Online 3D Path Planning and Exploration for Autonomous Mobile Robots in Unstructured Environments

Online 3D-Pfadplanung und Erkundung für autonome mobile Roboter in unstrukturierten Umgebungen

Master thesis by Stefan Manuel Fabian
Date of submission: September 2, 2019

1. Review: Prof. Dr. Oskar von Stryk
2. Review: Dr.-Ing. Stefan Kohlbrecher
Darmstadt

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science Department
Fachgebiet Simulation, Systemoptimierung und Robotik

Online 3D Path Planning and Exploration for Autonomous Mobile Robots in Unstructured Environments
Online 3D-Pfadplanung und Erkundung für autonome mobile Roboter in unstrukturierten Umgebungen

Master thesis by Stefan Manuel Fabian

1. Review: Prof. Dr. Oskar von Stryk
2. Review: Dr.-Ing. Stefan Kohlbrecher

Date of submission: September 2, 2019

Darmstadt

Abstract

For the past decades, path planning for robots has been an active field of research. Recent advances in computation have enabled the introduction of additional constraints and incorporating terrain structure. In rescue robotics, in particular, the expected deployment areas are unknown and unstructured environments either in nature or due to structural changes, e.g., a partially collapsed building. To be able to operate in unstructured terrains, autonomous unmanned ground vehicles have to predict their orientation and the contact points with the ground in order to prevent situations in which the robot may tip-over and require human intervention. Current approaches demand significant computation time to estimate the robot's pose and contact points, rendering them incapable of being used online on a robot without a connection to a remote operator station. This work presents a path planning approach using a pose prediction heuristic to quickly predict the robot's pose consisting of its location and orientation, including the contact points with the ground. The proposed algorithm can plan stable paths in real-time on limited hardware resources. It is evaluated on a robot and in simulation. On the robot, the real-time planning capability and a parameterized trade-off between stability are demonstrated. In the simulation, the accuracy of the pose prediction is evaluated. Finally, the computation time is evaluated for different graph resolutions, and it is shown that the presented approach is capable of planning stable paths on limited resources in real-time.

Zusammenfassung

Pfadplanung für Roboter ist ein seit Jahrzehnten aktiver Gegenstand aktueller Forschung. Die Fortschritte moderner Computer erlauben die Betrachtung zusätzlicher Nebenbedingungen und der Bodenstruktur. Besonders in der Rettungsrobotik sind die zu erwartenden Einsatzgebiete im Voraus unbekannt und natürlich oder durch strukturelle Veränderungen, z. B. durch Einsturz von Teilen eines Gebäudes, unstrukturiert. Um die Einsatzfähigkeit in diesen Gebieten zu gewährleisten müssen autonome unbemannte Bodenfahrzeuge ihre Orientierung im Raum und die Kontaktpunkte zum Boden vorhersagen können, damit Unfälle, die den Eingriff von Menschen erfordern, verhindert werden können. Aktuelle Ansätze benötigen signifikante Berechnungszeiten, um die Orientierung und Kontaktpunkte des Roboters zu bestimmen, wodurch ein Einsatz direkt auf einem Roboter, der keinen Kontakt zu einer externen Operationszentrale hat, unmöglich wird. In dieser Arbeit wird ein Pfadplanungsansatz vorgestellt, der mithilfe einer Heuristik die Orientierung und Kontaktpunkte des Roboters mit dem Boden schätzt. Dadurch ist der vorgestellte Ansatz in der Lage stabile Pfade in Echtzeit auf limitierten Hardwareressourcen zu planen. Der Ansatz wird auf einem Roboter und in Simulation getestet. Auf dem Roboter werden die Echtzeitfähigkeit und der Einfluss eines parametrisierten Kompromisses zwischen Fahrtzeit und Stabilität demonstriert. In der Simulation wird die Genauigkeit der Heuristik evaluiert. Außerdem wird die Berechnungszeit für verschiedene Graphauflösungen evaluiert.

Contents

1. Introduction

In this chapter, first, the motivation for the presented work is given. After that, the robot system this work was evaluated on is presented, and finally, an overview of the chapters in this work is given.

1.1. Motivation

Rescue missions aim to save human lives from the consequences of catastrophic events. A likely event would be a partly collapsed building as a result of an earthquake. Generally presented, however, are much rarer but more impactful events such as nuclear meltdowns. The most recent occurrence of a nuclear meltdown happened in Fukushima, 2011.

Rescue missions in dangerous environments such as a highly radioactive partially collapsed nuclear plant put the human rescue force at high risks. In contaminated environments, the duration of the rescue mission not only increases the risk for the rescue force but directly increases the effects on the humans' health.

Figure 1.1.: Nuclear disaster in Fukushima, 2011. Image from [1].

Ongoing research in rescue robotics aims to reduce the need to risk human lives in dangerous rescue missions. One of the first steps to achieve this goal is having a robotic platform that can enter the rescue scenario, e.g., a partly collapsed building, create a map of the environment and locate potential victims. This information can be used to assist the human rescue team in planning the rescue operation, thereby significantly reducing the involved risk.

Large buildings pose significant problems for these robotic platforms. The walls effectively block most forms of wireless communication, and cables can be very restrictive to the robot's movement. For these reasons, teleoperation of the robot by a human operator is often infeasible, and the robot needs to be able to do all of the tasks above fully autonomous.

If the connection to the robot is lost, the robot is required to complete its mission without human intervention. While creating a complete map of the environment, the robot has to be able to plan paths and move through the terrain autonomously. This work focuses on two aspects of path planning. First, the robot has to plan stable paths that it can follow without tipping over. Second, the building can have multiple levels; hence, the planning approach has to be able to generate a plan over more than a single level.

1.2. Current System

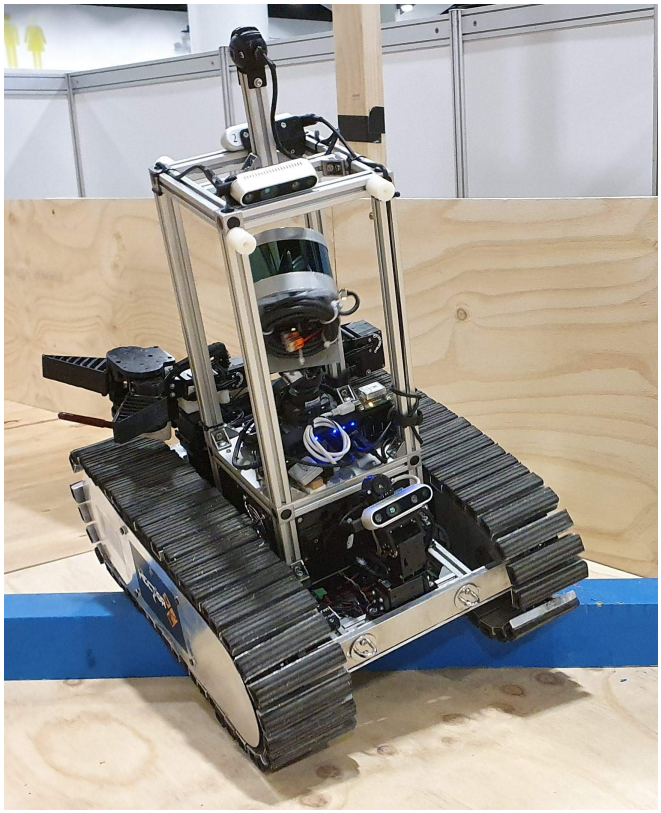


Figure 1.2.: The robot Jasmine at Robocup in Sydney, 2019.

Pictured in figure 1.2 is our lab's robot, *Jasmine*, a tracked robot with a sensor box on top. While the robot does not have flippers[1], it has a very low center of mass, allowing it to overcome obstacles of up to 10 cm. A 3D SLAM (*Simultaneous Localization And Mapping*) algorithm using the data provided by a Velodyne VLP-16 sensor is employed to build a map and localize itself in unknown environments. The software runs on an Intel NUC mini PC with an i7-8650U 15W quad-core processor.

In its current state, we use a 2D planning algorithm based on a simple OccupancyGrid (see 3.1 for a more detailed overview). Additionally, to prohibiting exploration across multiple levels, it does not take the terrain structure into account. Paths planned using this approach are guaranteed to be stable because any obstacle over a chosen threshold height is labeled as occupied and therefore can not be part of a planned path. However, although the current approach will not plan unstable paths, it achieves this at the cost of not finding existing stable paths that are filtered by the chosen threshold which is required to be conservative in order to guarantee stable paths.

[1] Reconfigurable tracks that can be used to adapt to the shape of the terrain

1.3. Overview

Chapter 2 will give a short introduction to the concepts and notations used in this work. Subsequently, a summary of existing research in this field is given in chapter 3. The proposed concepts and methods are explained in detail in chapter 4. Chapter 5 explains implementation details such as algorithm modifications and measures used to speed up or improve the stability of the computations. The algorithms developed as part of this work are evaluated in chapter 6. Finally, a conclusion and possible future improvements are listed in chapter 7.

2. Foundations

This chapter will first define the mathematical notations used in this work. Next, transformations using homogenous coordinates, heightmaps, and bounding boxes are introduced. Finally, applied concepts such as the *Unified Robot Description Format* and the *Force Angle Stability Measure* are presented.

2.1. Notation

Scalars

$$\rho \qquad \mu_{contact} \qquad {}^{M}z_{robot}$$

Scalar variables use a lower case roman or greek letter. If they represent a dimension in a coordinate frame, the frame is declared using an upper prescript. If deemed necessary, additional information in the form of a descriptive word is added as a lower postscript.

