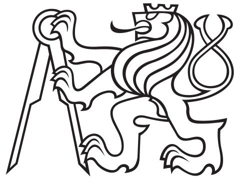
Master Thesis



Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Computer Science

Robot navigation driven by scene semantics

Jan Jirman

Supervisor: prof. Ing. Tomáš Svoboda, Ph.D.
Field of study: Open informatics
Subfield: Software
May 2023

CTU
CZECH TECHNICAL UNIVERSITY IN PRAGUE

MASTER‘S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Jirman Jan** Personal ID number: **466370**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Computer Science**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence**

II. Master’s thesis details

Master’s thesis title in English:

Robot navigation driven by scene semantics

Master’s thesis title in Czech:

Navigace robotu pomocí sémantické segmentace

Guidelines:

In many robotic applications, the robot's goal is to follow a prescribed path close enough. An outdoor deployment can use a sequence of GPS-defined waypoints. A standard geometric cost function penalizes distance from the ideal path. However, the environment may be too complex to follow the path exactly. Driving on a public road, street, or sidewalk could be an example.
Semantic scene segmentation shall provide another cost function combined with the terrain geometry estimation and apriori map information. The student should review existing literature and practical approaches to this problem.
The implementation shall be deployed on an outdoor robot and experimentally verified by driving on a roadside or sidewalk. Given a desired trajectory and allowing a large deviation from it, the robot shall use drivable and safe roads. A dataset usable for developing and testing segmentation models shall be collected during the experiments.

Bibliography / sources:

1. Jingdong Wang, Ke Sun, Tianheng Cheng, Borui Jiang, Chaorui Deng, Yang Zhao, Dong Liu, Yadong Mu, Mingkui Tan, Xinggang Wang, Wenyu Liu, and Bin Xiao. Deep High-Resolution Representation Learning for Visual Recognition. IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI), March 2020. https://github.com/HRNet/HRNet-Semantic-Segmentation
2. Jiacong Xu and Zixiang Xiong and Shankar P. Bhattacharyya. A Real-time Semantic Segmentation Network Inspired from PID Controller. arXiv.2206.02066, https://arxiv.org/abs/2206.02066 https://github.com/XuJiacong/PIDNet
3. Awet Haileslassie Gebrehiwot, Patrik Vacek, David Hurych, Karel Zimmermann, Patrick Perez, Tomáš Svoboda. Teachers in concordance for pseudo-labeling of 3D sequential data. https://doi.org/10.48550/arXiv.2207.06079, https://github.com/ctu-vras/t-concord3d
4. Dosovitskiy, A., Ros, G., Codevilla, F., Lopez, A. & Koltun, V.. (2017). CARLA: An Open Urban Driving Simulator. Proceedings of the 1st Annual Conference on Robot Learning, in Proceedings of Machine Learning Research 78:1-16 Available from https://proceedings.mlr.press/v78/dosovitskiy17a.html.
5. A. Giusti et al., "A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots," in IEEE Robotics and Automation Letters, vol. 1, no. 2, pp. 661-667, July 2016, doi: 10.1109/LRA.2015.2509024.
6. G. Kahn, P. Abbeel and S. Levine, "LaND: Learning to Navigate From Disengagements," in IEEE Robotics and Automation Letters, vol. 6, no. 2, pp. 1872-1879, April 2021, doi: 10.1109/LRA.2021.3060404.

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Name and workplace of master's thesis supervisor:

prof. Ing. Tomáš Svoboda, Ph.D.   Vision for Robotics and Autonomous Systems FEE

Name and workplace of second master's thesis supervisor or consultant:

Date of master's thesis assignment: **09.02.2023**   Deadline for master's thesis submission: **26.05.2023**

Assignment valid until: **22.09.2024**

prof. Ing. Tomáš Svoboda, Ph.D.
Supervisor's signature

Head of department's signature

prof. Mgr. Petr Páta, Ph.D.
Dean's signature

III. Assignment receipt

The student acknowledges that the master's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the master's thesis, the author must state the names of consultants and include a list of references.

.

Date of assignment receipt

Student's signature

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Acknowledgements

I would like to thank my thesis supervisor, prof. Ing. Tomáš Svoboda, Ph.D., for all the support, consultations, and comments provided to help me with this work. I would also like to thank Ing. Martin Pecka, Ph.D., and Ing. Tomáš Rouček for valuable consultation and advice on the technical aspects of this work and debugging hardware issues.

Declaration

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, May 26, 2023

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uved lveškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, 26. května 2023

Abstract

In many robotic applications, the robot's goal is to follow a prescribed path close enough. An outdoor deployment can use a sequence of GNSS-defined waypoints. A costmap is used for robot navigation, where the robot looks for the cheapest path. A standard geometric costmap penalizes distance from the ideal path. However, some environments require a special path dictated by traffic regulations or the robot's safety. Driving on a public road, street, or sidewalk could serve as an example. Semantic scene segmentation shall provide another cost function to be combined with the geometric one. The initial phase of the project focuses on driving on public roads or sidewalks – stable driving on a roadside. The scene segmentation may include multimodal data.

Keywords: Visual semantic segmentation, Navigation

Supervisor: prof. Ing. Tomáš Svoboda, Ph.D.

Abstrakt

V mnoha robotických aplikacích je cílem robotu následovat předepsanou dráhy dostatečně zblízka. Venkovní nasazení může využívat sekvenci GNSS bodů na trase. Pro navigaci robotu se využívá cenová mapa, ve které robot hledá nejlevnější cestu. Standardní geometrická cenová mapa penalizuje vzdálenost od idealní dráhy. Nicméně, v některých prostředích je vyžadována speciální trasa diktována dopravnímy předpisy nebo bezpečností robotu. Pohyb na veřejných silnicích, cestách nebo chodnících může sloužit jako jeden z příkladů. Sémantická segmentace scény může poskytnout jinou cenovou funkci, aby se mohla zkombinovat s tou geometrickou. Počáteční fáze tohoto projektu se zaměřuje na pohyb na veřejných silnicích a chodnících – stabilní pohyb po kraji cesty. Sémantická segmentace může využívat multimodálních dat.

Klíčová slova: Vizuální sémantická segmentace, Navigace

Překlad názvu: Navigace robotu pomocí sémantické segmentace

Contents

1 Introduction 1

2 Related work 3

3 Experimental platform description 5

3.1 Hardware platform description 5
3.2 Robot Operating System 6
3.3 Existing control architecture 7

4 Road Navigation Module 9

4.1 Version with GNSS and OpenStreetMaps only 9
4.2 Visual navigation 11
4.2.1 Semantic segmentation 12
4.2.2 Contour extraction from segmented image 13
4.2.3 Contour projection to ground-plane 16
4.2.4 Contour filtering method 17
4.2.5 Contour map – Temporal consistency 17
4.2.5.1 Grid map bayesian update 19
4.2.5.2 Contour observation model 20
4.2.6 Costmap 21
4.2.7 Additional grid map channel – Forbidden areas 22
4.2.8 Planning around the visible area 25
4.3 Encountered problems 25
4.3.1 Top down contour map transformations 25
4.3.2 Costmap resolution is too small 26
4.3.3 Strong priors in the lane detection 27
4.3.4 Vision module is slow 27
4.3.5 Compass guiding the robot away 28
4.3.6 Computation power issues – the overall latency 28
4.4 Implementation details 29
4.4.1 LLA / UTM / UTM_local coordinate frames 29
4.4.2 Contour extraction ROI 29
4.4.3 Measuring the camera delay 30

5 Experiments and results 33

5.1 Contour filtering 33
5.2 Choosing the appropriate profile function 35
5.3 Costmap strength 36
5.4 Mission navigation with replanning 37
5.5 Quantitative results 38

6 Conclusion 41

6.1 Code repository 41
6.2 Future work 41

References 45

Figures

3.1 Robot fleet, from the left: 2x Spot with arm, 2x Spot without arm, TRADR, Husky 5
3.2 Gridmap of terrain and obstacles. The robot is placed near an open door in a hallway. Green: terrain map, blue: distance to the nearest obstacle (distance encoded as `z` coordinate), purple: distance to the nearest “NoStep” region (distance encoded as `z` coordinate) 6
3.3 Spot’s sensor suite illustration – five internal grayscale cameras, point-cloud surrounding the robot, green terrain map grid, GNSS localization on a map 7
3.4 Control architecture diagram: Road navigation module interfaces with the other parts of the system. 8

4.1 Cost profile used for OpenStreetMap-based navigation. The plot shows a cost as a function of distance d_p from the border of the road, with labels of points or areas of interest. 10
4.2 Demonstration of costmap and planned path (green line). Left: Without Road Navigation Module, Right: With Road Navigation Module; Purple denotes highest cost, blue denotes medium cost, yellow to light green denotes very low cost, which corresponds to the area in which the desired path should generally follow, orange denotes zero cost and appears only where there is no data 11
4.3 Several chosen frames demonstrating segmentation results on our dataset. Left column: Good and relatively clean and correct discrimination between various classes. Further away from the camera, parts of the sidewalk are erroneously classified as a road; however, this is not a big issue. Right column: examples of less successful results, which produce a lot of noisy data to the downstream modules. 14
4.4 A selected image frame and the contours detected in it. 16
4.5 Transformation from the contours detected in the image (top; purple and pink lines) to the world coordinate system, placed on the ground plane (bottom; the grid represents the ground plane). The red dotted lines mark the ROI area (more in sec 4.4.2) for the contour extraction. Contours outside of this area are ignored. 18
4.6 Probability of detected contour at location l given distance d_p from the closest real border of the road 20
4.7 Currently used profile – a cost incurred for the planner given a distance to the nearest contour cell. 22
4.8 Computation of various top-down views of the robot. Each row shows a different scenario; the photos at the bottom are the views front the front-facing camera at those respective scenarios. Left column: Contour probability channel of the grid map. Middle column: Forbidden areas channel of the grid map. Right: Both channels combined into one costmap. 23
4.9 Top: Robot plans a path (green) alongside the sidewalk, rather than direct – shortest Euclidean path. The costmap shows one side of the road border with relatively low costs (white and close to white) and the other side with higher costs (the more saturated red, the higher the cost is). Without this last step of adding a channel for forbidden areas, both sides of the border would be more less white, and the path would have a tendency to be more straight, crossing the road border. Bottom: Same situation from the viewpoint of the robot. 24

4.10 Planned path (green) from the robot to the goal (red box). Left: The path ignores corridors and barriers in the detected area because the path planner managed to "slip" outside of the detected area near the robot. Right: This time, the path follows the shape of the road, as it cannot cross the detected area boundary near the robot. The boundary can be seen as darker red (= higher cost) cells in the right image. 25
4.11 Merged top-down contour map - the probability that there exists a contour at a given world location. Left: Images from the camera are out of sync by about one second with the transformation data of the robot. Right: Images and transformations synced using appropriate `CAMERA_DELAY_S`. Both images represent the same scenario, with the robot at roughly the same physical position. We should see a relatively straight road with a left right-angle turn at the end, which can be clearly observed in the right image. . . . 26
4.12 Two selected images with very incorrect detections of lane borders. All detected lane borders face the forward direction. . . . 27
4.13 Robot absolute heading misaligned due to a compass error. The roadside is detected correctly (right image, green and yellow contours, pointed to by an orange arrow). The same roadside is shown in the top-down view (left image, same contours, again, pointed to by an orange arrow). In the top-down view, the detected contour should be exactly aligned with the beige-green interface on the map. The red dotted lines mark the ROI area (more in sec 4.4.2). . . . 28
4.14 Rays projected from the camera onto a ground plane estimate. Purple: Ray pair (of two neighboring pixels in the camera) whose distance d_α is higher than the ROI threshold. Green: Ray pair (of two neighboring pixels in the camera), with the same angle α, whose distance d_α is within the ROI threshold. Red: Area which is considered a reliable region of interest. . . . 30
4.15 Left: Rendered horizon and heading lines do not line up with the image – the transformation used and the photo itself are each from a different time; Right: Rendered horizon and heading lines are in sync with the picture. . . . 31

5.1 Several values of `MIN_CONTOUR_LENGTH_PX` parameter. Top left: 0px – any detected shape passes the filter. Notice several small contours around the legs of the bench; Bottom left: 150px – The results are over-filtered. Contour along the length of the sidewalk did not get detected as one continuous piece, but rather two consecutive. As such, it was not long enough to pass; Right column: 25px – Balance between removing very small details but keeping many lines, which may be good contour candidates. This experiment was executed with an image resolution of 640x192. The red dotted lines mark the ROI area (more in sec 4.4.2). . . . 34
5.2 Contour detections before max curvature filtering. Notice the 90-degree bend in one curve between two straight segments (pointed to by the left arrow) and an odd-shaped blob containing many sharp angle bends (segment pointed to by the right arrow), which is caused by an incorrect detection. . . . 34
5.3 Left: parameter `MAX_ALLOWED_ANGLE` = 11.2 degrees – we can observe contours split into many different sub-contours. Only the very, very straight ones remained in one piece. Right: parameter `MAX_ALLOWED_ANGLE` = 90 degrees – the more contour stay in one piece for longer. In both images, each contour is depicted using its own hue. The red dotted lines mark the ROI area (more in sec 4.4.2). . . . 35

5.4 Illustration of sharp-angle-splits (at positions of blue arrows) along a relatively smooth but discrete contour (red), which is comprised of points (black) projected from the pixel grid with only a small deviation from an optimally fit contour (green). 36
5.5 Planned path (green) from the robot to the goal while utilizing the costmap. White parts of costmap – low cost; Red parts – high cost; Left: The `COST_GAIN` is set to a low value of 1. Notice that the planner cuts across the road border when the path starts to be much longer than the direct Euclidean path; Right: `COST_GAIN` is set to a much higher value of 40. The planner plans the path along the detected road border for a much longer span, essentially until the end of the visible area. *Side note: An observant reader may notice the misalignment between the OpenStreetMap overlay and the heading of the detected road. This is caused by a magnetic compass issue, discussed in section 4.3.5.* 37
5.6 Sequence of plans to complete the mission – move from the top-left corner to the bottom-right, using only roads or sidewalks. Top: Costmap built from the local information from the vision system; the rest is set to a default constant value. Bottom: Sequence of plan changes as the robot moves along the planned path and new parts of the world come into view. The robot first follows the green plan, then blue, purple, orange, and at the end, the very short green. *Not every plan update is shown in the figure, for the sake of clarity* 39
5.7 Cost values at the robot's position. The value at the exact position of the robot is shown (blue), as well as mean values in the robot surrounding area (orange, green). Several events that happened during the experiment are marked (in red) at their respective times of occurrence. A – robot drove very near the road edge; B – robot crossed the road; C – robot took a shortcut; D – robot vision distracted by a distracted driver. These events are described in more detail in the text. 40

6.1 The robotic dog in the real world environment – standoff with a meat-based dog. 42
6.2 Revised control architecture diagram: Road navigation module interfaces with the other parts of the system. 43

Chapter 1
Introduction

In many robotic applications, the robot's goal is to follow a prescribed path close enough. An outdoor deployment can use a sequence of GNSS-defined waypoints. A costmap is used for robot navigation, where the robot looks for the cheapest path. A standard geometric costmap penalizes distance from the ideal path. However, some environments require a special path dictated by traffic regulations or the robot's safety.

