, it is enough for the UGV to steer in a given direction that changes. With a deliberate planning strategy, the UGV needs to be able to follow a path.

A.2 Route B – Dirt road

Easily traversable for a terrain vehicle, we can assume that the road is wide enough for a vehicle, reasonably flat, and without sharp turns. These factors mean that the UGV should be able to drive



Figure A.1 The benchmark routes at Rødsmoen.

at higher speeds. The challenge is to detect the road accurately and use its location information to produce smooth paths. An example of a dirt road can be seen in Figure A.2c.

Perception The UGV should be able to identify the road through image segmentation or a combination of image segmentation and point clouds.

Motion planning The UGV should have a road-following mode where the planned paths follow the road itself and not anywhere it is drivable. The paths should also be smoother and more predictable than in terrain.

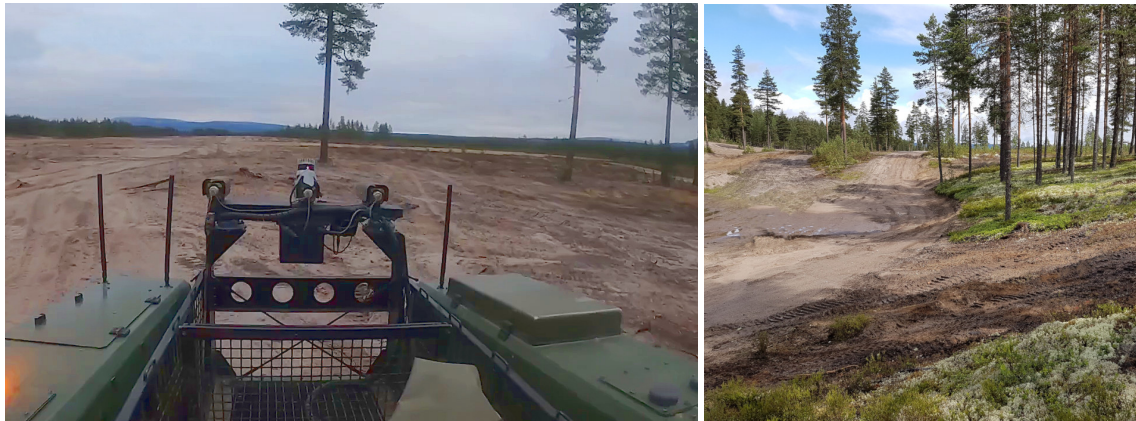
Trajectory following A road allows for higher speeds, and the UGV should therefore be able to follow the road at higher speeds than when driving in terrain.

A.3 Route C – Wide and open vehicle trail

With Route C, we assume that there is an open vehicle trail with few obstacles, potentially surrounded by areas densely populated with obstacles. It should be easily traversable as long as the correct decisions are made, i.e., the UGV follows the trail instead of leaving the trail, which could lead to the UGV getting stuck at a dead end. An example of a wide vehicle trail can be seen in Figure A.2b.

Perception The environment has a high obstacle density, but we can assume that there is a reasonably traversable trail that is possible to follow. Obstacle maps should suffice for this type of driving.

Motion planning Deliberate motion planning is needed. Reactive collision avoidance can easily become stuck in a dense obstacle environment.



(a) Route A – Open and flat terrain

(b) Route C – Wide and open vehicle trail



(c) Route B – Dirt road



(d) Route D – Vehicle trails in steep inclines

Figure A.2 Example of environments for the routes A–D.

Trajectory following The UGV must be able to follow a path with reasonable accuracy and speed.

A.4 Route D – Vehicle trails in steep inclines

Trails with steep ascents and descents pose an extra challenge for the perception system, especially if it is based on estimating the terrain geometry. Obstacle detection typically relies on detections above the ground plane, but if the ground is inclined, estimating height relative to the ground becomes more challenging. Route D is therefore suitable for revealing limitations in the perception algorithms. An example of a steep incline (with a UGV traversing it) can be seen in Figure A.2d.

Perception Binary obstacle maps can be sufficient, but the underlying method for creating obstacle maps may not work with large inclines.

Motion planning The same motion planning as for Route C should be sufficient.

Trajectory following The same as for Route C.