Vectors

$${}^{H}\boldsymbol{t} \qquad {}^{W}\boldsymbol{p} \qquad {}^{W}\boldsymbol{o}_{robot}$$

Vectors are written in bold with an upper prescript denoting the coordinate frame they are relative to. If a particular scalar value is needed, it is denoted with its index – or x for the first dimension, y for the second and z for the third – as a lower postscript, e.g., $\boldsymbol{p}_1$ or $\boldsymbol{p}_x$.

Vector Operators

$\leq$ The less-or-equal operator for vectors is defined as all elements of vector $\boldsymbol{a}$ are smaller than the corresponding elements of vector $\boldsymbol{b}$

$$\boldsymbol{a} \leq \boldsymbol{b} \quad \leftrightarrow \quad \forall i \in \{1, \cdots, D\} : \ \boldsymbol{a}_i \leq \boldsymbol{b}_i \qquad \text{for} \quad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^D$$

Euclidean Norm $\|\cdot\|_2$

$$\left\|\boldsymbol{a}\right\|_2 = \sqrt{\boldsymbol{a}^T \cdot \boldsymbol{a}} = \sqrt{\boldsymbol{a}_1^2 + \cdots + \boldsymbol{a}_D^2} \qquad \text{for} \quad \boldsymbol{a} \in \mathbb{R}^D$$

Matrices and 2D-Arrays

$$M_{contact} \qquad\qquad HM_{robot}$$

Matrices are denoted as uppercase letters with optionally a descriptive lower postscript. In this work, no distinction between matrices and 2D-arrays is made.

Special vectors and matrices

$$\mathbf{0} = \begin{bmatrix} 0 & \cdots & 0 \\ \vdots & \ddots & \vdots \\ 0 & \cdots & 0 \end{bmatrix} \qquad\qquad \mathbf{1} = \begin{bmatrix} 1 & 0 & \mathbf{0} \\ 0 & \ddots & 0 \\ \mathbf{0} & 0 & 1 \end{bmatrix}$$

2.2. Transformations

$${}^W R_R \qquad\qquad {}^W \boldsymbol{t}_R \qquad\qquad {}^W T_R$$

Transformations are used to transform vectors from one reference coordinate system to another. Since all coordinate systems are right-handed and have the same scaling, the transformation can be expressed as a combination of a 3x3 rotation matrix ${}^W R_R$ and a translation vector ${}^W \boldsymbol{t}_R$. The postscript denotes the source coordinate frame and the prescript the target coordinate frame. In other words, the example transformation transforms a vector from the coordinate frame R to the coordinate frame W.

$${}^W \boldsymbol{p} = {}^W R_R \cdot {}^R \boldsymbol{p} + {}^W \boldsymbol{t}_R$$

The transformation can be simplified into a single matrix multiplication by using homogenous coordinates. The transformation into homogenous coordinates is as follows:

$${}^R \boldsymbol{p} = \begin{bmatrix} x \\ y \\ z \end{bmatrix} \rightarrow {}^R \boldsymbol{p}_{hom} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

This allows to use a transformation matrix of the following form:

$$^{W}T_{R} = \left[\begin{array}{c|c} ^{W}R_{R} & ^{W}\boldsymbol{t}_{R} \\ \hline \mathbf{0}^{T} & 1 \end{array}\right] \qquad\qquad ^{W}\boldsymbol{p}_{hom} = {}^{W}T_{R} \cdot {}^{R}\boldsymbol{p}_{hom}$$

with the advantage that multiple transformations can be chained into a single transformation matrix:

$$^{W}T_{U} = {}^{W}T_{R} \cdot {}^{R}T_{U}$$

The resulting vector has to be transformed back into non-homogenous coordinates using:

$$^{W}\boldsymbol{p}_{hom} = \begin{bmatrix} x \\ y \\ z \\ \omega \end{bmatrix} \rightarrow {}^{W}\boldsymbol{p} = \frac{1}{\omega} \begin{bmatrix} x \\ y \\ z \end{bmatrix}$$

Since the transformations, using a transformation matrix of the given form, do not change the last element of a homogenous vector, the back transformation simplifies to dropping the last element. From now on, this transformation to homogenous coordinates and back is assumed implicit whenever necessary and will not be written down explicitly.

Inverse Transform While the transformation matrix $^{W}T_{R}$ can be inverted to obtain the inverse transform $^{R}T_{W}$ with

$$^{W}T_{R} \cdot {}^{R}T_{W} = \boldsymbol{I}$$

the inverse transform can be calculated more efficiently by splitting it back into a rotational and a translational part where the inverse is given as

$$^{R}R_{W} = \left({}^{W}R_{R}\right)^{-1} = \left({}^{W}R_{R}\right)^{T} \qquad\qquad ^{R}\boldsymbol{t}_{W} = -{}^{R}R_{W} \cdot {}^{W}\boldsymbol{t}_{R}$$

2.3. Heightmaps

Technically speaking, heightmaps or elevation maps are just 2D-arrays of elevation values with their location implicitly encoded by their row and column index. They are explained in more detail in 4.1.1; however, for convenience, some functions and notations used in this work are defined here.

Heightmaps are denoted as HM. If the value for a specific location is wanted, HM doubles as a function $HM : (\mathbb{N}, \mathbb{N}) \rightarrow \mathbb{R}$, accessing the stored elevation value for a given location $HM(x, y) = z$. This function may not be defined for all possible values $(x, y) \in \mathbb{N} \times \mathbb{N}$.

Another function used in this work is the minimum function min : $\mathbb{N} \times \mathbb{N} \rightarrow \mathbb{R}$, which for a given heightmap returns the minimum of all defined values $\min(HM) = z_{HM,min}$.

2.4. Bounding Box

A bounding box is a cuboid of minimal dimensions that fully encloses an arbitrarily complex shape. The bounding boxes used in this work are axis-aligned and therefore not rotated relative to their parent coordinate frame. There are several methods of defining a bounding box, e.g., using the center $\boldsymbol{c}$ and half the size in each dimension: $\frac{l}{2}, \frac{w}{2}, \frac{h}{2}$. The latter represent the length, width, and height, respectively. They represent the dimensions in x-, y- and z-direction.

In this work, two points are of particular interest. The first is the already mentioned center of the bounding box, denoted as BB_{center}. The second is the bounding box's minimum BB_{min}, which is the bounding box's corner, where each dimension is minimal. Despite not marked in bold, these particular locations are also vectors.

2.5. URDF

The **U**nified **R**obot **D**escription **F**ormat (URDF) is an XML format for the declaration of robot models. It allows representing the robot model as a hierarchy of links and joints. Additional information such as sensors, transmissions, simulation properties, etc. can also be declared but will not be detailed here as they are not relevant in the context of this work.

Links Link elements represent rigid bodies with inertia, visual, and collision properties.
The visual and collision properties both describe the shape of the link using geometries, which can be a *box* (or cuboid), *cylinder*, *sphere*, or a *mesh*.
The difference between visual and collision is that the visual is intended for visualization and therefore can be more complex and also use materials for the geometry whereas the collision geometry is used for computations and these are less expensive the simpler the geometry, i.e., no meshes and a reduced number of geometric shapes.
The collision geometry is often a trade-off between accurate surface representation and computational cost.

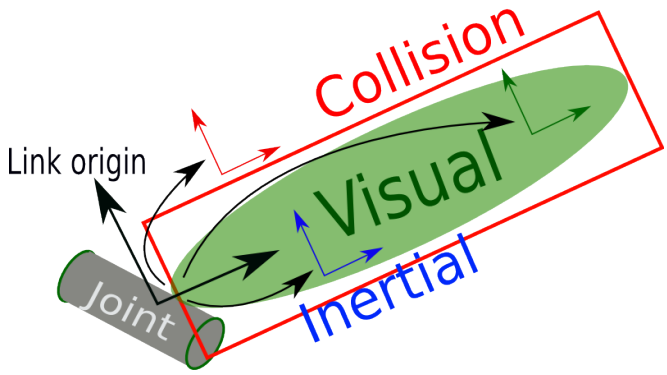


Figure 2.1.: Sketch of a link element. Image from [2].

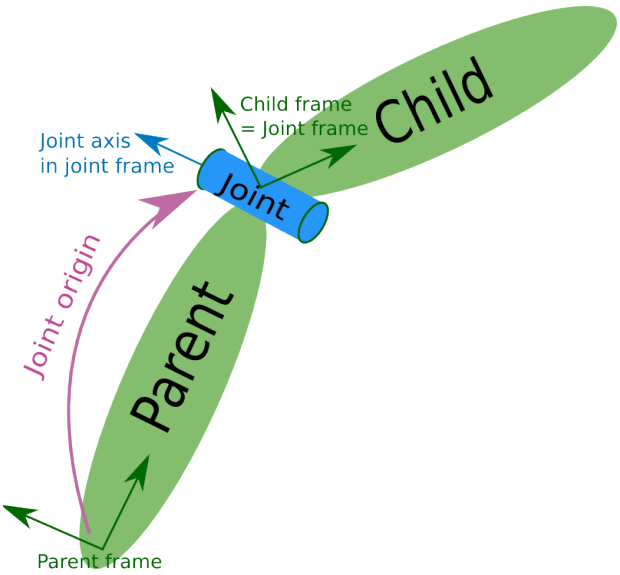


Figure 2.2.: Sketch of a joint element. Image from [3].

Joints The joint element describes the kinematics and dynamics of a joint in the robot model. It can also specify the joint's actual and safety limits. There is a number of different joint types, the most relevant being revolute or continuous joints which are both hinge joints rotating around an axis. The difference is that the former has an upper and lower limit, whereas the latter can turn continuously. Noteworthy are also prismatic joints that can slide along an axis, e.g., to extend the reach of a robotic arm, and fixed joints which are not actually joints but merely a fixed connection of two links that can not move.