When navigating in the environment, the robot may encounter obstacles that it needs to avoid (and thereby deviate from the original prescribed path) or areas that require special handling in order to obey the rules of that environment. One example is public roads or pathways, where the robot should keep to the side of the road when crossing, wait until the road is clear of cars, and then minimize the time spent on the road. When the robot is on a pathway, it should keep to the right where possible.

The purpose of this work is to guide the robot in the aforementioned environment of public roads. The robot navigation is achieved indirectly via generating a `costmap`, which is passed to a path planner module that generates a path for the robot to follow. This `costmap` penalizes the planner for choosing a path leading through undesired positions, such as the middle of the road. To generate this `costmap`, the algorithm uses the robot's front-facing camera; however, in the future, more sensor types will be added and explored, such as lidar inputs.

It is expected that the input data gathered from the sensors will be imperfect, as well as its subsequent analysis, such as semantic classification or object detection. One goal of this work is to utilize this imperfect input and guide the robot correctly in spite of this issue.

Chapter 2

Related work

Autonomous navigation on public roads is a frequently covered topic, especially in the domain of autonomous cars. There are various publications concerning the semantic segmentation [1–5] of datasets obtained while driving, such as KITTI [6] or cityscapes [7]. Semantic segmentation, however, is only one part that we need to achieve the goal. We need to navigate the robot according to what it sees in the camera. A similar work [8] by Gregory Kahn, et al. demonstrates the use of human operator disengagements as input data for learning how the robot should react correctly the next time. Alessandro Giusti et al. present in their work [9] a deep neural network used to classify a forest path direction. This classification translates directly into control inputs, whether the robot should steer left, straight, or right.

Neither of these works is directly comparable to this because this work aims to let the robot navigate according to its mission requirements and intervene or guide the robot only in situations when it would become dangerous for the robot to continue (such as navigation on the road). The desired output of this work is virtual barriers where the robot should not move. When these barriers are not crossed, the robot is free to navigate according to some other decision-making process.

Chapter 3

Experimental platform description

3.1 Hardware platform description

We have multiple robots, which are designed to be somewhat similar, both the software stack as well as the computational and sensory systems. The overall design supports the easy transfer of code from one robotic platform to the other. One such platform that I am using for this project is Boston Dynamics Spot [10]. Other available platforms include Husky [11] and TRADR [12]. Relevant sensor suites of these robots include cameras that cover 360° around

Figure 3.1: Robot fleet, from the left: 2x Spot with arm, 2x Spot without arm, TRADR, Husky

the robot, a set of stereo cameras to build a depth map, a lidar to build a point cloud, and GNSS antennas for global localization. However, there are

differences in the particular sensors used or data quality on each robot. With this in mind, the soft goal of this project is to be robot-independent, but in case some specific sensors or processing is needed, the primary robotic platform of choice is Spot, without the need to support all other platforms simultaneously.

Unlike the other robots, the Spot robot provides some already preprocessed data for the control system, such as a grid map of nearby terrain and obstacles. However, this preprocessed data was not used in this thesis.

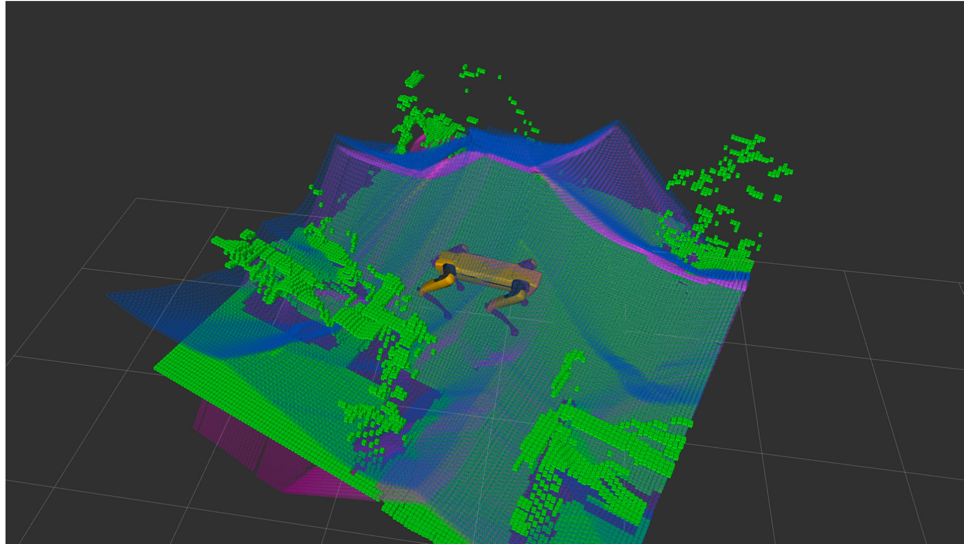


Figure 3.2: Gridmap of terrain and obstacles. The robot is placed near an open door in a hallway. Green: terrain map, blue: distance to the nearest obstacle (distance encoded as `z` coordinate), purple: distance to the nearest "NoStep" region (distance encoded as `z` coordinate)

3.2 Robot Operating System

All our robots use the framework of the Robotic Operating System (ROS) [13]. The Robot Operating System is an open-source set of software libraries and tools that help with building robotic applications.

For the testing and development process, I often use ROS Bag [14] files. ROS Bags are packaged recordings of behavior from previous robotic runs. These recordings include sensor data, such as camera frames, lidars, or GNSS positions. Aside from raw sensory inputs, various internal states and decisions of the robot are recorded. Using this recorded data from previous mission runs or challenging environments significantly improves development time, as it relieves the need for using physical hardware and robotic platform every single time.

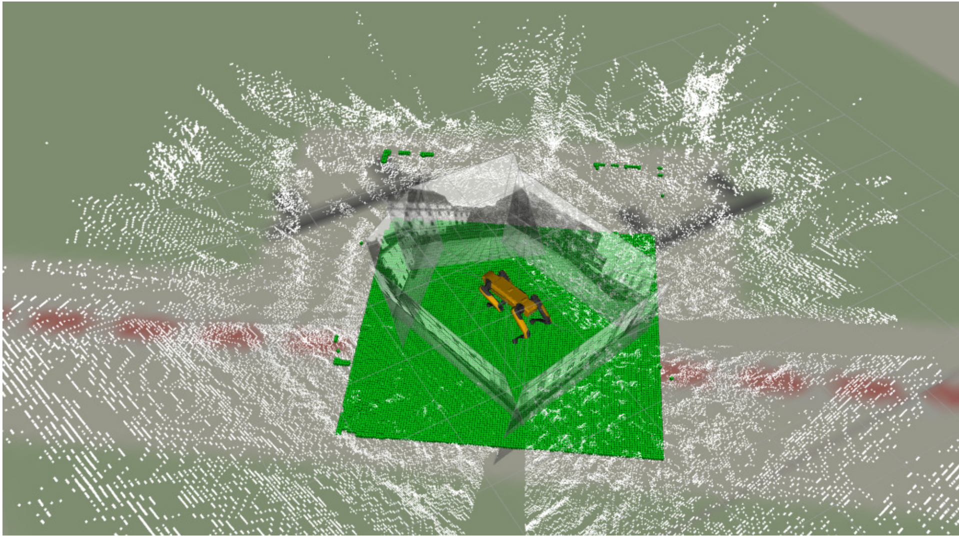


Figure 3.3: Spot's sensor suite illustration – five internal grayscale cameras, point-cloud surrounding the robot, green terrain map grid, GNSS localization on a map

3.3 Existing control architecture

The robot's control system is built from multiple independent modules that communicate using ROS. Many of these modules were already developed and prepared from previous work and use of the robot. I had to connect my module to the already existing architecture. Figure 3.4 illustrates the high-level overview of the used modules.

The `Road Navigation` module (my work) reads raw sensor inputs and data already preprocessed by other modules (such as information about the robot's current location, which is a sensor fusion between GNSS and visual odometry). The output of this module is a costmap in the near vicinity of the robot. This costmap is used by a planner module that was already developed. Developing my own planner module is outside of the scope of this thesis. However, I needed to tune this planner module substantially to obtain the results I was expecting. The next module in the chain is a path tracker module, which also already exists and is outside of this project's scope. This path tracker module uses the path from the planner as its input and generates locomotion commands for the robot.

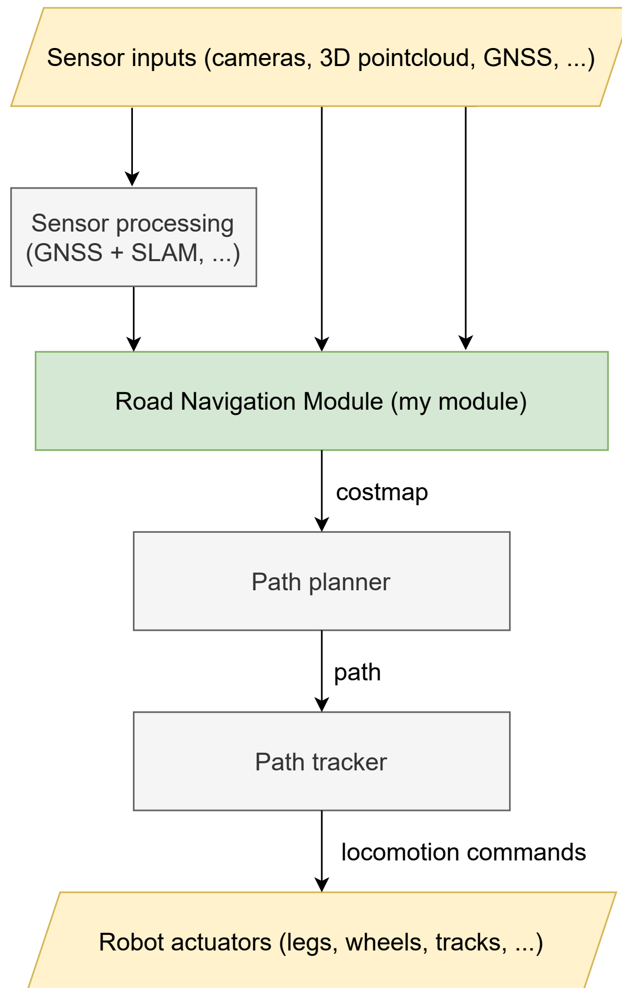


Figure 3.4: Control architecture diagram: Road navigation module interfaces with the other parts of the system.

Chapter 4

Road Navigation Module

4.1 Version with GNSS and OpenStreetMaps only

As a starting point, I have created a module that, given the robot's location, generates a costmap (in the form of a labeled point cloud) using nearby roads and pathways loaded from OpenStreetMaps [15]. The cost for each point p is assigned using a pre-determined profile, which is a function of distance d_p to the nearest border of a road.

$$d_p = \left\| p - nbp_p \right\| .$$

The symbol nbp_p denotes the nearest border cell to the point p – a cell representing part of the road border, one that is the closest to the point p of all road border cells. Please note that direct evaluation of calculating the distance between each pair p_i and border cell would be incredibly inefficient. To speed up the computation, a binary image representing a boolean map representing whether a road border is present at each particular cell, and then the Euclidean distance transform function [16] is applied. Note that these cells in the boolean map images are referred to as pixels interchangeably in this work.

The module uses a profile (see figure 4.1) that returns low cost values for points on either side of the road, and the cost gradually increases to a maximum towards the middle of the road. For locations outside the road polygon, there is a spike for the locations near the road (as a ditch may be present) and then gradually levels to a constant "background value." More specifically, this profile is a poly-line that connects chosen points and is constant on the range between infinity and the first defined point on either side. Figure 4.1 illustrates the costs provided by this profile, while table 4.1 shows specific points that were chosen. Cost output is always in the range of 0 to 1. However, there is parameter `COST_GAIN`, which multiplies the cost output of this module just before it sends it off to the next (planner) module. This parameter is important for tweaking the overall significance of the road borders for the robot's mission. The setting of this and other parameters is discussed more in section 5. In short, decreasing the value of this parameter will make the planner ignore the roads more.

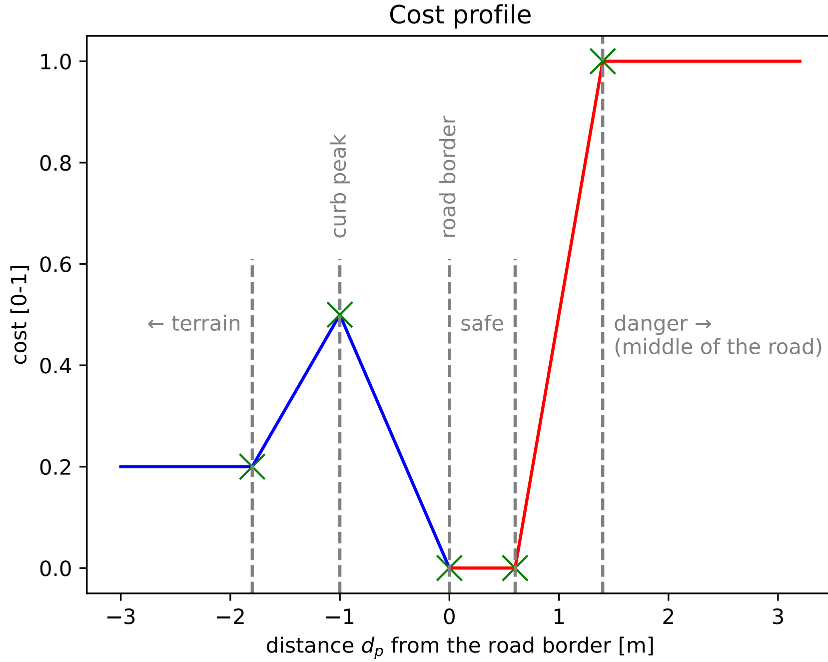


Figure 4.1: Cost profile used for OpenStreetMap-based navigation. The plot shows a cost as a function of distance d_p from the border of the road, with labels of points or areas of interest.