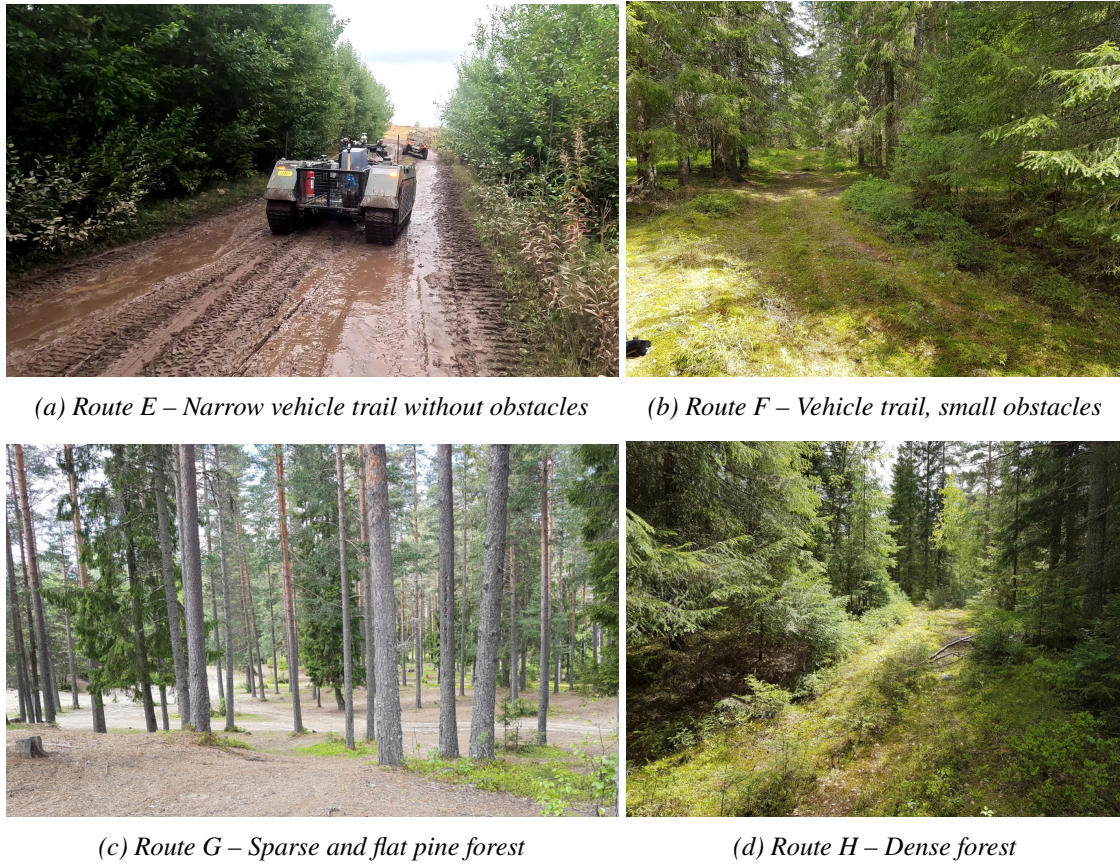


Figure A.3 Example of environments for the routes E–H.

A.5 Route E – Narrow vehicle trail without obstacles

A narrow trail should be traversable for the UGV, but it poses a challenge for the perception system to precisely determine where it is drivable. With a conservative perception system, obstacles can be classified as larger than they truly are, resulting in the narrow trail being perceived as too narrow to drive along. The system would also be vulnerable to misclassification errors of the environment. One example would be vegetation at the sides of or overhanging the trail, which can make the obstacles appear impassable on the narrow trail. An example of such an environment can be seen in Figure A.3a.

Perception Being conservative in obstacle classification means that there is little risk of colliding with obstacles. At the same time, being too conservative can mean that the vehicle trail is considered too narrow to pass when in reality it is not. Precise obstacle classification is therefore important.

Motion planning In narrow spaces, there is less room for error before collisions occur. The motion planner must therefore plan precise and robust trajectories to avoid collisions.

Trajectory following The UGV needs to be able to follow trajectories with high accuracy.

A.6 Route F – Narrow vehicle trail with small obstacles

With the addition of small obstacles, the requirements for the autonomy system are higher. Small obstacles can include low vegetation, tree stumps, and rocks. Small obstacles may sometimes be traversable, and classifying them as obstacles in a narrow environment will not work as it will lead to narrow trails being blocked off. Smarter obstacle detection is therefore necessary. An example of such an environment can be seen in Figure A.3b.

Perception Binary obstacle maps may be sufficient if there are no false positive obstacle detections. The underlying method for creating obstacle maps requires higher accuracy in how it classifies obstacles.

Motion planning The same motion planning as for Route C should be sufficient.