All joint types have in common that they connect two links, a parent and a child link. For all joint types except for the fixed joint, the transformation from the parent to the child frame depends on the state of the joint. The joint state is a scalar or vector describing the values of the joint variables. For example, for a revolute joint, the joint state would be determined by the rotation angle.

2.6. Force-Angle Stability Measure

Stability measures quantify the stability of a vehicle. The higher the measure, the more resilient the vehicle is to external forces that may tip it over. The stability measure used in this work is the Force-Angle stability measure first introduced by Papadopoulos and Rey[4]. It was chosen due to its sensitivity to top-heaviness and its straightforward and efficient computation.

Figure 2.3 depicts the Force-Angle stability measure for a planar wheeled vehicle. A net force[1] f_r is acting on the vehicle's center of mass (COM). The contact points with the ground and the COM form the vectors l_1 and l_2. Given these vectors, the angles between $l_{1,2}$ and f_r can be computed as $\theta_{1,2}$. The Force-Angle stability measure is given as the minimum of these angles times the magnitude of the net force f_r for heaviness sensitivity.

$$\alpha = \theta_1 \cdot \|f_r\|$$

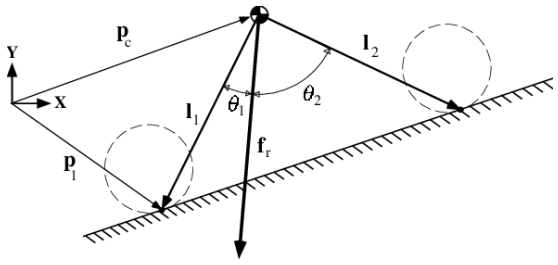


Figure 2.3.: "Planar Force-Angle stability measure." Image from [4].

[1]The sum of all forces acting on the vehicle, excluding the supporting reaction forces.

The assumption is that heaviness contributes to the stability which holds for the static stability and slow-moving vehicles. The opposite is true for fast vehicles.
As the minimal angle θ approaches zero, the vehicle becomes less stable and tipping over requires less force. Once the angle becomes negative, the vehicle tips over.

The top-heaviness sensitivity is given by the angles increasing the lower the COM is located. Vehicles that can move their COM may trade-off other properties such as sensor coverage for a lower COM when necessary.

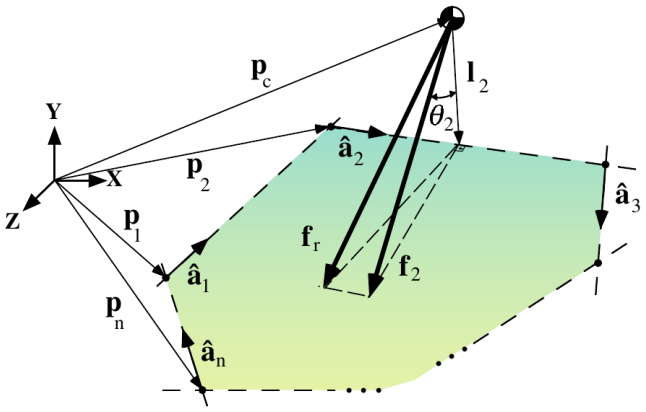


Figure 2.4.: "3D Force-Angle stability measure." Image from [4].

Analogous, the general form employed in the 3-dimensional case uses the vehicle's convex support polygon which is given by the convex hull of all ground contact points.

Let $\boldsymbol{p}_i$ be the i-th ground contact point that is part of the convex hull.

$$\boldsymbol{p}_i \in \mathbb{R}^3 \qquad i = \{1, \ldots, n\}$$

with $\boldsymbol{p}_c$ being the location of the COM.

The $\boldsymbol{p}_i$ are ordered clockwise when viewed from above. Each consecutive pair of contact points forms a candidate tip-over axis $\boldsymbol{a}_i$

$$\boldsymbol{a}_i = \boldsymbol{p}_{(i \bmod n)+1} - \boldsymbol{p}_i \qquad i = \{1, \ldots, n\}$$

Tipping will always happen over one of these axes unless an obstacle is encountered by one of the ground contact points or the ground conditions change.

Letting $\hat{\boldsymbol{a}}_i$ be the normalized tip-over axis $\boldsymbol{a}_i$, the tip-over axis normal $\boldsymbol{l}_i$ that intersects the vehicle's COM can be obtained by projecting the vector $\left(\boldsymbol{p}_{(i \bmod n)+1} - \boldsymbol{p}_c\right)$ from the COM to the axis' second ground contact point onto the plane that is orthogonal to the candidate tip-over axis.

$$\boldsymbol{l}_i = \left(\mathbf{1} - \hat{\boldsymbol{a}}_i \hat{\boldsymbol{a}}_i^T\right)\left(\boldsymbol{p}_{(i \bmod n)+1} - \boldsymbol{p}_c\right)$$

where $\mathbf{1}$ is the 3×3 identity matrix.

The same projection is used to obtain the portion of the net force $\boldsymbol{f}_r$ that acts about the tip-over axis.

$$\boldsymbol{f}_i = \left(\mathbf{1} - \hat{\boldsymbol{a}}_i \hat{\boldsymbol{a}}_i^T\right)\boldsymbol{f}_r$$

For a thorough description of the modeling of $\boldsymbol{f}_r$ and how to incorporate moments, see [4]. In this work, all forces except for the gravity acting on the COM are disregarded, and it is assumed that no significant moments are acting on the robot.

With $\hat{\boldsymbol{f}}_i$ being the normalized projected force vector and $\hat{\boldsymbol{l}}_i$ being the normalized tip-over axis normal intersecting the COM, the angle between $\hat{\boldsymbol{f}}_i$ and $\hat{\boldsymbol{l}}_i$ is given by

$$\theta_i = \sigma_i \cdot \cos^{-1}\left(\hat{\boldsymbol{f}}_i \cdot \hat{\boldsymbol{l}}_i\right)$$

where the sign of the angle θ_i is determined by

$$\sigma_i = \begin{cases} +1 & \text{if} \quad \left(\hat{\boldsymbol{f}}_i \times \hat{\boldsymbol{l}}_i\right) \cdot \hat{\boldsymbol{a}}_i < 0 \\ -1 & \text{otherwise} \end{cases}$$

Finally, the Force-Angle stability measure is obtained as

$$\alpha = \min_{i \in \{1,\ldots,n\}} \theta_i \cdot \left\| \boldsymbol{f}_r \right\|$$

3. Related Work

In the field of robotics research, and rescue robotics, in particular, path planning is one of the fundamental problems and an actively researched topic that has seen much attention in the past decades. While general path planning can usually be reduced to finding any viable path from start to goal, in rescue robotics, additional costs such as the traversability and the robot's stability, additional objectives such as sensor coverage, and live replanning due to incremental map updates either as a result of changes in the environment or previously unknown areas, have to be incorporated.

Path planning can be done in 3D space which is capable of encompassing the real world in its entirety, or the problem can be reduced to planning in 2D which simplifies the planning by a significant margin but also limits the areas that can be represented to a single level. Path planning in 2D is a mostly solved problem, and current research focuses on the extension using either additional constraints and objectives or dropping optimality constraints in favor of more time-efficient algorithms. An overview of early 2D path planning approaches and the history of robot motion planning can be found in [5].

3.1. Current System

The path planning software currently in use[6] is a 2D path planning algorithm using a cost map representation based on an occupancy map and a modified implementation of the exploration transform[7].

Occupancy maps (alt. occupancy grids) were first introduced by Moravec[9] as a 2D grid representation of the world, encoding in each cell the probability that it is occupied. When planning paths on an occupancy map, the robot is only allowed to pass through free cells, i.e., cells for which the probability that they are occupied is less than 0.5.

The robot itself is represented as a single point without radius. Instead of modeling the robot's dimensions, collisions are avoided by blowing up obstacles.

The path is obtained by using the exploration transform which finds the boundaries from free cells to unknown cells in the occupancy map and in a flood-fill fashion, the distance to the closest boundary large enough for the robot to fit is calculated for each cell. Additionally, a cost term penalizing paths closer than a minimal distance to a wall is used. Finally, the path is obtained by following the largest gradient.

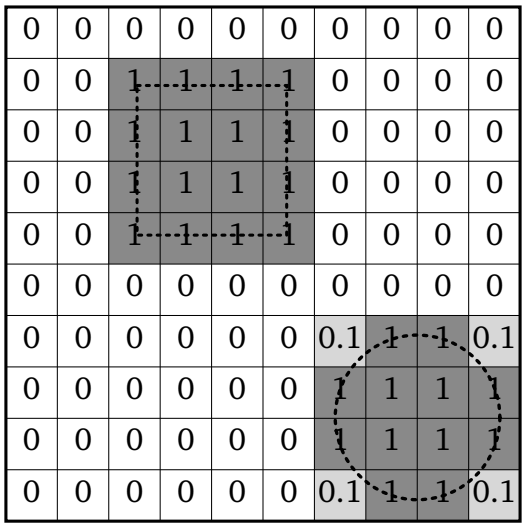


Figure 3.1.: An example of an occupancy grid. Image from [8].

Planning to a goal instead of exploring is done in the same manner, but instead of the edges between free and unknown cells, the goal is set as the boundary.

3.2. 3D Path Planning

One of the most significant limitations of 2D path planning is that a 2D representation prohibits us from taking the 3-dimensional structure of the ground into account. This requires labeling anything that may not be traversable in one specific configuration – usually based on height differences – as an obstacle regardless of whether the robot would be capable of traversing it or not.

The limitation of binary classification of the traversability in 2D planning is overcome in 3D planning where the terrain can be represented, and the traversability can be estimated on a continuous scale. This, however, comes at the cost of a larger search space and constraints that have to be satisfied.