Parameter	**Distance d_p**	**Cost**	***Note***
Terrain cost	$-\infty$	0.2	*Terrain*
Curb end	-1.8m	0.2	
Curb peak	-1m	0.5	
Roadside	**0m**	**0**	
Corridor	0.6m	0	*Safe zone from the roadside up to this point*
Road danger	1.4m	1	*Start of the danger zone*
Road cost	∞	1	*Danger zone – middle of the road*

Table 4.1: Parameters used in the described cost function. Each parameter is a cost-distance pair (point).

This profile, however, only captures a few real-life scenarios. It would be relatively easy to set up multiple different profiles and auto-select which one to use based on the general GNSS location of the robot. But this belongs to the deployment phase of this module. For the purposes of development and demonstration of functionality, the single profile is sufficient.

To demonstrate the functionality of the aforementioned module, I offer the following scenario: In figure 4.2, we can observe that with the `Road Navigation Module` enabled, the planned path keeps to the side of the road for as long as possible, then quickly crosses the road to minimize time spent in the middle and continues to the goal on the other side; For better clarity, used cost profile slightly penalizes non-road areas as if it was terrain in which the robot moves slower. This is a demonstration that the module I

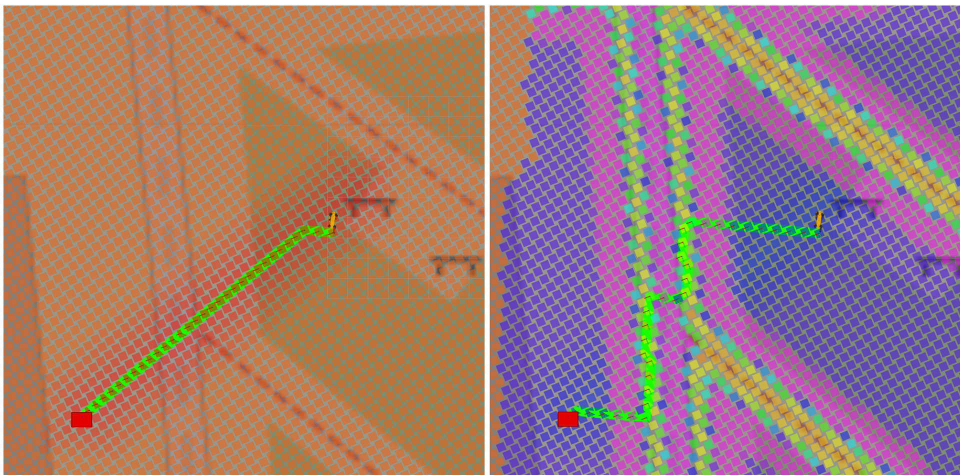


Figure 4.2: Demonstration of costmap and planned path (green line). Left: Without Road Navigation Module, Right: With Road Navigation Module; Purple denotes highest cost, blue denotes medium cost, yellow to light green denotes very low cost, which corresponds to the area in which the desired path should generally follow, orange denotes zero cost and appears only where there is no data

have built has the ability to control the physical robot's behavior (locomotion) and successfully integrates with the existing control stack.

4.2 Visual navigation

The overall goal of the thesis is to rely on visual data from the robot's cameras, not on the GNSS localization. I have used already existing software [1] for visual semantic segmentation. The segmentation output of this software misclassifies some parts of the image in many frames (more detail in section 4.2.1 and figure 4.3). Nevertheless, it provides results that are good enough for our purposes. One of the goals of this project is to be able to navigate the robot even with imperfect input data and handle discrepancies between the real world and what is detected. The real world is very messy, and there

won't ever be segmentation software achieving 100% accuracy in every single possible scenario.

The system I have developed could be described in several stages. Each stage processes some data and provides its output as input to the next stage. The following sections discuss the particulars of each different stage. The stages are in the code organized as follows:

1. **Semantic segmentation**
   This stage performs semantic segmentation of the image and finds and detects contours in this segmentation. Section 4.2.1 describes the details of the visual segmentation software used and why I decided to use this particular tool. Section 4.2.2 describes the contour extraction process.

2. **Project contours to ground-plane**
   Once the contours are isolated in the image frame, we need to project them to the ground-plane coordinate system for further processing. This transforms all the contours to the "top-down view". Section 4.2.3 describes how.

3. **Filter contours**
   Semantic segmentation introduces some noise to its output, and as such, not all contours accurately represent the real-world scenario. At this stage, we try to filter some to reduce noisy data for the following stages. Section 4.2.4 describes the filtering method used.

4. **Merge top-down contours** to a bigger map
   For various reasons, it is beneficial to consider more than just one image frame at a time. This is done by building a top-down map of all previously seen contours. Section 4.2.5 describes this process and reasons in more detail.

5. **Generate costmap**
   The final stage constructs a costmap from all the information gathered by the previous stages. This costmap is then used by a planner module, which generates a path for the robot to follow and thus guides the robot's movement. This is discussed in the section 4.2.6.

4.2.1 Semantic segmentation

There are several different visual segmentation tools publicly available [1–5], as well as tutorials on how to write a new one from scratch [17]. I chose to use software `Autonomous-Vehicle-Environment-Perception` [1] available on GitHub. This software also ships with pre-trained weights for the neural networks used, so I didn't have to train them from scratch. This code was able to process images using Python 3 and had all the pre-trained weights available. Originally, this provided several different outputs for each image frame.

- **Semantic segmentation classes, one for each pixel** – This is the most important part I was interested in. Combined together with the

lane estimation, I was expecting good results in contour detection of the drivable (walkable) space.

- **Estimated depth map** – provided by a network designed by [18]. The same network also outputs the semantic classes.
- **Lane detection** – in the background, a work by [19] was used for lanes estimation.
- **Objects in the scene** – this part was using YOLO [20] network to detect objects such as traffic lights (including the color it was signalizing), signs, cars, and similar objects.

However, in the course of my thesis, I had to modify this module, as discussed in sections 4.3.3 and 4.3.4.

I have tested this segmentation tool on our dataset, as shown in figure 4.3, and there are several issues observed relatively often. Most notably, some parts from the terrain class get misclassified as roads or sidewalks. Less frequently observed misclassification was the other way around – parts of the sidewalk class getting classified as terrain. However, the results seemed satisfactory in spite of these problems, especially with the thesis goal in mind that the robot should be able to deal with the somewhat noisy input information.

4.2.2 Contour extraction from segmented image

The visual semantic segmentation module used is geared more toward automotive applications. Among other things, this is apparent from the selection of semantic classes that this module classifies into the following classes. Only some classes are interesting for this work, marked in the list; all other classes are ignored.

- **CLS_ROAD** - *area class*
- **CLS_SIDEWALK** - *area class*
- **CLS_BUILDING** - *area class*
- **CLS_WALL** - *area class*
- **CLS_FENCE** - *area class*
- CLS_POLE
- CLS_TRLIGHT
- CLS_VEGT
- **CLS_TERR** - *area class*
- CLS_SKY
- CLS_PERSON
- CLS_RIDER
- CLS_CAR
- CLS_TRUCK
- CLS_BUS
- CLS_TRAIN
- CLS_MCYCLE
- CLS_BCYCLE

We don't need many of these classes, such as poles or traffic lights. We are only interested in classes representing areas on the ground, such as `road`, `wall`,

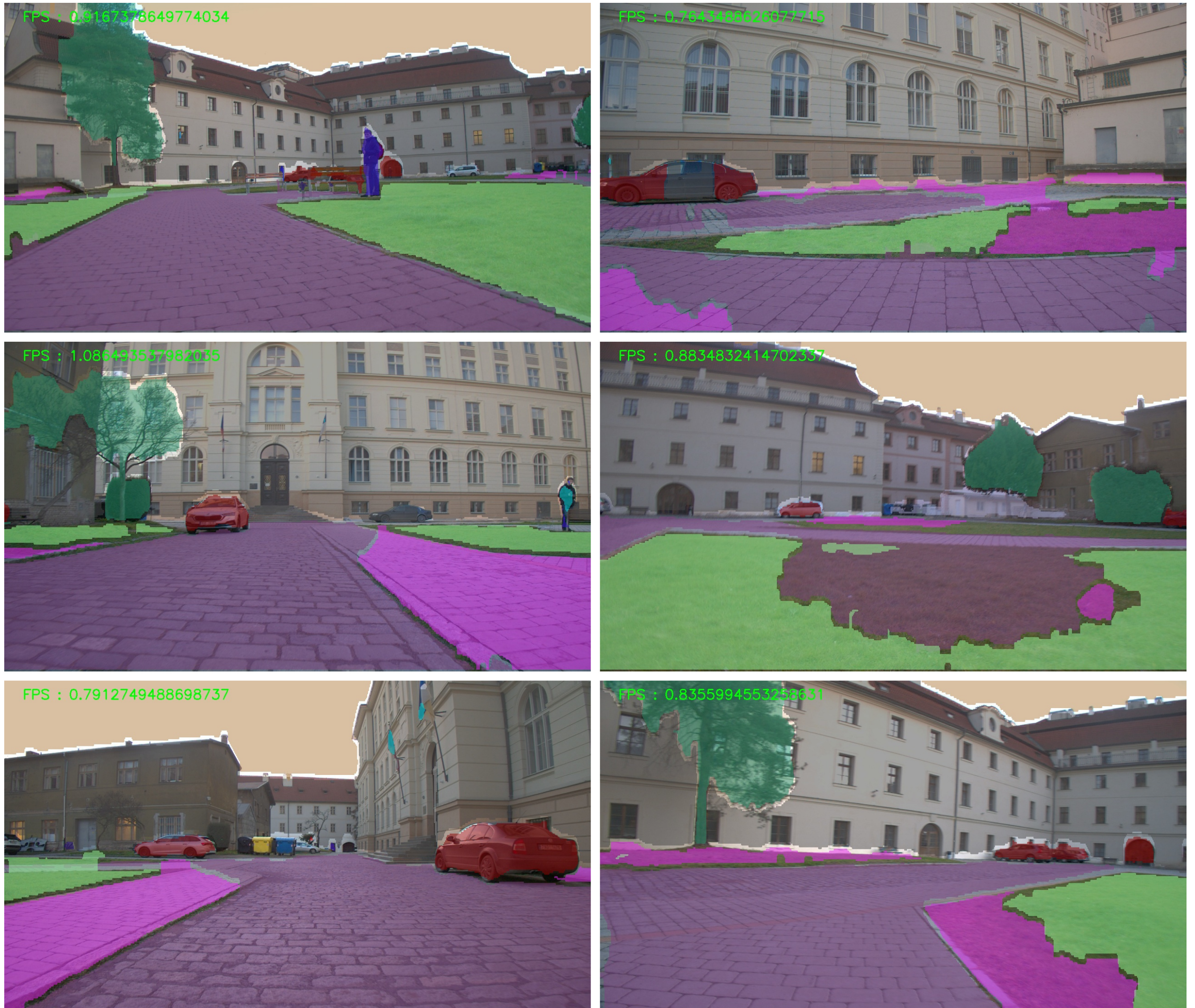


Figure 4.3: Several chosen frames demonstrating segmentation results on our dataset. Left column: Good and relatively clean and correct discrimination between various classes. Further away from the camera, parts of the sidewalk are erroneously classified as a road; however, this is not a big issue. Right column: examples of less successful results, which produce a lot of noisy data to the downstream modules.

or `terrain`. In fact, what we need are not even pixel areas classified with a specific class but rather borders of neighboring classes, such as the border where `road` and `terrain` meet. Only the borders between "area" classes are taken into account. Meaning that the border between `road` and `car` class is irrelevant, as the presence of a car does not imply the end of the road. Quite the contrary, there is a high chance that underneath a car, there is also a road, but it is just obscured in the picture. I call these borders "contours". The following pseudocode demonstrates the process of contour extraction. Figure 4.4 shows an image frame with the contours detected.

Pseudocode – contour extraction from segmentation mask

```
for each relevant class c:
  let mask m = segmentation output for c

  # Noise filtering
  se = [1  1  1]  # structuring element for the morphological operations
       [1  1  1]  # simple max/min from the 3x3 area is used
       [1  1  1]
  m = dilation(m, se, iterations=2)
  m = erosion(m, se, iterations=3)

  # Find the mask edge by convolving the mask with a laplacian kernel
  kernel = [0  1  0]
           [1 -4  1]
           [0  1  0]

  let edge image e = convolve(m, kernel)




for each pair of relevant classes c1 and c2:
  # Find the overlap near the edge area of the two classes
  #   -> First, expand them a little so that they can overlap

  # e1, e2 are the edge images, with class labels c1 and c2, respectively

  e1 = dilation(e1, se, iterations=2)
  e2 = dilation(e2, se, iterations=2)

  e = e1 * e2  # e is true whereever both e1 and e2 is true.

  # Find the core of the contour and ignore bigger "blobs"
  e = thin(e, max_iterations=25)

  # Connect bitmap contour pixels and vectorize
  contours = find_iso_contours(e, fully_connected="high")

  for each contour in contours
    if length(contour) > MIN_CONTOUR_LENGTH_PX
      yield contour
```

`MIN_CONTOUR_LENGTH_PX` is a parameter of the algorithm used as part of noise-filtering. The exact effects and setting of this parameter are described in detail in the section 5.

Figure 4.4: A selected image frame and the contours detected in it.

4.2.3 Contour projection to ground-plane

All contours are then transformed from an image frame into a robot odometry frame and projected on the ground plane estimate underneath the robot. All of the following processing is done using this 2D top-down view representation.

The module receives `camera_info` and `TF` messages from ROS. The first contains information about the image sensor used, such as its focal length, distortion correction parameters, and others. The most important parameter obtained from this message is the camera projection matrix P. This matrix contains all the information we need to project a 3D ray R_{uv} corresponding to a particular pixel uv, according to the pinhole camera model. Let's define ray using a parametric equation

$$R_{uv}(t) = \vec{O_r} + \vec{D_r} \cdot t \,,$$

where $\vec{O_r}$ is the origin vector of the ray, and $\vec{D_r}$ is its direction. When inversing the pinhole camera projection, we obtain

$$\vec{D_r} = \begin{pmatrix} u - c_x \\ v - c_y \\ 1 \end{pmatrix} / \begin{pmatrix} f_x \\ f_y \\ 1 \end{pmatrix} .$$

The symbol / denotes an element-wise division. We should normalize the vector $\vec{D_r}$ before further processing. The origin vector $\vec{O_r}$ is equal to $\vec{0}$ as the ray originates from the center of the camera. Both vectors $\vec{D_r}$ and $\vec{O_r}$

together fully define the ray R_{uv}, which we need. It is currently in the camera coordinate system; however, we might need to transform it later.