Trajectory following The same as for Route E.

A.7 Route G – Sparse and flat pine forest

This terrain type is characterized by sparsely placed pine trees and is usually fully traversable but can require sharp turns to move between trees. An example of such an environment can be seen in Figure A.3c.

Perception Obstacle maps are sufficient.

Motion planning The environment can be challenging to maneuver in, and the same algorithms used for driving along a vehicle trail might not work for driving back and forth between trees. The motion planner must also be able to plan precise paths between narrow openings and close to obstacles, while at the same time being able to maintain safe distances to obstacles when there is no need for driving close to them.

Trajectory following The UGV must be able to follow trajectories with sharp turns accurately and should also have the ability to perform neutral turns when this is the only option to proceed driving.

A.8 Route H – Dense forest

Dense forest is a broad category, and the density of the forest will greatly affect the ability of the autonomy system to drive here. A dense forest route needs to be able to be driven manually by an operator; otherwise, the autonomy system can never expect to complete it either. An example of such an environment can be seen in Figure A.3d.

Perception Obstacle maps alone are not enough. The system needs to classify environments into categories, such as ground, trees, rocks, etc. It also needs to evaluate the incline and roughness of the ground. It may be necessary to drive through light vegetation, and the perception system needs to know what vegetation can and cannot be driven through.

Motion planning The environment can be challenging to maneuver in, and the same algorithms used for driving along a vehicle trail might not work for driving between trees. The motion planner must also be able to plan precise paths between narrow openings and close to obstacles, while at the same time being able to maintain safe distances to obstacles when there is no need for driving close to them.

Trajectory following The UGV must be able to follow trajectories with sharp turns accurately and should also be able to perform neutral turns when this is the only option to proceed to drive.

B Platooning controller

The trajectory tracking method for following an AprilTag builds upon the trajectory tracking methods previously described in this report. For more details about this method the reader is referred to [26].

B.1 Path between vehicles

The path between vehicles is defined using a third order Cartesian polynomial. Given the following vehicle (initial state) $\mathbf{x}_i = [x_i, y_i, \theta_i]^T$ and the lead vehicle (final state) $\mathbf{x}_f = [x_f, y_f, \theta_f]^T$ the path between these two vehicles is given as [46, pp. 489–502]

$$x(s) = s^3 x_f - (s-1)^3 x_i + \alpha_x s^2 (s-1)^2 + \beta_x s (s-1)^2 \quad (\text{B.1a})$$

$$y(s) = s^3 y_f - (s-1)^3 y_i + \alpha_y s^2 (s-1)^2 + \beta_y s (s-1)^2 \quad (\text{B.1b})$$

where $s \in [0, 1]$ and represents the normalized position of the vehicle along the path and

$$\alpha_x = k \cos \theta_f - 3x_f \quad \alpha_y = k \sin \theta_f - 3y_f \quad (\text{B.2a})$$

$$\beta_x = k \cos \theta_i + 3x_i \quad \beta_y = k \sin \theta_i + 3y_i \quad (\text{B.2b})$$

The parameter k is a free parameter and influences the path. The parameter can be interpreted as a geometric velocity parameter, as the parameter increases, the longer will the vehicle follow the initial heading direction in the beginning of the path. When following a lead vehicle, the parameter k is modified to scale with both velocity and distance, compared to the original method. This was done to ensure that the following vehicle would avoid sharp turns when driving at higher velocities. The parameter k is defined as

$$d = \sqrt{(x_i - x_f)^2 + (y_i - y_f)^2} \quad (\text{B.3})$$

$$k = s_d d + s_v v_m \quad (\text{B.4})$$

where d is the Euclidean distance between the vehicles, v_m is the estimated velocity of the lead vehicle, and s_d and s_v are scaling parameters for distance and velocity respectively. In the test done during this project, the parameters were set as $s_d = 0.7$ and $s_v = 0$.

Although intuitively, one might expect it to be beneficial to drive straighter paths as the speed increases, this feature was ultimately not used. One issue was that the following vehicle did not turn sharply enough when the lead vehicle was turning in high velocities. Also, there was noise on the estimated velocity, which could have negative effects on the stability of the system.