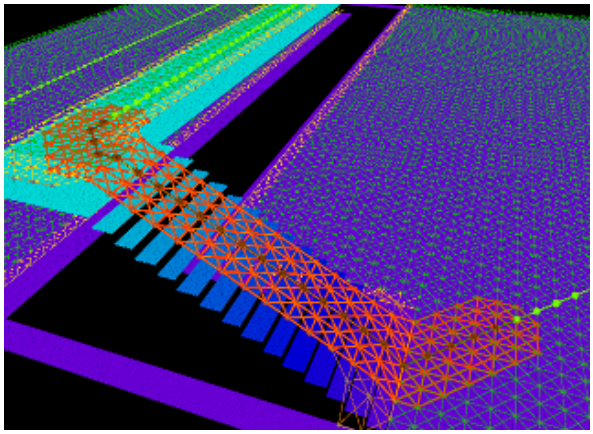


Figure 3.2.: Graph search restricted to tube around initial path. Image from [10].

Brunner, Brüggemann, and Schulz proposed a path planning algorithm that first computes a risk map based on height differences in the heightmap representation of the real world[10]. To avoid the narrow passage problem, first, a graph search is done on the risk map to find an initial path which is consequently used to restrict a search of the vast search space of robot configurations to a tube around the initial path.

Additionally to the path, they also optimize the configuration of their robot's (a Telerob Telemax) four actuated tracks based on the Normalized Energy Stability Margin which is a stability measure based on

the force necessary to tip a robot over an axis of its support polygon and provide a configurable tradeoff between safety and execution time of the path.

One limitation of their approach is the requirement of a static previously known and preprocessed map. They do not mention the planning time nor whether their world representation facilitates the planning of paths on more than a single level, which is a known limitation of heightmap representations.

Colas, Mahesh, Pomerlau, Liu, and Siegwart introduced a graph search based path planning algorithm that operates on a statically loaded 3D map consisting of point cloud data from a LIDAR sensor[11]. A dense representation of the map is obtained by lazy tensor voting at each graph node and used to determine possible neighbor nodes to expand to based on the lack of obstacles, a minimal surface to support the robot and a surface plane within the roll and pitch constraints. The path from the start to the goal is found using the D*-Lite graph search algorithm. Finally, a robot configuration is chosen for each node based on the geometric properties of each position. The limitations of their approach are the requirement of a static previously known map, a fixed number of predefined robot configurations, and initial planning times that are not real-time feasible.

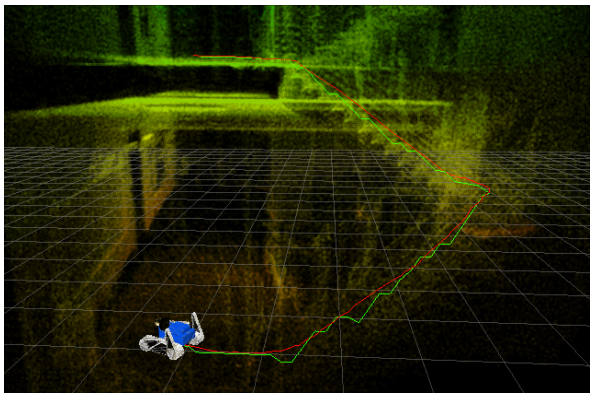


Figure 3.3.: "Climbing down the stairs". Image from [11].

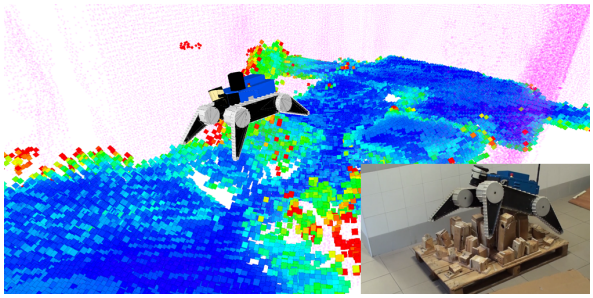


Figure 3.4.: Traversability map while traversing rubble. Colors indicate traversability cost from blue (low) to red (high). Image from [12].

Ferri, Gianni, Menna, and Pirri propose an approach capable of planning directly on dynamic point cloud data[12]. They use one set of light features such as surface variation and normals for points far away from the robot and more accurate but also computationally more expensive features such as the principal curvatures closeby. Using the curvatures and normals, the point cloud segments are labeled as one of the predefined classes: *wall*, *terrain*, *surmountable obstacle*, and *stairs/ramp*.
As opposed to the previously mentioned approaches, they can deal with on-line map updates including dynamic obstacles and use a sampling-based planner based on randomized A* for which they prove that it behaves like A* in the worst case and that the path cost can not exceed those of a path found by A*.
Limitations include the requirement of predefined classes for the traversability. Nevertheless, it should be noted that the classes are reasonably general and only influence the cost term as a heuristic. They do not evaluate the robot's stability along the path. There is also no mention of the computation time, which can be significant when processing 3D data.

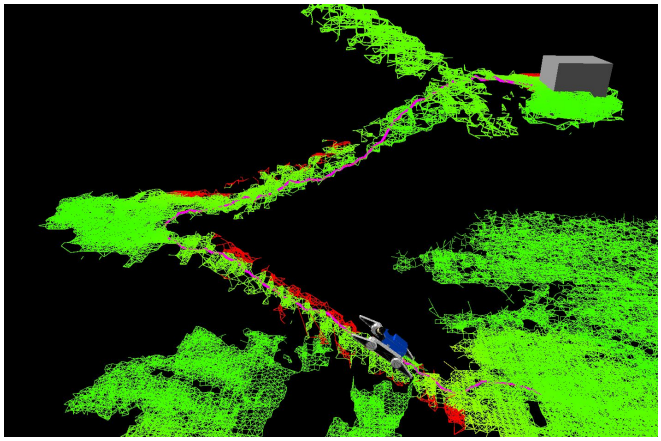


Figure 3.5.: Path (magenta) on fire escape stairs. Image from [13].

Menna, Gianni, Ferri, and Pirri also propose a real-time capable 3D navigation algorithm[13]. As opposed to the previously mentioned algorithm, they do not differentiate between points near and far from the robot. It also operates on point cloud data, which is segmented, clustered, and classified as one of the predefined classes: *ground*, *walls*, *ramp/stairs*, and *surmountable obstacles*. A traversability graph is built by inflating all clusters except for those labeled as *walls* and connecting the points that within half a robot length in euclidean distance. Finally, the path is planned on a graph using the Dijkstra algorithm. While the planning time is real-time with planning times in the low single-digit seconds, there remain limitations such as the need for predefined classes that were handcrafted for the specific capabilities of their robot and no method of evaluating the stability of the robot along the path.

Norouzi, Miro, and Dissanayake propose a strategy to generate stable paths for reconfigurable robots[14]. They use a *Packbot* which is a tracked robot with one pair of flippers and a 1 Degree of Freedom (DoF) shoulder joint to which a 2 DoF sensor unit with several cameras and lights is attached. A 3D mesh is used as a map representation which has the advantage over heightmaps that it does not require a fixed discretization of the mapped space but remains a more compact representation than point clouds. The contact points of the robot on the map at a given location are estimated using the physics simulation ODE (Open Dynamics Engine). Based on these contact points, the stability is evaluated using the Force Angle Stability Margin and simultaneously optimized by moving the center of mass using the 1 DoF shoulder joint. Finally, the path is planned both with a grid-based approach using A* and a sampling-based approach using the rapidly exploring tree (RRT) algorithm. The computation time for the path generation is not mentioned.

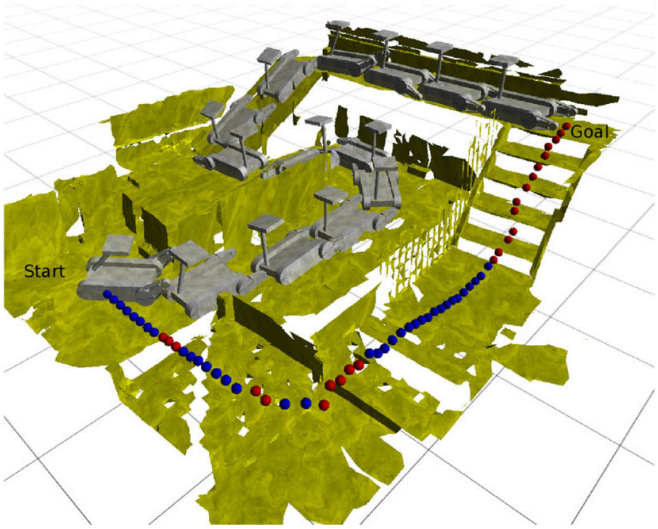


Figure 3.6.: "Standard A* path shown in blue, with unstable regions in red. Robot poses derived from the most stable with lowest reconfiguration cost A* path are outlined in light grey." Image from [14].

Sebastian and Ben-Tzvi propose a path planning architecture that utilizes the simulator Bullet Physics to simulate the closed-loop motion from a graph node to its neighbor[15]. By using a physics-based simulation, they want to incorporate slip, terrain slope, actuator limits, and the robot's dynamics. While the idea to plan the path using a simulation can be expected as an approach that is going to be the state of the art once it is computationally feasible to do in real-time, the approach currently has significant limitations. Given that it takes several tenths of a second to evaluate a single edge in the graph, it is simply not feasible for global planning where a global graph for a small map with a size of 3 m by 10 m with a resolution of 25 cm already consists of multiple hundred nodes and, in the case of an eight-connected graph, upwards of a thousand edges.