The `TF` messages contain rigid transformations between various frames on the robots, among others, physical camera location with respect to the robot body and estimated position and orientation of the ground plane underneath the robot. Using the relative transformation between the ground plane estimate *gpe* and the camera, we can find the intersection where the ray R_{uv} hits the *gpe* plane. For the plane, it holds that each point $\vec{x}$ that lies on the plane satisfies the equation $\vec{n} \cdot \vec{x} = z$. The vector $\vec{n}$ is the normal vector of the plane, and z is a shift along this vector. Now, combining this with the ray equation, we obtain

$$\vec{n} \cdot \left(\vec{O_r} + \vec{D_r} \cdot t\right) = z\,,$$

solving for t,

$$t = \frac{z - \vec{n} \cdot \vec{O_r}}{\vec{n} \cdot \vec{D_r}}\,.$$

If $t > 0$, we can substitute it to the ray equation and obtain the one intersection point between the ray R_{uv} and the plane *gpe*; otherwise, they do not intersect. We apply this method to each pixel in each contour detected in the previous stage. This gives us all the contours we have detected from the "top-down" view, "painted" on the *gpe*, as shown in the figure 4.5. This transformation assumes flat terrain. The transformation gives use incorrect results when the detected contours in the image are not on the ground plane but at a different height (such as the top of a wall or the sides of a staircase). It does not pose an issue, however, because *gpe* generally represents areas in the near vicinity well, and we are interested only in the contours which are near us.

4.2.4 Contour filtering method

A set of isolated contours is further filtered to remove the noisy detections. More specifically, the curvature of the contours is observed, and when the angle between two consecutive line segments is over a threshold `MAX_ALLOWED_ANGLE` (contour is not smooth, alongside the probable road, but zig-zag-ey), it is split into two at that offending place (or places). Sometimes it is desired for contours to have sharp edges – for example, 90-degree turns. That's why high curvature by itself is not a reason to discard the contour right away. Nevertheless, after this step, contours that are too short are removed. This, in turn, removes all contours which are very zig-zag-ey as they will be cut into many (too) short segments with acceptable curvature.

4.2.5 Contour map – Temporal consistency

This section discusses the importance and method of fusing contours together from multiple frames – multiple measurements, each at a different time. Information obtained from the previous stage is gathered only from a single frame. It can still be noisy, in spite of multiple attempts to clean the data as

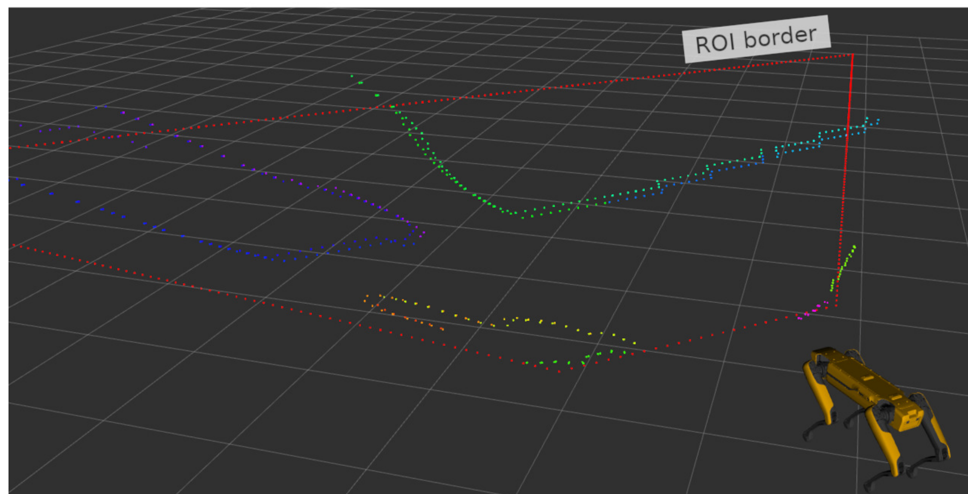


Figure 4.5: Transformation from the contours detected in the image (top; purple and pink lines) to the world coordinate system, placed on the ground plane (bottom; the grid represents the ground plane). The red dotted lines mark the ROI area (more in sec 4.4.2) for the contour extraction. Contours outside of this area are ignored.

much as possible. Some contributing factors to the occurrence of this noise include incorrect segmentations by the underlying semantic segmentation tool, which may not be trained on a domain that perfectly represents the deployment environment or the robot moving too quickly, and the processed image being blurred and hard to read. For this reason, it is important that the robot does not act solely on the observation provided from the latest frame but fuses contours from multiple consecutive frames.

The way I decided to tackle this problem is by building a probabilistic map, representing a probability that at each particular discretized location cell, there is a contour – a border between traversable and untraversable terrain.

When updating this map with a new observation, the first step is to translate the previous map correctly (this had its own issues, see section 4.3.1) and then perform a Bayesian update (described in section 4.2.5.1) according to the model of measurements (described in more detail in section 4.2.5.2). This procedure is known as a grid map approach.

Assuming that the observation model accurately represents error characteristics of the semantic segmentation tool, the resulting grid map will stabilize in the state, which shows a high probability at locations where is the real road border and a low probability in all other (explored) areas. This is achieved because observations that are consistent across multiple frames have one thing in common – the underlying feature in the real world – the side of the road. These correct observations will reinforce each other, while erroneous observations will appear at random locations (different location in each frame) and cancel each other out.

Another advantage of the grid map approach, except for the temporal consistency, is that it enables the later steps of this computation pipeline to reason also about areas that are not currently visible in the camera view.

4.2.5.1 Grid map bayesian update

To update the grid probabilities, the module uses Bayesian update [21]. In the beginning, the probability $P(l = border)$ is initialized to 0.5. In each step, the posterior probability (after the measurement) is calculated from the new measurement and an apriori probability at that location. The apriori probability used is the posterior probability from the previous step. The full equation is shown here:

$$P(l = border|z) = \frac{p(z|l = border)P(l = border)}{p(z|l = border)P(l = border) + p(z|l = free)P(l = free)}$$

Symbol $P(l = border|z)$ denotes a probability that a cell with location l contains a road border after the fusion of measured data. $P(l = border)$ is the previous probability of the cell being occupied by the road border, and $p(z|l = border)$ is the probability that a cell is observed as one containing a road border, given an actual occurrence of the road border at the location l in the real world. This is the model of the measurements, and it is described in the following section. *Note: $l = free$ is exact negation of $l = border$, $P(l = free) = 1 - P(l = border)$*

4.2.5.2 Contour observation model

The purpose of this contour observation model is to capture the relation between imperfect measurement and what it may correspond to in the real world. The measurement is the detected location of the contour in the image, and we are trying to estimate the position of the side of the road in the real world.

This model assigns a probability of real-world occurrence of the side of the road given distance from the closest detected contour in the image. Again, this distance is not in the image coordinate frame but in the top-down grid map frame.

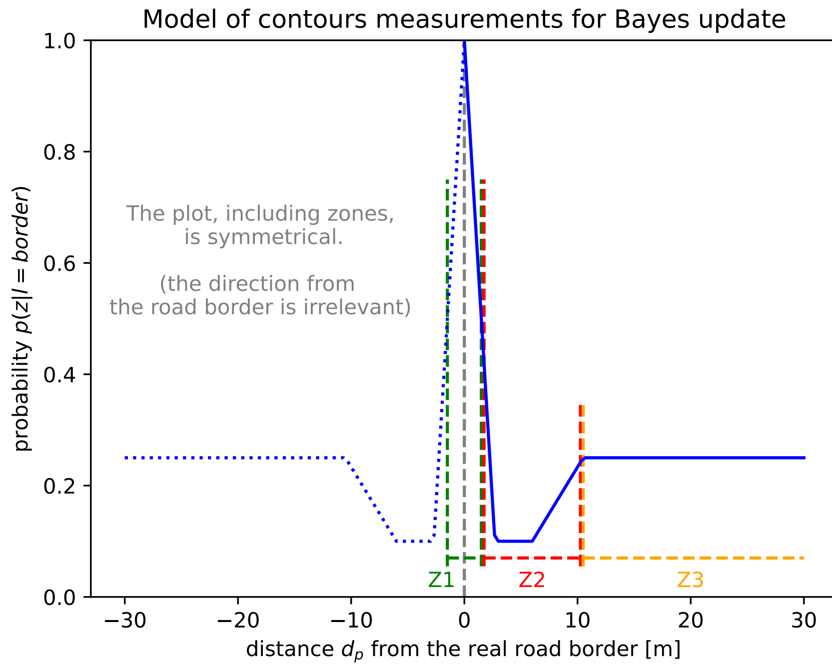


Figure 4.6: Probability of detected contour at location l given distance d_p from the closest real border of the road

Parameter	Distance d_p	$p(z\|l = border)$
Max	0m	1.0
Midpoint	0.45m	0.5
Z2 Start	0.9m	0.1
Z2 Mid	6m	0.1
Z3	10.5m	0.25
No detection	∞	0.25

Table 4.2: Points of polylines of the contour update model.

Figure 4.6 depicts the values I have chosen for this model. It is a polyline,

with all corner points shown in a table 4.2. As you can see, the plot shows symmetric values in the graph, as the model considers only distance, not the direction from the border. The peak is in the middle, at distance=0 (which is the exact position of the border). In the near proximity around this peak, the probability is still very high (but slightly decreases with the distance) as the detection could have been just slightly off. In figure 4.6, this is shown as a green zone Z1. The next section – red zone Z2 – represents an area that is further away from the border, about one meter or so, and it is assumed that it is unlikely that the detector would make such an error in the detection. And finally, the last zone (orange, zone Z3) represents the fact that places that are far away from the nearest border (are not the border themselves) have a very low probability of being detected as borders. It is important that both zones 2 and 3 have their probabilities (of positive occurrence) less than 0.5, meaning that it is more likely that there won't be any contour detection at that particular location.

This particular model was chosen somewhat arbitrarily and worked well with the tool for semantic segmentation I used. It may be appropriate to change this model for a different image semantic segmentation tool used. I have not explored any other models in this work, as it didn't seem to be necessary.

4.2.6 Costmap

This part discusses how the flattened contour map is used to guide the robot. The path planning system already implemented on the robot uses costmaps. The path planning module receives different costs from various modules running on the robot, builds one global costmap grid, and finds the shortest path in this grid between cells representing its current location and the target location. This path planner module also takes into account the cost of heading angle adjustments (steering).

To ensure compatibility with this system, my module needs to export one such costmap. The first somewhat naive version was to consider a distance to the nearest contour as a cost function. In theory, the robot should move exactly on top of the contours – zero distance = zero cost, and great distance = great cost. However, this cost function is not sufficient.

First of all, we don't want to walk on top of the contour, but rather just to the side of it. And in the ideal scenario, we don't want to walk too far away from it (we want to walk on the side of the road, not in the middle). We can re-use the idea of profiles from the previous GNSS-only version (section 4.1), but we will need some modifications.

Profiles of the cost function from the GNSS version are functions of vectorized geometric shapes of roads – the road has location, direction, and width. We don't have access to this information in this visual approach; we only have a grid map of cells containing contours. For this reason, the profile used in this work is a function of distance from the nearest contour cell. Figure 4.7 illustrates the mapping between distance from the contour and the cost that it incurs for the planner. The cost c of shown profile can be calculated using

the following equation, where o is parameter `OFFSET_M` and s is parameter `PROFILE_SLOPE`. These parameters are later discussed in section 5.

$$c = \begin{cases} 1 - d_p/o & d_p < o \\ \min\{1, (d_p - o) \cdot s\} & d_p \geq o \end{cases}$$

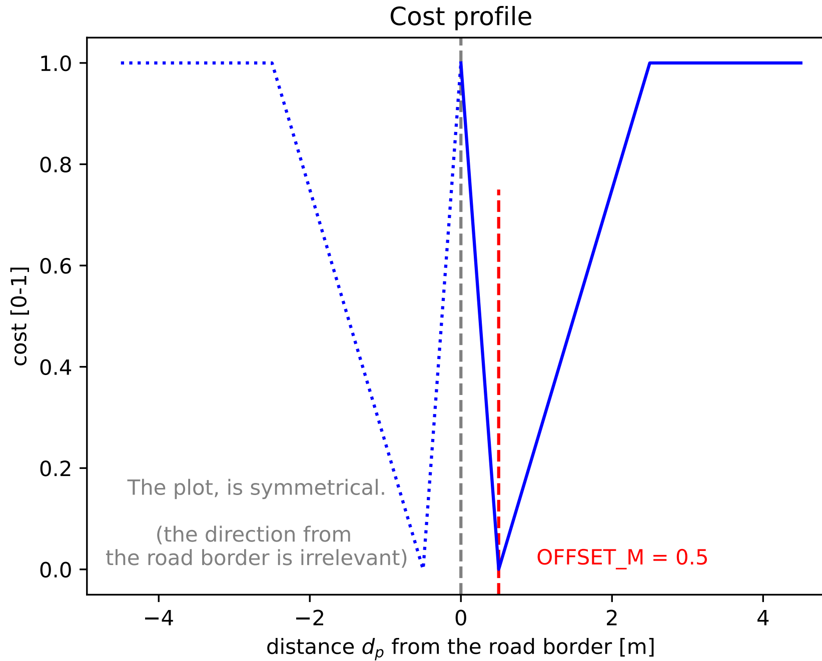


Figure 4.7: Currently used profile – a cost incurred for the planner given a distance to the nearest contour cell.

The profile is symmetric; a contour in the grid map does not have a defined direction on which side of the road it is. Notice the peak of minimum cost at a distance `OFFSET_M`. This is a parameter of how far from the road border we want the corridor through which the robot will walk. Two such corridors exist, one on either side of the contour. In theory, the robot should follow the one which is closer to his current location and not cross the middle barrier. This was, however, not observed for various reasons, discussed in sections 4.3.2, 4.3.5, and 4.2.8.