B.2 Forward velocity control

The forward velocity controller was changed from only using a constant value to having a control law controlling the distance between the following and lead vehicle. This distance is specified using

two parameters: $l_{set,min}$ which is the minimum distance between the vehicles, and t_{dist} which is the desired time the following vehicle should be behind the lead vehicle (in general traffic this is three seconds). This yields a distance set-point as

$$l_{set} = \max(l_{set,min}, t_{dist}v_m) \quad (B.5)$$

where v_m is the estimated lead vehicle velocity. Based on the setpoint the error is calculated as

$$l_e = -\min(0, l_{set} - l_{current}) \quad (B.6)$$

where $l_{current}$ is the current distance along the path between the vehicles (calculated by the path-generating library), and l_e is the distance error. The vehicle should not proceed if the distance between the vehicles is less than the set-point distance, for safety reasons. Therefore, we constrain the error to be non-negative.

The control law for forward velocity is a combination of the estimated lead vehicle (a feed-forward action) and the distance error (a proportional action), as given here

$$v = K_v \left(1 - e^{|l_e|/T_s}\right) v_m + K_p l_e \quad (B.7)$$

where K_v and K_p are control parameters for the feed-forward velocity action and the proportional action, respectively. A damping factor is added to the feed-forward action, to limit this action as the following vehicle gets close to the lead vehicle. This is to avoid discontinuities close to when the distance error becomes zero. The parameter T_s determines the damping of the feed-forward action in meters.

When testing this controller in the project, the following values were used: $l_{set,min} = 6.0$, $t_{dist} = 1$, $K_v = 0.8$, $K_p = 0.7$ and $T_s = 1$.

About FFI

The Norwegian Defence Research Establishment (FFI) was founded 11th of April 1946. It is organised as an administrative agency subordinate to the Ministry of Defence.

FFI's mission

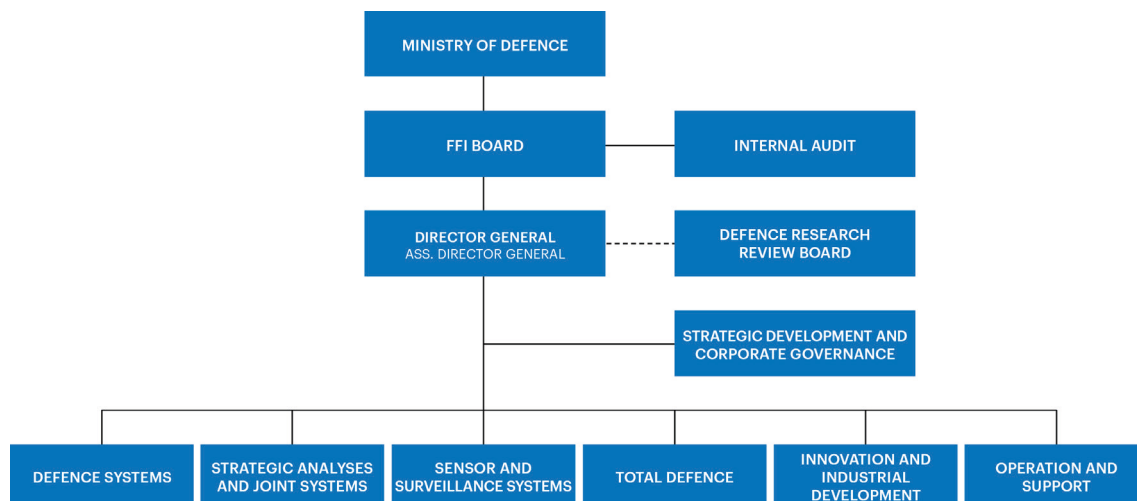
FFI is the prime institution responsible for defence related research in Norway. Its principal mission is to carry out research and development to meet the requirements of the Armed Forces. FFI has the role of chief adviser to the political and military leadership. In particular, the institute shall focus on aspects of the development in science and technology that can influence our security policy or defence planning.

FFI's vision

FFI turns knowledge and ideas into an efficient defence.

FFI's characteristics

Creative, daring, broad-minded and responsible.



Forsvarets forskningsinstitutt (FFI)
Postboks 25
2027 Kjeller

Besøksadresse:
Kjeller: Instituttveien 20, Kjeller
Horten: Nedre vei 16, Karljohansvern, Horten

Telefon: 91 50 30 03
E-post: post@ffi.no
ffi.no

Norwegian Defence Research Establishment (FFI)
PO box 25
NO-2027 Kjeller
NORWAY

Visitor address:
Kjeller: Instituttveien 20, Kjeller
Horten: Nedre vei 16, Karljohansvern, Horten

Telephone: +47 91 50 30 03
E-mail: post@ffi.no
ffi.no/en