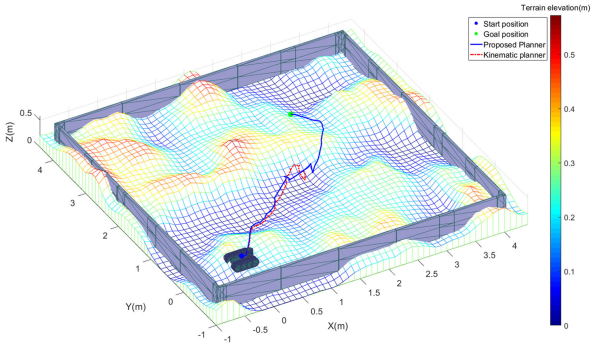


Figure 3.7.: "Terrain topography map showing the 3D path followed by the robot under both the planners in extreme terrain simulation". Image from [15].

4. Method

4.1. Map

A plentitude of terrain representations with different strengths and weaknesses has been proposed. In this section, first, the different representations are presented and evaluated in the context of this work. Then, the proposed map representation is explained in detail. Due to their inability to capture the three-dimensional structure of the terrain, 2D map representations are not part of this evaluation.

4.1.1. Map Representations

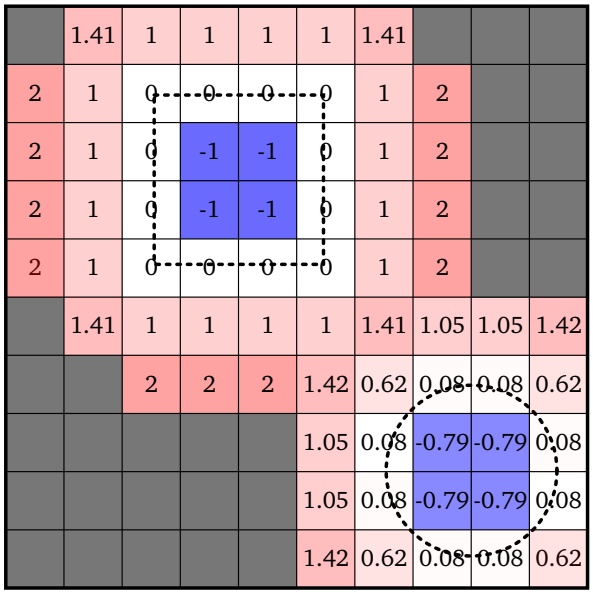


Figure 4.1.: An example of a TSDF. Image from [8].

[Truncated] Signed Distance Fields ([T]SDF)
Extended from cells in the two-dimensional case, voxels [1] in the three-dimensional adaptation represent the distance to the closest surface – negative distances are inside of the object. In truncated SDFs[16][17], the voxels only store the distance to the closest surface up to a truncation distance. This representation is state of the art for autonomous UAVs (Unmanned Aerial Vehicles) because looking up the distance to the closest wall for a given 3D location is $\mathcal{O}(1)$. For UGVs (Unmanned Ground Vehicle), it is not as well suited due to their limitation to a 2-manifold of 3D space. Estimating the robot's pose at a given location in the world requires knowledge of the ground contact points. (T)SDFs, however, do not differentiate between ground and walls; hence, the distance measurement of a single voxel is of no use. In short, most advantages of (T)SDFs are of little use for this application, and the downsides like higher computational effort and higher memory consumption outweigh.

[1] Similarly to the division of a 2D surface into pixels or cells of a grid, a voxel is a cube representing a discretization of 3D space.

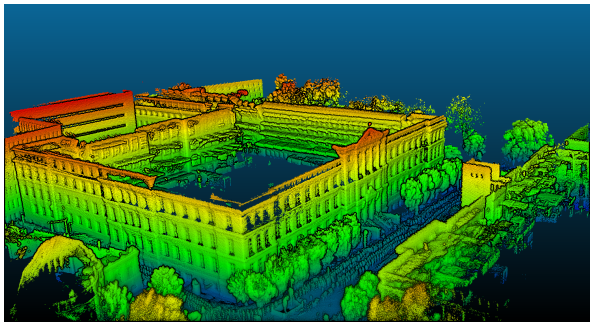


Figure 4.2.: Point cloud of a building. Colored based on height.

Point clouds Raw or filtered point clouds can accurately represent any type of terrain since the accuracy of their representation is only limited by the sensors, not by any inherent limitation of the approach, e.g., due to discretization. They are very demanding on memory, and while efficient lookup structures such as Octrees exist, access is still comparatively time-consuming. Given the real-time constraints, it was dismissed as a representation after some short experiments. The author does not in any way imply that point clouds can not be of use in real-time 3D path planning as different implementations of efficient storage partitions may significantly improve results.

Polygon Mesh A representation for the surface of 3D objects commonly used but not solely in the video game industry due to their ability to represent any shape up to any level of detail efficiently. The most commonly referred to mesh consists of a collection of surface triangles. Hence, a mesh can be seen as a collection of polygons defined by the corner points in world coordinates. This representation is one of the most memory-efficient of the presented and can represent the terrain to any degree of accuracy. Unfortunately, to the author's knowledge, there was no public algorithm available for or adaptable to our robot that can generate a 3D mesh online.

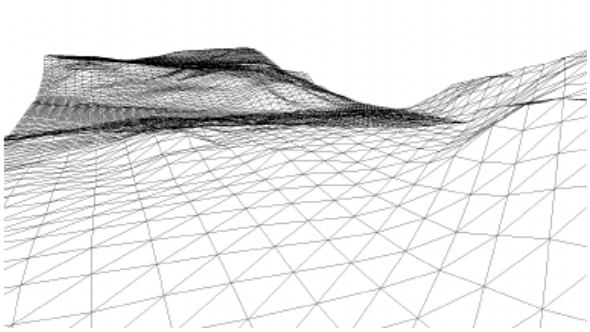


Figure 4.3.: Terrain mesh. Image from [18].

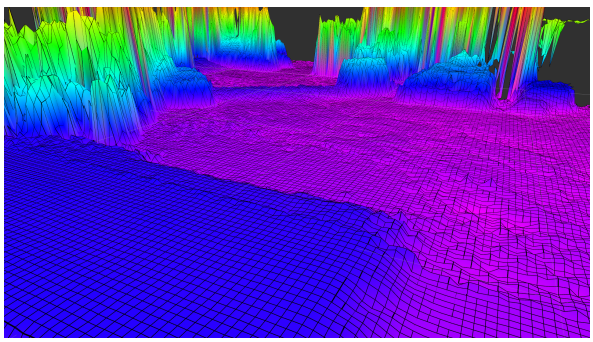


Figure 4.4.: An example of a heightmap. Colored based on the elevation.

Heightmaps Strictly speaking, height or elevation maps are not 3D representations but 2.5D representations. They generally only represent the elevation relative to a flat plane and thereby cannot represent more than a single layer. However, extensions to multiple layers exist. Heightmaps are capable of very efficiently representing the ground at the cost of not being able to represent structures. This is not a limitation for this work since solely the ground is of interest when estimating the pose of the robot.

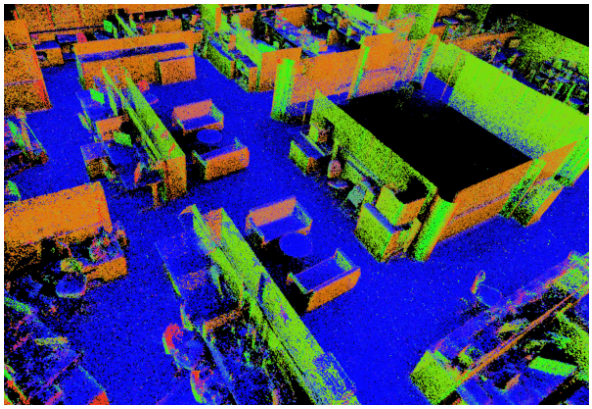


Figure 4.5.: Surfel map of an office. Image from [19].

Surfel Maps An alternative to polygon meshes, surfel (**surf**ace **el**ement) maps [20] represent the surface of an object using a set of surface elements. A surface element is a planar disk defined by a center point, a normal vector and a radius. Surfel maps share many of the advantages of polygon meshes, namely, the memory efficiency and the ability to capture terrain structure with high accuracy. While multiple real-time surfel mapping approaches exist, there is as of the time of writing no adaptation to our robot.

4.1.2. Proposed Representation: Bag of Heightmaps

Heightmaps are locally a sufficient approximation of the ground – only lacking the ability to represent multiple levels. Multiple connected heightmaps are used to extend the mapping ability from a local to global representation. The requirement for this extension to be valid is that the terrain can locally be represented by a single level heightmap. This precondition is only violated if the robot is capable of traversing obstacles that are higher than its minimal height, which is generally only possible if the robot has reconfigurable shape, e.g., using flippers. Of the robots in our lab, none violate this precondition.

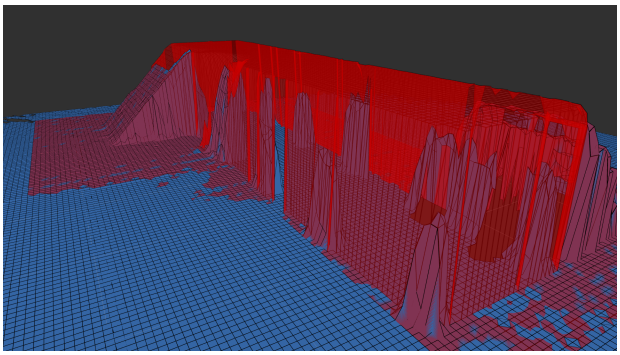


Figure 4.6.: Two separate partly overlapping heightmaps in blue and red.