4.2.7 Additional grid map channel – Forbidden areas

To improve the quality of the generated paths, I have propagated more information to the costmap generation stage in order to give better hints to the path planner. More specifically, I started building a new grid map in the same manner as the contour one from "forbidden areas," which are defined as areas segmented by the segmentation module as one of the non-road classes

(not `CLS_ROAD` and not `CLS_SIDEWALK`). These forbidden areas are transformed to the top-down view in the robot's odometry frame, the same as contours, and then stitched together using the same Bayesian update method to the full map.

When building a costmap, this information about forbidden areas is combined with the previous method by adding high cost to areas that have a high probability of being forbidden. This usually results in completely blocking off one of the two corridors around the border.

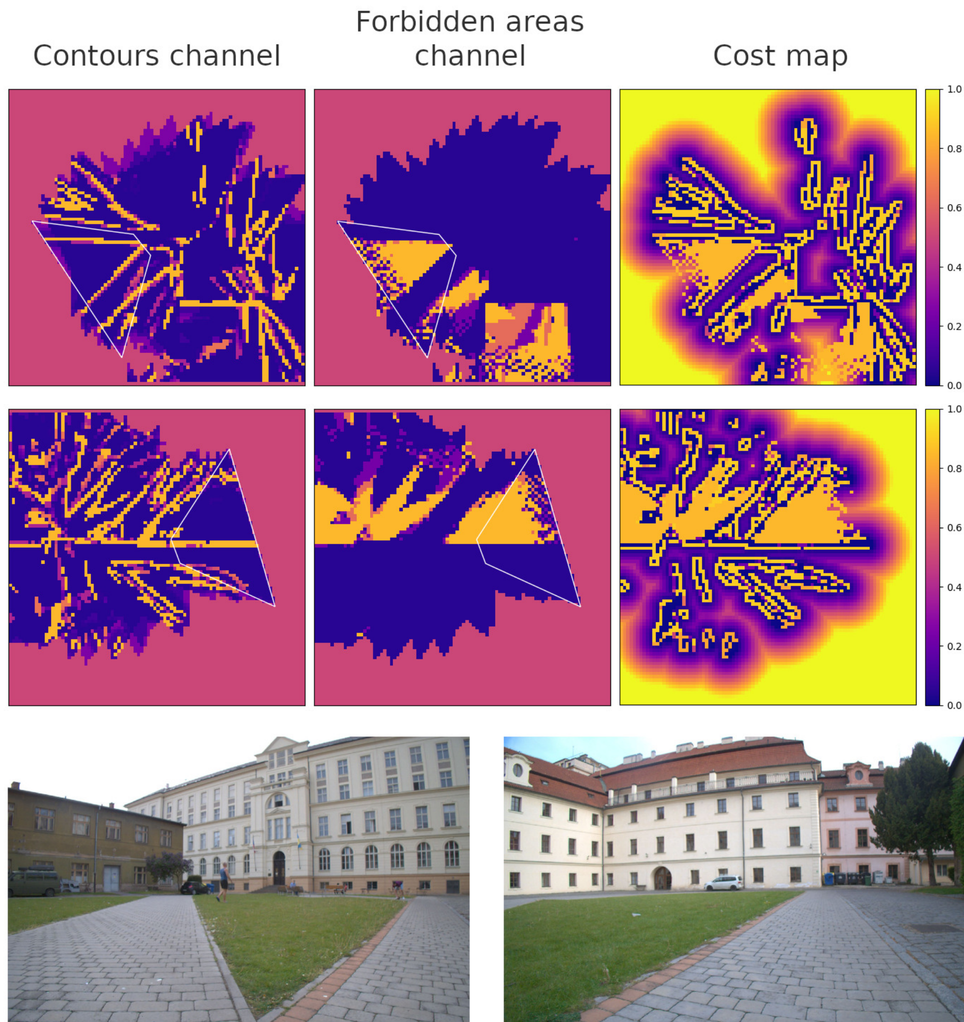


Figure 4.8: Computation of various top-down views of the robot. Each row shows a different scenario; the photos at the bottom are the views front the front-facing camera at those respective scenarios. Left column: Contour probability channel of the grid map. Middle column: Forbidden areas channel of the grid map. Right: Both channels combined into one costmap.

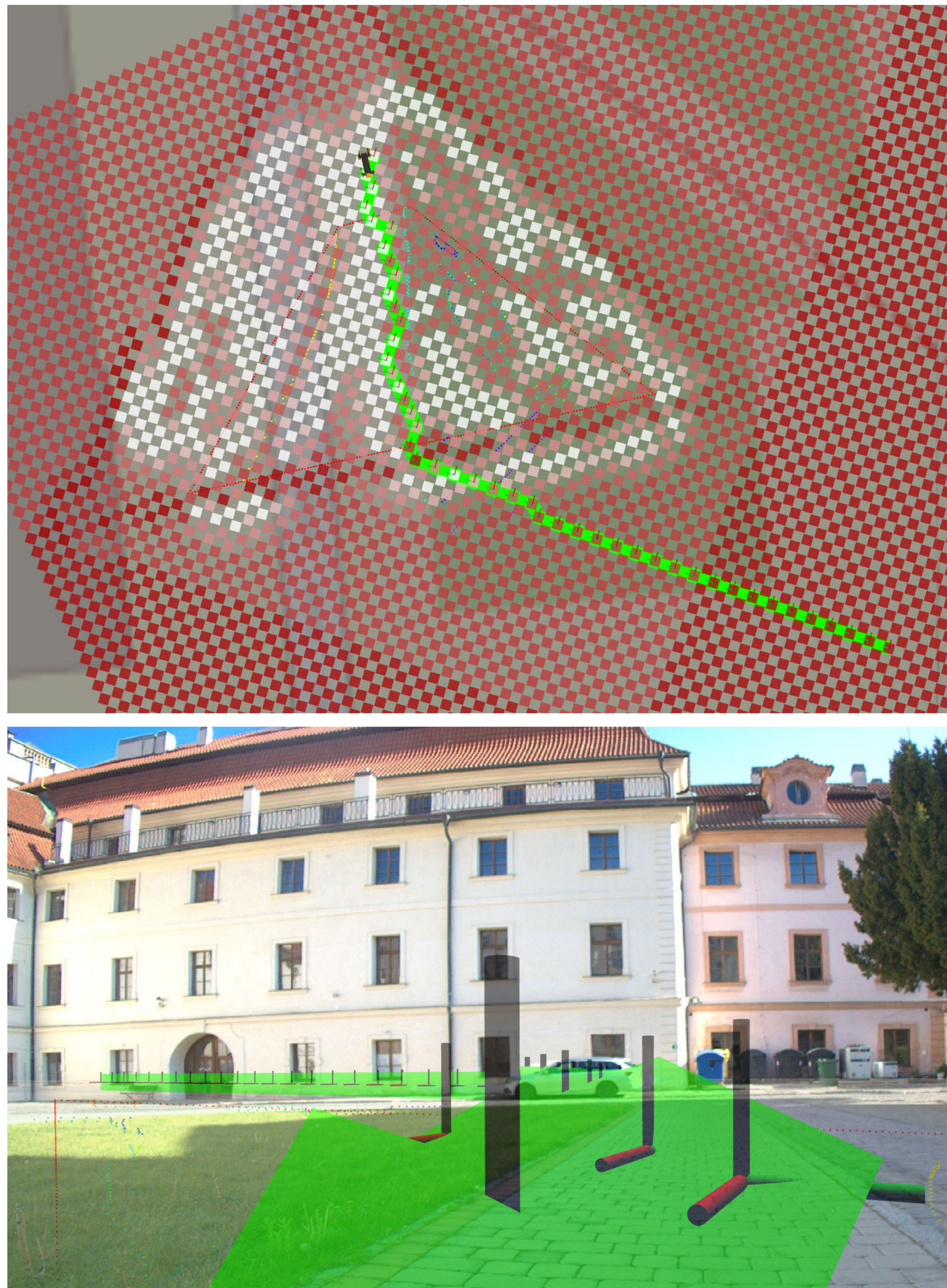


Figure 4.9: Top: Robot plans a path (green) alongside the sidewalk, rather than direct – shortest Euclidean path. The costmap shows one side of the road border with relatively low costs (white and close to white) and the other side with higher costs (the more saturated red, the higher the cost is). Without this last step of adding a channel for forbidden areas, both sides of the border would be more less white, and the path would have a tendency to be more straight, crossing the road border. Bottom: Same situation from the viewpoint of the robot.

4.2.8 Planning around the visible area

There is still a relatively lot of noise from the contour grid map, which leads to creating arbitrary barriers in the middle of the road. This effect is exacerbated even more when a part of the non-road area is in view. A lot of false-positive contours tend to appear in these areas. The planner sometimes has a tendency to avoid all of these "high-cost clusters" altogether and steer around the observed areas into the unknown. The problem is that once it leaves the road and ventures into the unobserved/terrain area, it will never return back on the road as it doesn't have enough incentive to cross the border of the road again. This behavior can be observed in figure 4.10.

I have partially fixed this behavior by introducing an artificial cost wall around the currently observed area, which dissipates with the distance. This incentivizes the robot to go forward and cross the visibility border there rather than to turn on the spot and walk "around" the mapped area.

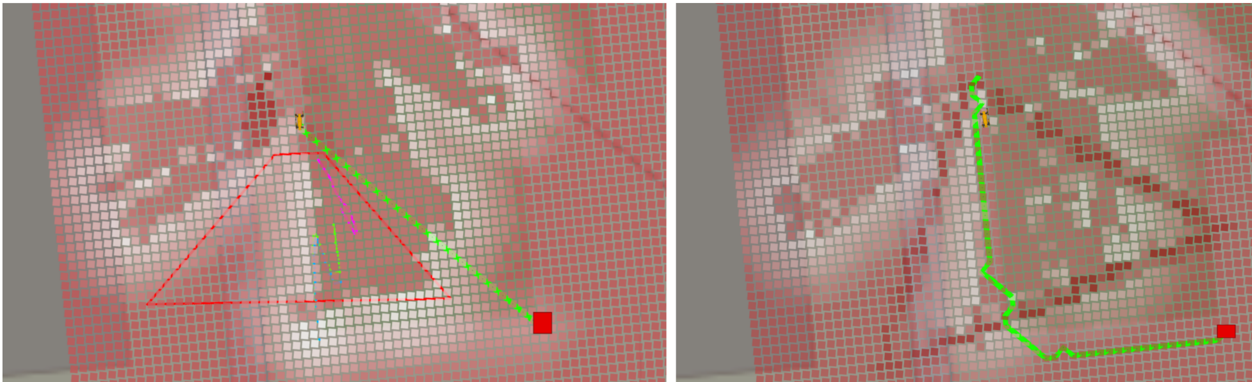


Figure 4.10: Planned path (green) from the robot to the goal (red box). Left: The path ignores corridors and barriers in the detected area because the path planner managed to "slip" outside of the detected area near the robot. Right: This time, the path follows the shape of the road, as it cannot cross the detected area boundary near the robot. The boundary can be seen as darker red (= higher cost) cells in the right image.

4.3 Encountered problems

4.3.1 Top down contour map transformations

An essential part of building a top-down contour map from multiple frames is knowing the correct transformation of how the robot moved (or rather, camera pose) in between those frames. Spot's API provides a coordinate frame called `vision` [22], which is an inertial frame that estimates the fixed robot 6DOF pose in the world (relative to where the robot booted up) and is calculated using visual analysis of the world and the robot's odometry. My module grabs a robot pose in the `vision` frame at the time each image is taken and uses the difference between the current and last robot's pose

to transform the top-down contour map. Once the map is transformed, it updates it with the information acquired from the current image analysis.

At first, this approach had very bad results, as shown in figure 4.11 (left). After some investigation, I found an issue with the cameras themselves. The image sent by the cameras was significantly delayed compared to the timestamp it was associated with. This delay ranged from about 1s in some experiments to 12s in the worst observed case. To mitigate this issue, I added a parameter to my module `CAMERA_DELAY_S`, which should be set to the observed delay. The robot software then uses older `vision` frame transformation, older exactly by the time delay set by this parameter. This solved the issue while testing on already recorded data, as shown in figure 4.11 (right); however, it is not a full solution to this issue. With the help of other colleagues, we managed to find issues in camera configuration and decrease this delay to about 0.3s, which is much better, but still not ideal. *The details of this configuration are not described, as it concerns a specific hardware issue and is outside of the scope of this thesis.*

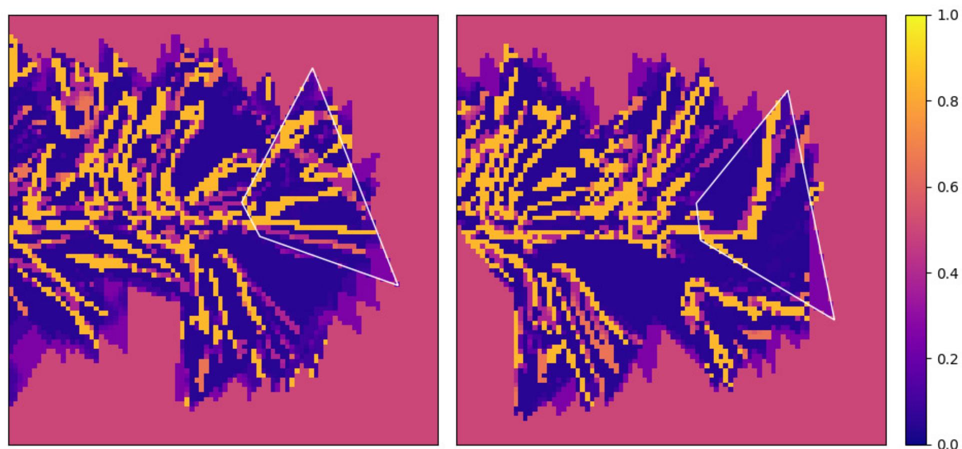


Figure 4.11: Merged top-down contour map - the probability that there exists a contour at a given world location. Left: Images from the camera are out of sync by about one second with the transformation data of the robot. Right: Images and transformations synced using appropriate `CAMERA_DELAY_S`. Both images represent the same scenario, with the robot at roughly the same physical position. We should see a relatively straight road with a left right-angle turn at the end, which can be clearly observed in the right image.