Attachments Additionally to the terrain structure represented by the heightmap, each entry – which we define as an instance of a heightmap combined with other attributes – also allows for map attachments. These attachments also represent a map of floating-point values with the same size and resolution as their base heightmap. The meaning of those values depends on the attachment. One example of such an attachment would be the distance to the closest non-traversable height difference. Here, non-traversable is decided by using a threshold for the height difference is certainly not traversable for the robot, e.g., because it is higher than the robot's tracks.

Map Updates Attachments have to be synchronized with the map. This means whenever the map changes, the updates have to propagate to the map attachments. The need for synchronized updates is not limited to attachments, in any case. Information about locations in the map that depends on the terrain structure such as the estimated pose could be cached as long as the relevant part of the map does not change. For this reason, the map allows registering for updates and notifies registered subscribers about map changes.

4.2. Graph

A three-dimensional graph is required to facilitate planning across multiple levels. However, given that the robot is restricted to a two-manifold because it can not leave the ground, the graph is locally two-dimensional.
The graph is built by starting at a given start position, which is usually the robot's current position and expanding in all directions.
For this expansion, an x- and y-offset is added to the current node's position whereas the z-coordinate is initially inherited. The value of the heightmap is requested at the initial position, and the z-value of the expanded node is replaced by the value in the map.
If there is no valid value in the heightmap for the given position, the expanded node is rejected.
Further, this expansion is controlled by a *NodeFilter* which serves as an early rejection of unreachable nodes.

4.2.1. Node Filters

Many candidate positions for the robot can be rejected quickly, for example, if they are too close to a significant height difference in a heightmap. Filtering these nodes early is beneficial because it prevents unnecessary costly evaluations. It should be noted that they are merely an optimization and not required to ensure proper planning as invalid positions would still be rejected by the pose prediction.

In this work, a node filter based on the distance to significant height differences is used. It operates in conjunction with the map attachments presented in 4.1.2. For a given position, the node filter looks up the value in the attachment representing the distance to the closest non-traversable height difference and rejects the proposed node if the value is below a given threshold, e.g., the minimum of the robot's length and width.

4.3. Pose Prediction

Pose prediction is the problem of predicting the pose of a robot on a virtual landscape at a location that the robot has possibly never traversed before based on available data. An accurate prediction of the robot's pose permits estimating other crucial properties such as the robot's stability at the given location. A sufficiently accurate pose prediction is a fundamental requirement for any planning algorithm aiming to plan stable paths.

As an efficient heuristic to approximate the robot's pose on a given terrain, an approach using heightmaps is introduced.
For a given location ${}^{W}\boldsymbol{p}$ in the world frame W, first, a heightmap of the robot in an initial orientation, e.g., identity or the orientation at the previous location, is generated from the robot's URDF and a

specified robot configuration with the same resolution ρ as the map. Let this heightmap be HM_{robot}, and the rigid transformation from the URDF's coordinate frame U to the heightmaps coordinate frame H will be referred to as ${}^{U}T_{H}$ [2].

Given the robot heightmap, a chunk of the same size is extracted from the map at position ${}^{W}\boldsymbol{p} + {}^{U}\boldsymbol{t}_{H}$ where ${}^{U}\boldsymbol{t}_{H}$ is the translation part of ${}^{U}T_{H}$. In the following section, this part of the global map is referred to as HM_{ground}. Both heightmaps are used to estimate the ground contacts of the robot with the ground for the given orientation. If the ground contacts do not form a stable support polygon, the robot is rotated around the support polygon's weakest axis, and the process is repeated for the new robot orientation.

4.3.1. Robot Heightmap Generation

The generation of the robot's heightmap is done in three steps.
First, the robot's bounding box BB for the given robot orientation ${}^{W}R_{U}$ is calculated. This is used to estimate the height and width of the heightmap which correspond to the bounding box's size in x- and y-direction respectively. Second, the size of the heightmap and the transformations are estimated. The number of rows and columns of the heightmap are set as

$$\text{rows} = \left\lceil \frac{\text{length}(BB)}{\rho} \right\rceil \qquad \text{columns} = \left\lceil \frac{\text{width}(BB)}{\rho} \right\rceil$$

Next, the transformation from the robot to the heightmap is needed. The heightmap should be flat in the world, i.e., it is a plane parallel to the x-y-plane. The robot itself is rotated with the given rotation ${}^{W}R_{U}$ relative to the world frame. The heightmap can be seen as the bottom side of the bounding box, and the center of the heightmap is equivalent to the center of the bounding box's x-y-slice and the bounding box's minimum z coordinate.

$${}^{H}R_{U} = {}^{W}R_{U} \qquad {}^{H}\boldsymbol{t}_{U} = -\begin{bmatrix} BB_{center,x} \\ BB_{center,y} \\ BB_{min,z} \end{bmatrix}$$

Finally, the transform is applied recursively in combination with the hierarchical link transforms to all geometries in the robot's URDF, each geometry is projected straight down with an orthogonal projection ray $\begin{bmatrix} 0 & 0 & -1 \end{bmatrix}^{T}$, and for each cell, the minimal distance is stored.

[2] Note that the transformation ${}^{U}T_{H}$ depends on the orientation and has to be recomputed for each orientation.

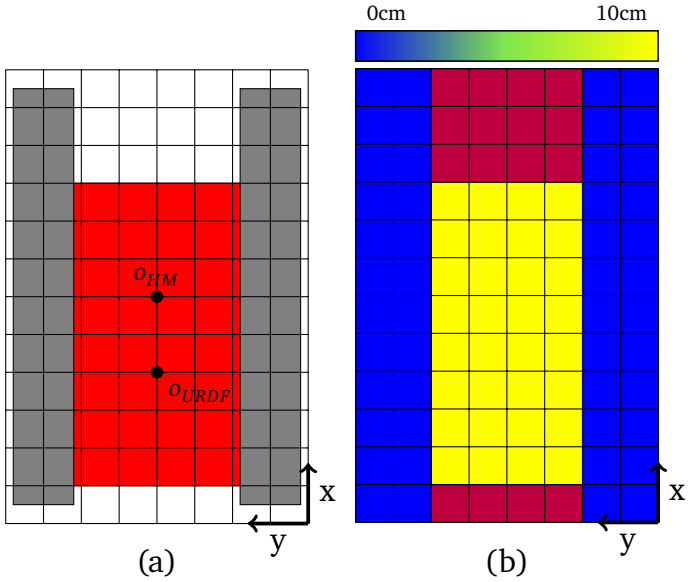


Figure 4.7.: Robot viewed from below (a) and the resulting heightmap (b). Grid visualizes discretization. Purple cells have no value due to no part of the robot intersecting with the cell.

4.3.2. Coordinate Transformations

Due to the heightmap being a two-dimensional function implicitly representing 3D points as a combination of their location and the value at the location, the transformation for a point ${}^{M}\boldsymbol{p}$ from the map frame to heightmap coordinates and value is a non-linear transformation given by the following equations

$$\begin{bmatrix} x_{2D} \\ y_{2D} \end{bmatrix} = \begin{bmatrix} \frac{1}{\rho} \cdot {}^{M}\boldsymbol{p}_x + \frac{rows-1}{2} \\ \frac{1}{\rho} \cdot {}^{M}\boldsymbol{p}_y + \frac{cols-1}{2} \end{bmatrix} \qquad HM(x_{2D}, y_{2D}) = {}^{M}\boldsymbol{p}_z$$

and the inverse transformation from a map coordinate and value to a point ${}^{M}\boldsymbol{p}$ in the map frame is given by

$${}^{M}\boldsymbol{p} = \begin{bmatrix} \left(x_{2D} - \frac{rows-1}{2}\right) \cdot \rho \\ \left(y_{2D} - \frac{cols-1}{2}\right) \cdot \rho \\ HM(x_{2D}, y_{2D}) \end{bmatrix}$$

4.3.3. Contact Estimation

The contact points of the robot with the grounds are estimated using the contact map $M_{contact}$, which is obtained by subtracting the ground heightmap HM_{ground} from the robot heightmap HM_{robot}.

$$M_{contact} = HM_{robot} - HM_{ground}$$

This contact map represents the distance of each cell in the robot heightmap to each cell in the ground heightmap. The minimum of $M_{contact}$ is the missing value necessary to estimate the z-coordinate of the robot in the map frame M.

$$\mu_{contact} = \min(M_{contact})$$
$${}^{M}z'_{robot} = {}^{M}z_{ground} - \mu_{contact} - {}^{M}z_{HM,robot}$$

To update the position of the robot in the world, the position has to be transformed into the map frame, the z-coordinate is replaced by ${}^{M}z'_{robot}$, and the result is transformed back into the world frame.

$${}^{M}\boldsymbol{p} = {}^{M}T_{W} \cdot {}^{W}\boldsymbol{p} \qquad {}^{M}\boldsymbol{p} = \begin{bmatrix} {}^{M}x_{robot} \\ {}^{M}y_{robot} \\ {}^{M}z_{robot} \end{bmatrix} \qquad {}^{M}\boldsymbol{p}' = \begin{bmatrix} {}^{M}x_{robot} \\ {}^{M}y_{robot} \\ {}^{M}z'_{robot} \end{bmatrix} \qquad {}^{W}\boldsymbol{p}' = {}^{W}T_{M} \cdot {}^{M}\boldsymbol{p}'$$

The contact points of the robot at its new position is the set of points

$$\{p : (x, y) \in p, M_{contact}(x, y) <= \mu_{contact} + \delta_{contact}\}$$

where $\delta_{contact}$ is a contact threshold accounting for map errors due to sensor noise and other error sources.

4.3.4. Support Polygon and Stability

Support Polygon The support polygon is estimated by collecting the location[3] of all contact points and estimating the convex hull using *Andrew's monotone chain convex hull algorithm* (see proc. 4.1). This algorithm was chosen because of its linear runtime if the set of points is sorted. This requirement is satisfied by the nature of collecting the candidate locations. Given that the contact map is traversed by row then column or column then row, the candidate locations are always sorted.

The algorithm works by separately constructing the upper and lower hull of the given points. It was adapted from its common form to construct the hull in clockwise order rather than counter-clockwise

[3]The location of a point in the heightmap is given by its row and column index.

Procedure 4.1 Monotone Chain (Andrew's algorithm)

Input: A list of points P with at least 3 distinct points
Output: Convex hull of given points in clockwise order

sort(P) {By x-coordinate, resolve ties with y-coordinate}
$upper \leftarrow [\ ]$ {Initialize upper and lower hull with empty list}
$lower \leftarrow [\ ]$
$n \leftarrow \text{size}(P)$
for $i = 1, \cdots, n$ **do**
    while $upper$ contains at least two points and the sequence of the last two points in $upper$ and $P[i]$ does not make a clockwise turn **do**
        remove last point from $upper$
    end while
    append $P[i]$ to $upper$
end for
for $i = n, n-1, \cdots, 1$ **do**
    while $lower$ contains at least two points and the sequence of the last two points in $lower$ and $P[i]$ does not make a clockwise turn **do**
        remove last point from $lower$
    end while
    append $P[i]$ to $lower$
end for
remove last point from $upper$ and $lower$
return $lower \cup upper$

to make it compatible with the stability measure which expects the contact points to be in clockwise order. The upper hull runs from the leftmost to the rightmost point in clockwise order. Hence, if the points are sorted, the points can be added one by one, and every previous point that does not make a clockwise turn with the added point would be a convexity and is removed. The lower hull is constructed analogous but from the rightmost to the leftmost point. Finally, the last point is removed from both the lower and upper hull as they are also the start of the other and they are concatenated to obtain the convex hull.

Stability As described in [4], the robot is stable if its projected center of mass is within its support polygon. Hence, to be stable, the robot needs at least three contact points. If it has less than three contact points, it is unstable.

Otherwise, the robot's stability given the estimated support polygon and the ground orientation needs to be estimated. To estimate the stability, we need the contact points as well as the robot's center of mass (COM). The COM is obtained by assuming each link as a point mass and building a weighted sum of each link's COM. To get the support polygon, the contact points are transformed from the image plane back to 3D space. For simplicity, the URDF frame is used as the reference frame.

The 2D points are transformed into 3D points, as described in section 4.3.2. To obtain the points in the URDF frame, the points are transformed by ${}^{U}T_{H}$

$$ {}^{U}\mathbf{c}^{(i)} = {}^{U}T_{H} \cdot {}^{H}\mathbf{c}^{(i)} $$

Given the obtained contact points, the stability for each axis between consecutive points is estimated using the Force-Angle stability measure (see 2.6) and the gravity force vector transformed from the world frame into the URDF frame as the acting force.

4.3.5. Rotation

If the robot is stable, the pose prediction is completed. Otherwise, tipping the robot over the unstable axis has to be simulated, and the process is repeated with the new orientation.

Finding the rotation axis An axis for the rotation is needed to estimate the required rotation. How this axis is determined depends on the number of contact points. For a single contact point, the rotation axis is given by the orthogonal to the line from the contact point to the projected center of mass. With two contact points, there is a distinct axis of rotation, and only the direction is needed. The direction can be determined given that the rotation happens in the direction of the projected center of mass. If there were three or more contact points, the axis of rotation is the least stable axis determined using the employed stability measure.

Finding the rotation angle A naive approach to finding the rotation amount is to build a triangle from each cell in the ground and robot heightmap $\boldsymbol{p}_{heightmap} = (x_{heightmap}, y_{heightmap})$ and the rotation axis given by its origin $\boldsymbol{p}_{rotation}$ and its normalized direction $\hat{\boldsymbol{v}}_{rotation}$.

The points in the heightmaps both lie on a plane with the rotation axis as normal. The point where the rotation axis crosses the plane marks the third point in the triangle (see figure 4.8).

The length of the triangle's adjacent side $d_{heightmap}$ is given by the distance to the rotation axis in the 2-dimensional heightmap. The distance can be calculated by transforming the rotation axis to the origin and forming the dot product between the (normalized) orthogonal $\hat{\boldsymbol{n}}_{rotation}$ to the direction of the rotation axis and the transformed $\boldsymbol{p}_{heightmap}$.

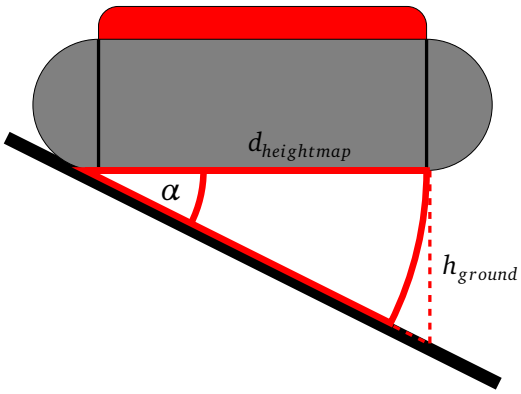


Figure 4.8.: Rotation on a ramp with a slope of α.

$$\hat{\boldsymbol{n}}_{rotation} = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \cdot \hat{\boldsymbol{v}}_{rotation}$$

$$d_{heightmap} = \hat{\boldsymbol{n}}_{rotation} \cdot (\boldsymbol{p}_{heightmap} - \boldsymbol{p}_{rotation})$$

With h_{ground} given by the difference in the two heightmaps:

$$h_{ground} = HM_{robot}(x_{heightmap}, y_{heightmap}) - HM_{ground}(x_{heightmap}, y_{heightmap})$$

Finally, the rotation $\tilde{\alpha}$ can be estimated as:

$$\tilde{\alpha} = \text{atan2}(h_{ground},\ d_{heightmap})$$

This is done for each cell in the heightmap, and the minimal $\tilde{\alpha}$ is used.

It should be noted that this naive approach is slightly inaccurate, as can be seen in figure 4.8. Rotating the robot changes the position of the evaluated cell in the heightmap. In the depicted case where both the robot and the ground can be approximated by planes with a constant slope, this is not an issue, but in real maps, with increasing true angle α, this can lead to wrong results.

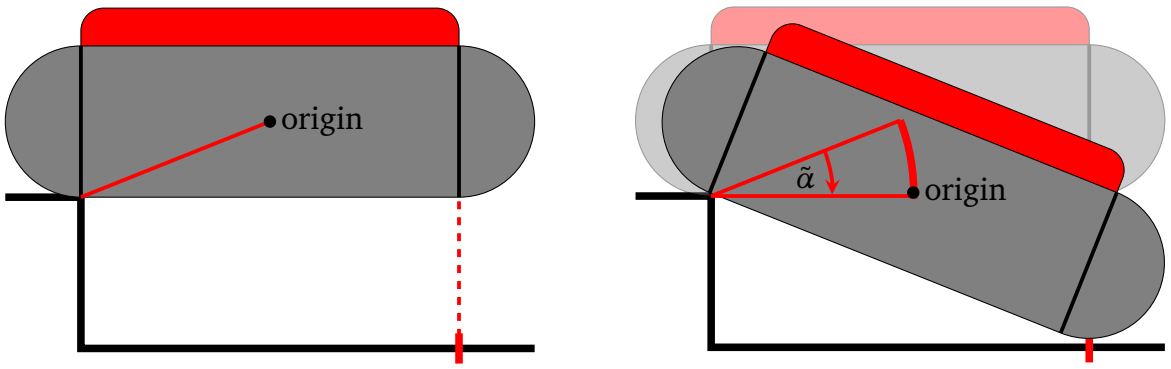
Rotating the robot In effectively all cases, the rotation axis does not pass through the robot's origin, and therefore, a simple rotation of the robot is insufficient. For accurate results, the translation of the origin caused by the axis rotation also has to be included.

The position of the displaced origin can be found by transforming it into the rotation frame S, applying the rotation $\tilde{\alpha}$ around the rotation axis and transforming it back into the world frame.

$$
\begin{aligned}
{}^{S}\boldsymbol{p}_{origin} &= {}^{S}T_{U} \cdot {}^{U}\boldsymbol{p}_{origin} & {}^{S}\boldsymbol{p}'_{origin} &= R(\hat{\alpha}) \cdot {}^{S}\boldsymbol{p}_{origin} \\
{}^{U}\boldsymbol{p}'_{origin} &= {}^{U}T_{S} \cdot {}^{S}\boldsymbol{p}'_{origin} & {}^{W}\boldsymbol{p}'_{origin} &= {}^{W}T_{U} \cdot {}^{U}\boldsymbol{p}'_{origin}
\end{aligned}
$$

where the transformation to the rotation frame is given by translating the rotation axis such that it passes through the origin. A possible transformation from the URDF frame to the rotation frame is given by

$$
{}^{U}R_{S} = \boldsymbol{I} \qquad {}^{U}\boldsymbol{t}_{S} = {}^{U}\boldsymbol{p}'_{rotation}
$$



(a) (b)

Figure 4.9.: Depiction of the example rotation illustrating why a pose update is necessary when rotating around an arbitrary axis.

Example For simplicity, we assume a two-dimensional robot. The formulas above work for both the three-dimensional as well as the two-dimensional case. As can be seen in figure 4.9 (a), in the first step of the pose prediction, the robot's contact point is on top of an edge, and it will subsequently tip over.