4.3.2 Costmap resolution is too small

The planner's costmap spatial resolution (0.5m per grid cell) is too small for precise robot guiding. Features generated by different profiles are too fine. The profile has to be very coarse, without big details. Otherwise, the planner is not able to correctly navigate the corridors or react to the cost gradients. We can't increase the resolution of the planner's costmap because that is already on the limit of computational power. Currently, the planner's resolution is two samples per meter. It may fill a relatively large area with

these points and then run a graph planning algorithm to find the shortest path. Nevertheless, for our task, we don't actually need a large area of planning with a far-reaching horizon. It would be sufficient to just "nudge" the robot locally in the desired direction. When the "nudging force" wouldn't be strong enough, the robot could cross the road border if the overall planner wanted to. The magnitude of the "nudging force" could be calculated in the same fashion as the profiles already described. I have not tried this approach due to the lack of time.

4.3.3 Strong priors in the lane detection

The lane detection part of the `Autonomous-Vehicle-Environment-Perception` [1] module did not produce the results I was hoping for on our dataset. It seemed like it was affected by a strong prior forcing the detected lane borders to follow in the forward-facing direction of travel. This prior may work in an automotive environment, but in our, it is not desired. Figure 4.12 shows lane borders detected where there are no lanes and all facing in the direction of travel. I removed the lane detector completely because it was practically useless for our application.

Figure 4.12: Two selected images with very incorrect detections of lane borders. All detected lane borders face the forward direction.

4.3.4 Vision module is slow

The performance of the overall module, including the vision stage, is crucial for an online control system. The time it took for the vision stage from a received image frame to outputting of the results was long, more than one second, which is not acceptable. I have solved this by partially rewriting some parts but mainly removing all code that was not absolutely necessary for the functionality of this stage. I removed the lane detection sub-system, YOLO-based object detection (such as cars, cyclists, traffic signs, ...), as well as signal color detection of the traffic lights and estimation of the distance to the stopping line.

4.3.5 Compass guiding the robot away

The magnetic compass of the robot has a relatively large drift. Usually, this can be corrected by estimating the direction of travel from the GNSS when the robot is walking; however, this is not possible when the robot is stationary for some time. In such cases, it happens very often that robot plans a correct route, sticking to the side of the road, relative to the observed contours in the camera, but when it tries to execute this path, it first "corrects" its pose, to align the magnetic reading with the desired orientation even before starting to move. This results in the robot turning away from the road at the beginning of the experiment and then moving forward. Sometimes "moving forward" means directly across the road border. It is then almost impossible for the robot to return back to its original intended path. Magnetic compass misalignment is shown in figure 4.13. We can see that the path gets planned correctly (projected in the camera image); however, when the robot starts following such a path, it steers off course instantly.

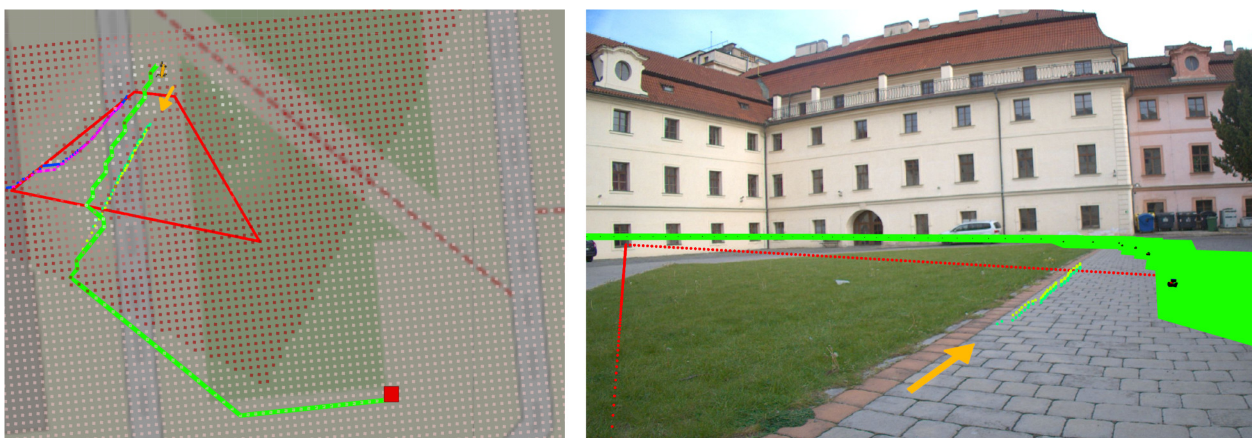


Figure 4.13: Robot absolute heading misaligned due to a compass error. The roadside is detected correctly (right image, green and yellow contours, pointed to by an orange arrow). The same roadside is shown in the top-down view (left image, same contours, again, pointed to by an orange arrow). In the top-down view, the detected contour should be exactly aligned with the beige-green interface on the map. The red dotted lines mark the ROI area (more in sec 4.4.2).

4.3.6 Computation power issues – the overall latency

The latency of each computational stage adds up to the overall photon-to-movement latency. While working on this project, I have continuously struggled with the lack of computational power, or rather, this latency being too high. My module runs around 2 Hz, introducing 0.5s of latency on average. The images from cameras this module takes as input were already delayed, as described in section 4.3.1, adding about 0.3s of latency at best. Processing the costmap of my module also took a relatively long time. Sometimes it was merged and used by the planner module nearly instantly (< 0.25s), but

sometimes it took several seconds. I am not sure where this time delay comes from. This compounds to about 1s of latency of the entire chain, which proved to be too long for online robot guidance. I had temporarily mitigated this issue by slowing down the speed of the robot's locomotion significantly, but this isn't a very good long-term solution.

4.4 Implementation details

4.4.1 LLA / UTM / UTM_local coordinate frames

The robot operates internally using `UTM` coordinates instead of `latitude` and `longitude`. `UTM` coordinate has some nice properties; namely, it is metric. However, values representing a location in a single `UTM zone` can quickly overflow 32-bit floating point precision and cause issues in the processing pipeline. ROS uses 32-bit floating point numbers in all of its standard messages, which means we cannot use 64-bit ones. The solution is another coordinate frame, `UTM_local` – which is `UTM` translated such that the robot starting position is represented by `0,0 (UTM_local)`. As a result, all coordinates are near-enough for 32-bit floating point precision.

An unfortunate byproduct of using the `UTM_local` frame is that modules using this frame (including my road navigation module) do not work without a good GNSS lock, as they do not have this initial transformation – for example, when starting the robot indoors.

4.4.2 Contour extraction ROI

The robot's camera may see many different contours, but we are interested only in some. More specifically, only in those which are on the ground and are close enough to be considered relevant and reliable. Figure 4.14 shows a side view of the robot with a camera and rays from the camera that project to specific pixels. As we can see, the ground can be seen only by a part of the field of view of the camera. The camera has a limited resolution (limited number of pixels), which in turn sets the minimum angle α between two rays of consecutive pixels, and that limits the minimum distance d_α, which is a distance between two ground points in the real world. Two such ray pairs are shown in the figure. The further away from the camera these points are, the longer this distance is, decreasing the precision of pixel-to-real-world projection. Setting a constraint on this precision constraints the maximum view distance. The other (near) end of the camera is constrained by the nearest place the rays from cameras can reach (see Ray_n in figure 4.14). Everything between the robot and the place where this ray intersects with the ground plane is an invisible area for the robot. For my work, I have defined the reliable ROI (Region of interest) as an area in the image where the distance between any two neighboring pixels projected to the ground plane estimate is less than 0.5m.

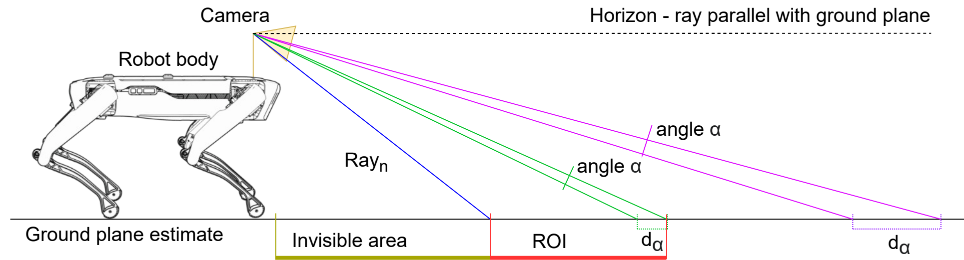


Figure 4.14: Rays projected from the camera onto a ground plane estimate. Purple: Ray pair (of two neighboring pixels in the camera) whose distance d_α is higher than the ROI threshold. Green: Ray pair (of two neighboring pixels in the camera), with the same angle α, whose distance d_α is within the ROI threshold. Red: Area which is considered a reliable region of interest.

Note that this ROI calculation is entirely dynamic. Whenever the camera orientation or ground plane is changed, the ROI region will be different. When the robot walks and the camera tilts slightly, the ROI shifts accordingly. As well, when the robot encounters stairs, the ground plane estimates rotate to match the staircase. In theory, the camera could be mounted on a robotic arm on top of Spot, move around, and the ROI would remain well-defined all the time. The algorithm is able to work with both landscape and portrait camera orientation. The illustration shows only a 2D side view; however, the reasoning is the same in the horizontal axis as well. *The ROI is shown in the figures throughout this work as a dotted red contour.*

4.4.3 Measuring the camera delay

To correctly measure the delay between the actual pictures provided by the main camera and the transformation information from the robot, I have written a little test program. This program reads the images from ROS as well as the `TF` messages, containing the rigid transformations between the camera, robot body, and ground plane estimate. The program reads the image data and the transform data from different times, with constant time offset between them. With this information, it renders red lines where the horizon should be, according to the transform data. The pink lines represent absolute heading marks. When the time offset is set correctly, the rendered horizon should match the real horizon in all the images in the sequence. The pink lines should stay in consistent heading directions when the robot rotates around its yaw axis. Figure 4.15 shows two different settings of the time offset, one to demonstrate incorrect camera-transform synchronization and one to demonstrate the correct one.

Figure 4.15: Left: Rendered horizon and heading lines do not line up with the image – the transformation used and the photo itself are each from a different time; Right: Rendered horizon and heading lines are in sync with the picture.

Chapter 5

Experiments and results

This section discusses the importance and effects of various parameters mentioned in the description of the methods. It also discusses failed experiments and what algorithmic changes and improvements they lead to.

5.1 Contour filtering

The initial semantic segmentation and following contour extraction (detection of the bordering areas between different segmentation masks) yield a relatively high number of undesired detections. For this reason, we need to filter some detection, as described in the section 4.2.4. The parameters of this algorithm are discussed here in this section. Some of the undesired erroneous detections may be characterized as very short segments. The parameter `MIN_CONTOUR_LENGTH_PX` classifies segments as noise or not based on their length. Figure 5.1 shows several values of this parameter tested and which contours are kept.

The second very strong feature of undesired contours is their shape. Man-made environment (roads, sidewalks, city in general) tends to utilize long straight lines very often. As such, the edges of a road or sidewalk tend to be straight or with minimal curvature (a gradual turn). This means that a correctly detected contour should copy this shape of minimal curvature. In the detections, however, I have observed many odd-shaped "blobs", which were incorrectly detected (as shown in figure 5.2 – pointed to by the right arrow). These blobs contain many low-radii turns (= high curvature turns). However, not only do these incorrectly detected blobs contain low-radii turns, but also sharp or even 90-degree angles between multiple straight line segments (as shown in the same figure 5.2 – pointed to by the left arrow). The key difference between these two situations is the overall length of the segments between the low-radii turns. My filter splits the contour at the places of low-radius turn, and in the case of the "odd-shaped blob", it gets cut into many short segments, and these segments will be removed using the `MIN_CONTOUR_LENGTH_PX` filter. In the case of a sharp turn between two straight lines, the new contours created by the split remain in the system as they are both long enough. The radius is controlled by the parameter `MAX_ALLOWED_ANGLE`. Figure 5.3 illustrates two of these values.

Figure 5.1: Several values of `MIN_CONTOUR_LENGTH_PX` parameter. Top left: 0px – any detected shape passes the filter. Notice several small contours around the legs of the bench; Bottom left: 150px – The results are over-filtered. Contour along the length of the sidewalk did not get detected as one continuous piece, but rather two consecutive. As such, it was not long enough to pass; Right column: 25px – Balance between removing very small details but keeping many lines, which may be good contour candidates. This experiment was executed with an image resolution of 640x192. The red dotted lines mark the ROI area (more in sec 4.4.2).

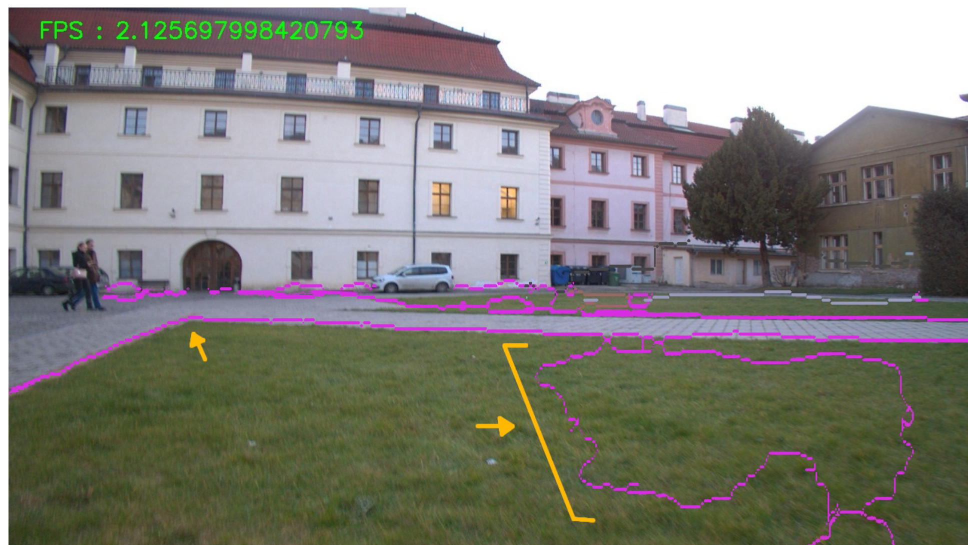


Figure 5.2: Contour detections before max curvature filtering. Notice the 90-degree bend in one curve between two straight segments (pointed to by the left arrow) and an odd-shaped blob containing many sharp angle bends (segment pointed to by the right arrow), which is caused by an incorrect detection.

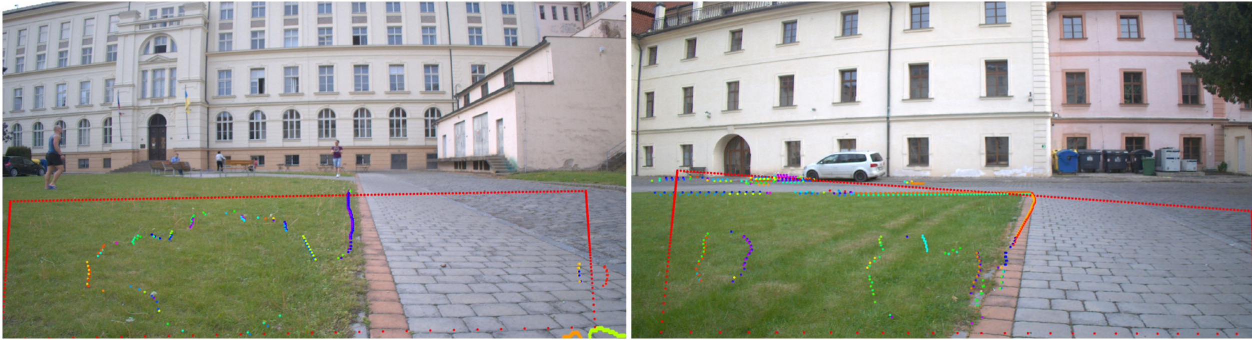


Figure 5.3: Left: parameter `MAX_ALLOWED_ANGLE` = 11.2 degrees – we can observe contours split into many different sub-contours. Only the very, very straight ones remained in one piece. Right: parameter `MAX_ALLOWED_ANGLE` = 90 degrees – the more contour stay in one piece for longer. In both images, each contour is depicted using its own hue. The red dotted lines mark the ROI area (more in sec 4.4.2).

This filtering method did not work very well as is. The problem was with the discretization of the contours. Contour positions come from the pixel locations of the detected segmentation borders. In the image, all these contours are 8-neighborhood connected. After projecting the contours from the image frame to the top-down frame, the angles between consecutive points become relatively large, especially at the places which were previously 4-neighborhood connected. Many out-of-line pixels may appear, even if it is just barely, due to noise in the detection. Each misalignment has a big potential to incur a sharp angle that would split the contour even when the contour is overall nearly straight. These sharp-angle-splits are shown by blue arrows in figure 5.4. For this reason, I had to have the value of `MAX_ALLOWED_ANGLE` set to a really high number of 90 degrees. Future improvement could employ some Gaussian smoothing of the shape of the line or spline fitting in smaller segments. More changes would be required, however, because the contour points are not spaced uniformly (projection from image to top-down view places pixels which are further away from the camera to be far from each other).

5.2 Choosing the appropriate profile function

The profile is a function with some properties. It has to be continuous, defined on all real numbers in the interval $\langle -\text{MAP_RANGE}, \text{MAP_RANGE} \rangle$, and output values between in an interval $\langle 0, 1 \rangle$. Aside from these requirements, the function can be arbitrary. There were two different profile functions shown in this work. One for the GNSS version (figure 4.1) and one for the vision-only (figure 4.7). Both of these profile shapes are the results of multiple iterations of tweaking. The main thing to consider when designing a profile shape for this module is the resolution of the planner module's costmap. I tried using a different profile (for the vision version) with much finer features (similar to

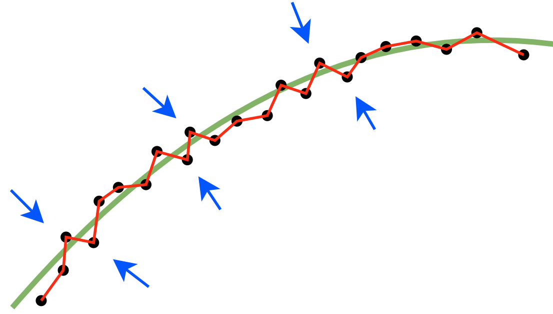


Figure 5.4: Illustration of sharp-angle-splits (at positions of blue arrows) along a relatively smooth but discrete contour (red), which is comprised of points (black) projected from the pixel grid with only a small deviation from an optimally fit contour (green).

the GNSS type); however, most of these details got lost when transferring to the planner. Furthermore, when downscaling the calculated values for the resolution of the planner costmap, there is no notion of which profile features are more important to keep and which ones are not. Whenever merging the two costmaps, there may also be a rotational misalignment (planner module's reference frame being different than mine or similar reasons). The downscaling algorithm is a simple sampling at the center points of the grid, which more often than not has a contra-productive effect on the final costmap (and, in turn, the planned path), as there is no control over which parts of which profile features will be selected. This washes away most of the guiding information, and the results may seem like random noise, completely ignoring any corridors the profile may have created.

In the section Future work 6.2, I propose a new guiding idea, which is not to use the path planner at all. I believe would have better results and could accommodate finer details of the profile function for finer robot navigation. It is expected that during deployment, the robot will have several fine-tuned profiles for different geographical areas and auto-choose the appropriate profile using the current GNSS location. This will allow for an even better definition of the desired robot behavior. For example, in some areas, there might be a ditch near the road, and the used profile would penalize navigating near it with a high-cost wall. However, in other areas, we might want the robot to walk in a corridor outside the road, not on the road itself (for example, because of high-speed traffic). This is achievable using a different profile.

5.3 Costmap strength

The costmap from this road navigation module is only one part of the overall input for the planner. There may be other costmaps from different modules, and a natural question arises "How to combine multiple costmaps and balance their effects?" The planner constructs one total costmap, which it uses for

planning. The cost at a given location in the total costmap is calculated as a weighted sum of all other input maps. The weight of the costmap of my module is set using a parameter `COST_GAIN`, which is a constant that multiplies the costmap before it is sent as a ros message. The higher this constant is, the more effect this costmap will have. When this parameter is set to a small value, the cost walls around the road borders may be ignored. Figure 5.5 shows the planned path, which results from different gain settings.

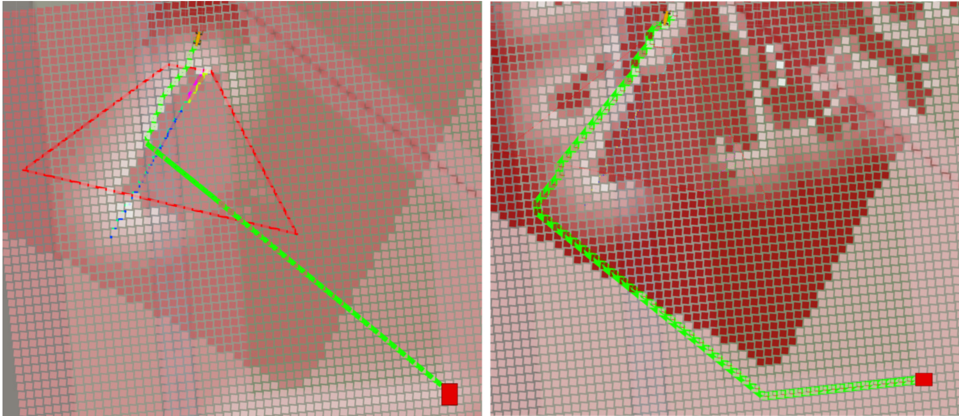


Figure 5.5: Planned path (green) from the robot to the goal while utilizing the costmap. White parts of costmap – low cost; Red parts – high cost; Left: The `COST_GAIN` is set to a low value of 1. Notice that the planner cuts across the road border when the path starts to be much longer than the direct Euclidean path; Right: `COST_GAIN` is set to a much higher value of 40. The planner plans the path along the detected road border for a much longer span, essentially until the end of the visible area. *Side note: An observant reader may notice the misalignment between the OpenStreetMap overlay and the heading of the detected road. This is caused by a magnetic compass issue, discussed in section 4.3.5.*

5.4 Mission navigation with replanning

The costmap that is being built from the scene semantics is only local – what is in the camera ROI (section 4.4.2 discusses the ROI computation) and in the short-term Bayesian map. The missions usually span much larger areas than what is covered by this. For this reason, the planner plans a path according to what it already knows, and the cost of the unknown area in the costmap is set to a constant value. This way, when planning outside of a known area, the shortest path is equivalent to the shortest Euclidean path. (Note that the path is planned in an eight-neighborhood grid, so the direct Euclidean path is not always achieved.) The robot starts moving along this path, leading toward the final goal, and as the robot moves, it collects more information about the environment. It updates the internal costmap with the contours from the newest images and replans the path. The robot is then kept on the road all the time, even though the initial path plan proposed to go through non-road terrain. This behavior is shown in figure 5.6, where several updated

plans are shown as the robot moves along its path. The unused parts of the older plans are shown in slightly less saturated color, where the robot started moving along the new plan with newly detected contours.

5.5 Quantitative results

The robot is supposed to walk in the corridor of a minimal cost in the costmap. We can observe the cost at the position where the robot has been and in some areas around it. It should hold that the robot position has a minimal cost (the center of the corridor), and all other surrounding places should have a higher cost. We can log these values throughout the experiment, and if we observe lower cost values at the robot's path than around it for the majority of an experiment, it shows us that the robot believes it walked in the area of the lowest cost. It is ok when in some places, the robot's local cost exceeds those of its surroundings, as it may be necessary for the globally cheapest plan to drive through some of the more expensive areas.

Figure 5.7 shows the costs throughout the run of the experiment described in the previous section, and figure 5.6. We can see that at several places, the robot wasn't at the cheapest possible cost – Event A: around time 40, the robot was very near the terrain sidewalk, and some detections of it may have been erroneous. Event B: around time 90, the robot switched from tracking the left contour of the sidewalk to the right side of it, crossing the centerline. This can be observed as the orange path plan in figure 5.6. Event C: Nearing the end, around time 125, the Spot took a little shortcut across the terrain and didn't stay on the road. Event D: Around time 140, a car arrived and waited for a while in front of the robot camera, allowing the driver to take some pictures. However, this made Spot's vision system a little confused.

Note that this experiment considers only what Spot believes about its surroundings (cost-wise). A similar experiment should be conducted, with the addition of having the ground truth location data and comparing the cost at the location at which Spot believes to be with the cost calculated by applying the profile function on the OpenStreetMap data at the ground truth location. One possible source for this ground truth data is a different software pipeline, which already runs on the Spot and performs SLAM from the onboard lidar. The generated map can be (manually) aligned with the OpenStreetMaps, providing accurate robot localization. I wanted to perform this and more experiments, particularly testing longer missions and different improvements and parameter settings described in this work, but due to technical difficulties[1], I didn't have time to finish them.

[1] For example, a forced firmware update from BostonDynamics introducing breaking API changed for our Spot–ROS interface

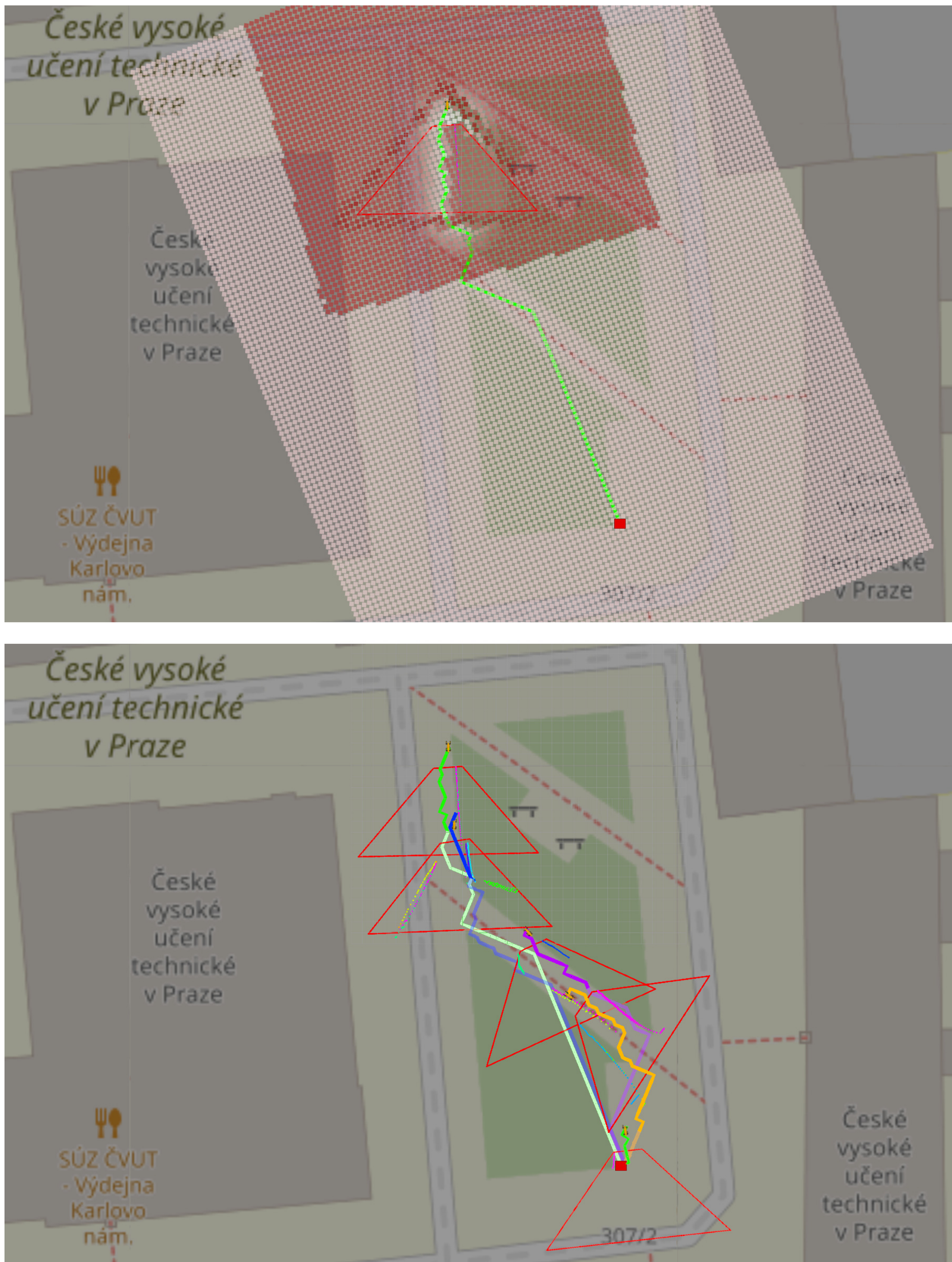


Figure 5.6: Sequence of plans to complete the mission – move from the top-left corner to the bottom-right, using only roads or sidewalks. Top: Costmap built from the local information from the vision system; the rest is set to a default constant value. Bottom: Sequence of plan changes as the robot moves along the planned path and new parts of the world come into view. The robot first follows the green plan, then blue, purple, orange, and at the end, the very short green. *Not every plan update is shown in the figure, for the sake of clarity*

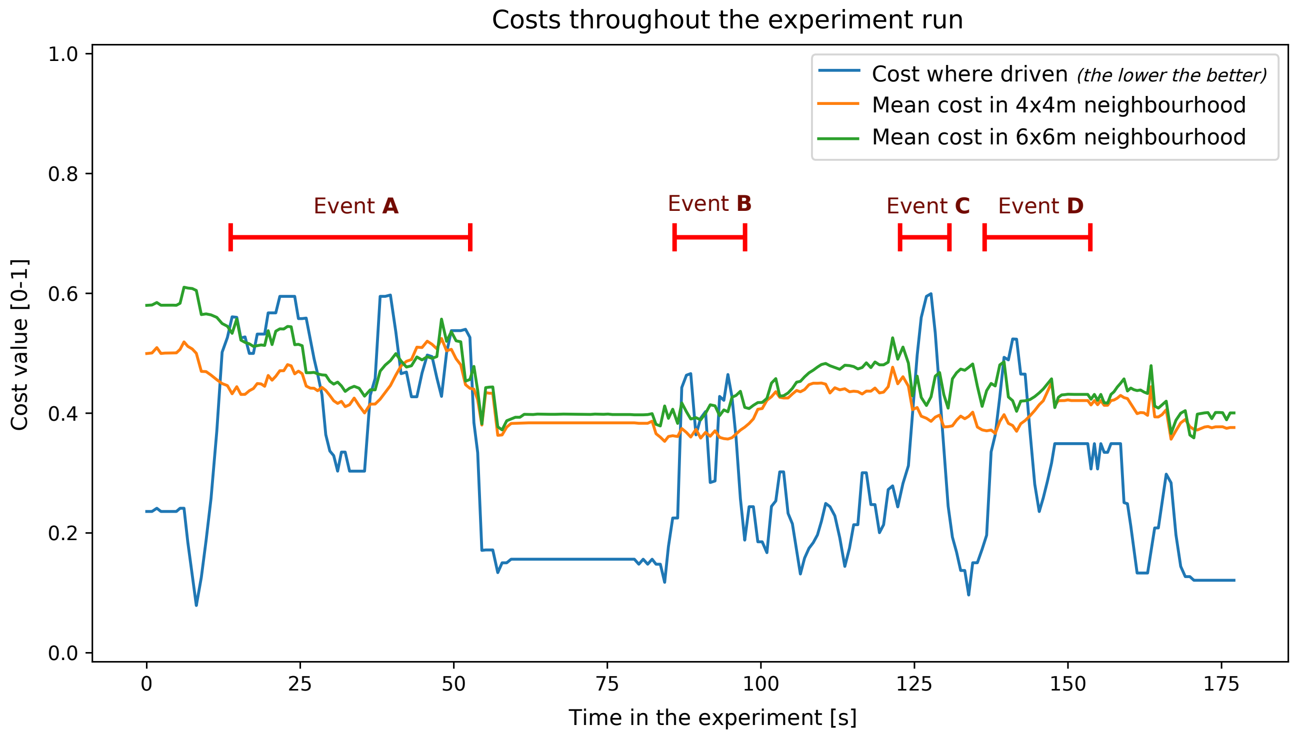


Figure 5.7: Cost values at the robot's position. The value at the exact position of the robot is shown (blue), as well as mean values in the robot surrounding area (orange, green). Several events that happened during the experiment are marked (in red) at their respective times of occurrence. A – robot drove very near the road edge; B – robot crossed the road; C – robot took a shortcut; D – robot vision distracted by a distracted driver. These events are described in more detail in the text.

Chapter 6
Conclusion

In this thesis, I have proposed an algorithm that generates a new costmap using a front-facing RGB camera. This costmap represents the semantic meaning of the scene. The costmap serves as a main input for path planning and navigating the robot. I have proposed using a Bayesian update to improve the stability and temporal consistency of the computed results, which enhanced the quality of the final costmap. In a controlled environment (the CTU courtyard), the robot is able to navigate on the sidewalks, even when the direct path to the target GNSS waypoint is through the non-road terrain. However, the module is relatively sensitive to various real-world disturbances or noisy measurements. The navigation system requires a good GNSS position lock as well as precise heading information, which is not always available. Based on the valuable experience gained whilst working on this project, I have several ideas and improvements for future work. They are discussed in the following section. The module is not completely finished yet and is not ready for real-world deployment yet. Figure 6.1 anecdotally illustrates one situation we have encountered outside of university grounds.

6.1 Code repository

Source codes for this work are available at Czech Technical University's GitLab. Repository [23] contains the source codes of the main module, all stages except the vision module. It runs on Python2 with ROS Melodic or Python3 with ROS Noetic. Repository [24] contains the vision module, including my changes to it. This runs using Python3 only. The main project invokes the vision project as its sub-process. This split into multiple was necessary for the deployment on the physical robot, as only the older ROS Melodic was available, but the vision module requires Python3.

6.2 Future work

Experience gained during this work leads me to believe that guiding the robot using a costmap approach is not ideal. Two main reasons are:

Figure 6.1: The robotic dog in the real world environment – standoff with a meat-based dog.

- **Global vs Local planning**
  The robot planner makes a full plan (path) of how to get to the target waypoint. But while making this plan, most of the information about the road is not available (it is not in view). The planner needs to replan every so often because of this, and it does not always do so correctly (this could however be addressed separately). However, frequent re-plans are desired in the near vicinity of the robot, as finer information about the exact road position comes into view, or the GNSS location of the robot drifts a little. In general, these re-plans do not get triggered, as the change is not considered important by the planner from the global perspective. That means the robot is not going to adjust by half a meter sideways, as half a meter is negligible in the overall 30m route, but it is important locally for the robot not to get hit by a car.

- **Fine-ness of navigation**
  The difference between the two robot poses, one on the side of the road and the other in the middle is very short in terms of Euclidean distance (it can be even less than one meter), however very big in terms of cost or safety of the operation. This disproportion is not addressed in the global costmap. The global map cannot have a resolution adequate for guiding in such a fine grid. And even if it did, it would need to recalculate the

robot path even for a small change in the underlying costmap.

- **Planner operates in absolute coordinate system**
  The planner uses a coordinate frame linked with the GNSS navigation system. More specifically, UTM frames. The vision system, however, operates in the local system centered at the robot location. The output of the vision system is in terms of "there is a danger on the left side of the robot". We need to transform this into an absolute UTM frame for the planner to be able to use this information. The accuracy of this transform relies on the accuracy and temporal stability of the GNSS lock and compass heading. Neither of these reached high enough accuracy in our experiments. The magnetic compass is especially problematic. It is easily confused by electromagnetic noise in the environment as well as one generated by the robot's own powerful locomotion motors. These magnetic deviations can reach up to 90 degrees within a single experiment.

If I were designing a system for visual roadside navigation again, I would choose a different approach. It would be a module that communicates with the path follower module (figure 6.2 shows a revised architecture scheme) and sends "nudges", which way the robot should deviate from the original plan. Similar to a cost wall, which would get applied only locally – the close the robot gets to the wall, the stronger the "nudge". These nudges would force the robot to stick to the side of the road (or similar behaviors, as defined by the currently selected profile). The locality of this approach solves all three aforementioned problems with the costmap-based path planner. This nudge-based system is similar to an obstacle avoidance system but using virtual obstacles.

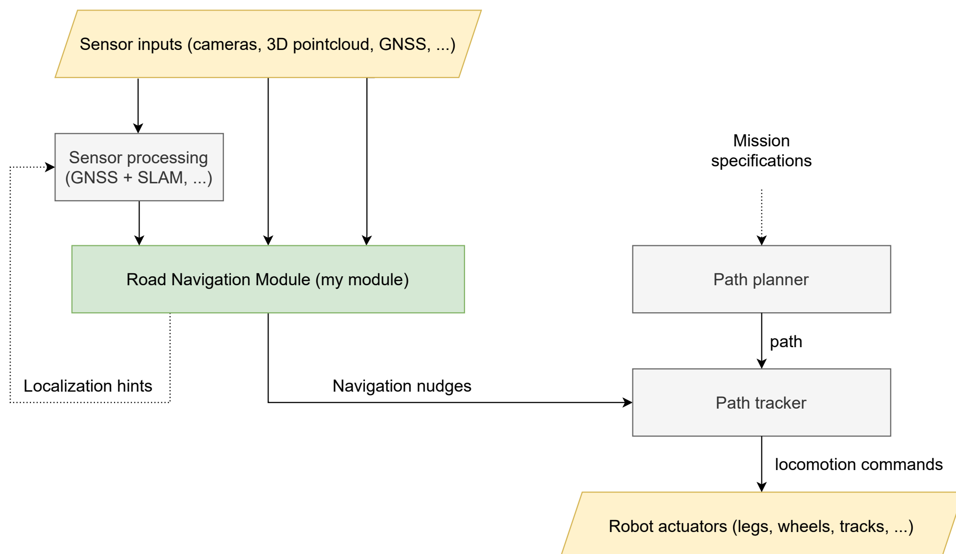


Figure 6.2: Revised control architecture diagram: Road navigation module interfaces with the other parts of the system.

There are other possible areas where to improve the system, but these do not seem as important. Or rather, they may further improve the results of the new local-based system.

- **Semantic detection of roads in the top-down contour map** and steering using the knowledge of the road object, not just by the contours by themselves. This improvement would allow more sophisticated profiles (distinction of the sides of the road; “middle of the road” defined as the geometric middle, not as “a place far enough from the edge” and so on).
- **Bayesian update model** was chosen relatively arbitrarily in this work. An improvement would be to learn this model by comparing the detection against some ground truth. Other information could be used for this model, not only contours but segmentation masks of different areas, the confidence of the segmentation output, or OpenStreetMaps as a prior. One mistake that is present in the current implementation is that the lack of detected contour automatically implies that no contour is present at that particular place. That may not be true in the case of occlusion. Such occlusion may be extracted from the segmentation masks, and the top-down Bayesian costmap should be updated accordingly. The lack of this functionality did not pose an issue on our dataset.
- **Absolute localization hints** may be provided by this module when it detects a road in the camera. The position and heading of the road may be compared with the map data obtained from OpenStreetMaps, and if a similar road is found in the OSM data, the robot may readjust its belief about its current heading or position to minimize the inconsistencies between these two data sources.

References

[1] Amirhossein Kazerouni et al. "An intelligent modular real-time vision-based system for environment perception". In: *arXiv preprint arXiv:2303.16710* (2023). URL: https://github.com/Pandas-Team/Autonomous-Vehicle-Environment-Perception.

[2] Ayush Ishan. *Road_Segmemtation – github*. URL: https://github.com/AYUSH-ISHAN/Road_Segmentation.

[3] NavBlind. *RSidewalk-semantic-seg – github*. URL: https://github.com/navblind/Sidewalk-semantic-seg.

[4] *Semantic Segmentation Baselines*. URL: https://github.com/Kautenja/semantic-segmentation-baselines.

[5] *Semantic Segmentation Fully Convolutional Neural Network*. URL: https://github.com/italojs/road-semantic-segmentation.

[6] Andreas Geiger et al. "Vision meets Robotics: The KITTI Dataset". In: *International Journal of Robotics Research (IJRR)* (2013).

[7] Marius Cordts et al. "The Cityscapes Dataset for Semantic Urban Scene Understanding". In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2016.

[8] Gregory Kahn, Pieter Abbeel, and Sergey Levine. "LaND: Learning to Navigate From Disengagements". In: *IEEE Robotics and Automation Letters* 6.2 (2021), pp. 1872–1879. DOI: 10.1109/LRA.2021.3060404.

[9] Alessandro Giusti et al. "A Machine Learning Approach to Visual Perception of Forest Trails for Mobile Robots". In: *IEEE Robotics and Automation Letters* 1.2 (2016), pp. 661–667. DOI: 10.1109/LRA.2015.2509024.

[10] *Boston Dynamics Spot*. URL: https://www.bostondynamics.com/products/spot.

[11] *HUSKY: Unmanned ground vehicle*. URL: https://clearpathrobotics.com/husky-unmanned-ground-vehicle-robot.

[12] *TRADR: Long-Term Human-Robot Teaming for Disaster Response*. URL: http://www.tradr-project.eu.

[13] Stanford Artificial Intelligence Laboratory et al. *Robotic Operating System*. Version ROS Melodic Morenia. May 23, 2018. URL: https://www.ros.org.

[14] *A bag is a file format in ROS for storing ROS message data*. URL: http://wiki.ros.org/Bags.

[15] OpenStreetMap contributors. *Planet dump retrieved from https://planet.osm.org*. https://www.openstreetmap.org. 2017.

[16] *Euclidean Distance Transform – Scipy documentation*. URL: https://docs.scipy.org/doc/scipy/reference/generated/scipy.ndimage.distance_transform_edt.html.

[17] Towards Data Science Constantin Toporov. *Robot following a walkway using image segmentation*. URL: https://towardsdatascience.com/robot-following-a-walkway-using-image-segmentation-272bebd93a83.

[18] Marvin Klingner et al. "Self-Supervised Monocular Depth Estimation: Solving the Dynamic Object Problem by Semantic Guidance". In: *European Conference on Computer Vision (ECCV)*. 2020. URL: https://github.com/ifnspaml/SGDepth.

[19] Yeongmin Ko et al. "Key Points Estimation and Point Instance Segmentation Approach for Lane Detection". In: *IEEE Transactions on Intelligent Transportation Systems* 23.7 (2022), pp. 8949–8958. DOI: 10.1109/TITS.2021.3088488. URL: https://github.com/koyeongmin/PINet_new.

[20] Glenn Jocher et al. *ultralytics/yolov5: v7.0 - YOLOv5 SOTA Realtime Instance Segmentation*. Version v7.0. Nov. 2022. DOI: 10.5281/zenodo.7347926. URL: https://doi.org/10.5281/zenodo.7347926.

[21] *Czech Technical University course – UIR – Occupancy Grid Maps*. URL: https://cw.fel.cvut.cz/b221/courses/uir/labs/lab03.

[22] *Boston Dynamics documentation – Reference frames*. URL: https://dev.bostondynamics.com/docs/concepts/geometry_and_frames.

[23] *Czech Technical University GitLab – Main source codes*. URL: https://gitlab.fel.cvut.cz/cras/robingas/roadnav.

[24] *Czech Technical University GitLab – Vision source codes*. URL: https://gitlab.fel.cvut.cz/cras/robingas/roadnav_vision.