The ground contact point is the point that we assume to be in contact with the ground after the rotation. In our example, we would assume 1D heightmaps and a 1D contact map $M_{contact}(x)$ and obtain the ground contact $x_{contact}$ and the corresponding height over the ground $h_{contact}$ as:

$$
\begin{aligned}
x_{contact} &= \min_{a<x<b} \operatorname{atan2}(M_{contact}(x) - \mu_{contact}, x - x_{edge}) \\
h_{contact} &= M_{contact}(x_{contact}) - \mu_{contact}
\end{aligned}
$$

with a and b being the heightmaps lower and higher bound.
Essentially, this means we use the x for which the rotation necessary to touch the ground is minimal. For this example, we use the following values

$$
(x_{edge}, y_{edge}) = (-2.5, -1) \qquad x_{contact} = 2.5 \qquad h_{contact} = -2
$$

The rotation angle is given by:

$$\tilde{\alpha} = \text{atan2}\left(h_{contact}, x_{contact} - x_{edge}\right) = \text{atan2}\left(-2,\ 5\right) \approx -21.8$$

Given $\tilde{\alpha}$, the origin can be transformed, and the new origin is obtained as

$${}^{S}\boldsymbol{p}_{origin} = \begin{bmatrix} 2.5 \\ 1 \end{bmatrix} \qquad {}^{S}\boldsymbol{p}'_{origin} = \begin{bmatrix} cos(\tilde{\alpha}) & -sin(\tilde{\alpha}) \\ sin(\tilde{\alpha}) & cos(\tilde{\alpha}) \end{bmatrix} \cdot {}^{S}\boldsymbol{p}_{origin} \approx \begin{bmatrix} 2.693 \\ 0 \end{bmatrix}$$

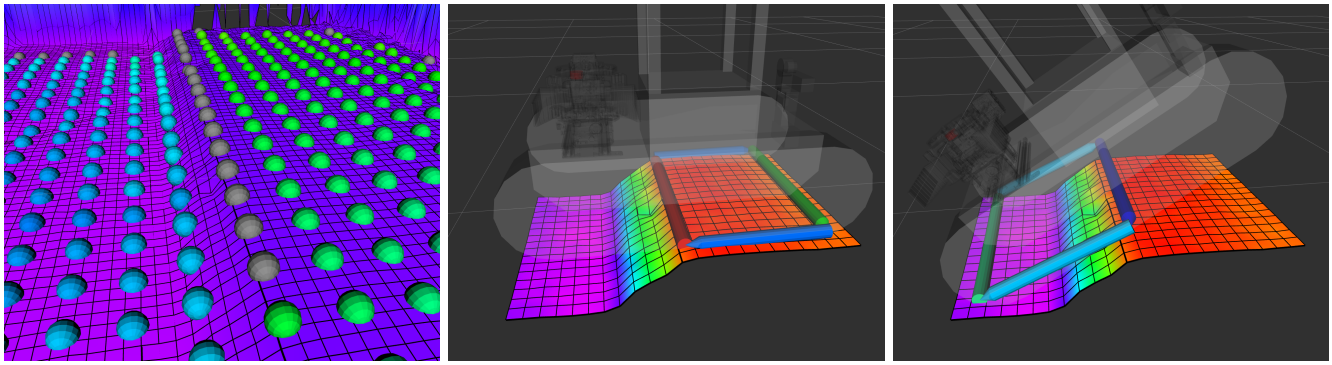
$${}^{U}\boldsymbol{p}'_{origin} = {}^{S}\boldsymbol{p}'_{origin} + {}^{U}\boldsymbol{t}_{S} = \begin{bmatrix} 0.193 \\ -1 \end{bmatrix}$$

Hence, the robot's pose has to be moved by 0.193 in x-direction.

Examining fig. 4.9 closer, it is evident that the rotated robot does not actually touch the ground. This is due to the section of the robot for which the distance was calculated not actually being the part that will touch the ground as explained earlier and illustrated in fig. 4.8

4.3.6. Tipping over the least stable axis

So far, only the stability for a stable pose has been considered. In complex environments, this is insufficient. Many obstacles require tipping over, e.g., ramps or edges. If the pose after the tipping occurred is stable, the tipping may be intended. This problem is illustrated in figure 4.10, where an otherwise viable path straight across the obstacle is deemed not traversable because the pose at the edge of the obstacle has very low stability.



(a) (b) (c)

Figure 4.10.: (a) An example of the limitations encountered if tipping over instabilities is not considered. Colored nodes are reachable, and unreachable nodes are grey. Due to the unstable axis shown in (b), there exists no direct path from the left onto the obstacle. (c) If the robot were to tip over the axis all axes except the rotation axis would be stable.

To address this problem, the stability estimate for each axis is refined by tipping the robot over each axis. For the tipped pose the stability is estimated, and if the minimal stability of all axes except the

rotation axis is greater than the stability of the rotation axis, this stability is assumed for that axis. In essence, the stability returned is the minimal stability of all axes over which the robot may not tip.

4.4. Planning

This section explains how the concepts developed in the previous sections are used to plan stable paths. First, an overview of the overall solution structure is given before examining the steps in detail.

4.4.1. General structure

There are two different types of planning presented in this work. The first is planning from a start to a goal position. The other is exploration planning which plans a path from a starting position to any location fulfilling a previously specified objective function. As an example, let us imagine the robot platform is deployed in an unknown building to build a map for the first responders. In that case, the robot should drive to areas it has not yet mapped. This is done using exploration planning and using an objective function that is fulfilled if the map at the location is mostly unknown.

As a general first step, the graph is expanded from the start (and if planning to a goal, from the goal position simultaneously) to check if the goal or a goal node is reachable. If that is not the case, there is no need to evaluate the stability for a non-existent path.

Assuming that the goal state is reachable, the planning starts with the start location and expands using the Dijkstra algorithm until the goal is reached or all reachable nodes have been visited. The Dijkstra algorithm was chosen because of its simplicity and because the focus of this work is mainly on the efficient pose prediction and stability estimation. Replacing the graph-search algorithm is straightforward without any significant modifications. In case that the goal is reached, the path is returned. Otherwise, a planning failure is reported.

4.4.2. Expansion

The first reachability analysis is expanding the graph solely on the basis that the map at the expanded node is known, which means the value at (or at one of the cells bordering) the target cell has to be valid, and the node is not filtered if a node filter is in use. In this work, as default, a node filter is used that filters based on the distance to height differences using a very conservative threshold of 30 cm for the height difference.

A node is expanded by checking in each of the discretized directions on the local x-y-plane if there is a valid node in that direction and if there is, adding that neighbor to the list of nodes to be expanded. The number of directions

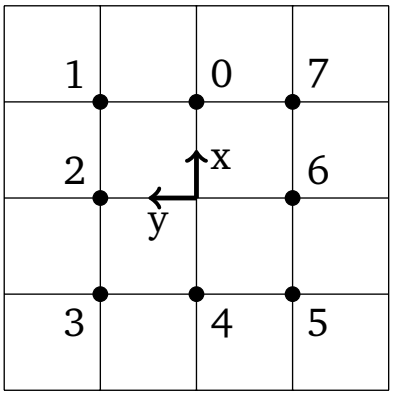


Figure 4.11.: Local directions for the case of 8 discretized directions viewed from above.

in this work was set to 8, and as indicated in figure 4.11, the directions point from 0 in positive x-direction counterclockwise in increments of 45 degrees.

4.4.3. Planning

The planning was split up in two independent but connected parts. One is the algorithm used to determine the node that is expanded, such as Dijkstra or A*. The other is the expansion of the node and the calculation of costs to get from one state to another. This split was chosen to allow a simple exchange of the search algorithm used.

Planning Structure

The entire planning process is outlined in procedure A.1. Since this a standard planning procedure, I will not describe the structure here. Instead, we will focus on a significant problem that arises when switching from shortest-distance 2D planning to 3D planning with multiple cost functions, namely, the robot's orientation.

In 2D planning with distance and optionally a locally monotone cost term such as a distance to obstacle penalty, the orientation can often be ignored because it is implicitly forced by the path and the path of lowest cost is usually a mostly straight valley.

With an increasingly noisy cost term, the planning algorithm has more incentive to switch directions seemingly arbitrarily. At that point, it is obligatory that the cost for the rotations are taken into account. This, however, requires knowledge about the robot's orientation. The straightforward solution would be to expand the search space and instead of having a search space defined by (x, y, z), for a ground-bound robot it expands to (x, y, z, θ) where θ is the rotation around the z-axis.

Rather than expanding the search space and increasing the memory consumption by a factor of the resolution of θ, the orientation can also be captured by extending the node's history. Instead of only storing a node's predecessor, the pre-predecessor is also stored. This implicitly represents the rotation necessary at the predecessor without a need to explicitly add the orientation to the search space.
The downside is that the complexity of the evaluation of neighbor nodes increases (see proc. 4.2). The evaluation of the cost from the node to its neighbor remains the same. If the cost is higher than the current lowest-cost path to the neighbor, we still have to check the neighbor's neighbors because including rotation costs it might be cheaper to reach one of them when coming from the node via the neighbor. Hence, for each neighbor, the cost with the neighbor as predecessor and node as pre-predecessor has to be evaluated and if the cost is lower, the lowest-cost path is set to go via node as pre-predecessor and neighbor as predecessor.

The concept of implicit orientations is best explained by looking at an example. Figure 4.12 shows an example graph problem with the cost for each edge annotated on the right of the arrow. The objective is to get to I from A with minimal cost. Without rotation costs, the solutions will only use the vertical and horizontal edges. Now, we view the graphs as locations on a grid, i.e., G is two steps

Procedure 4.2 Implicit Orientation

```
1: for direction ∈ Directions do
2:   neighbor ← expand from node in direction
3:   c ← total cost to neighbor via node coming from predecessor(node)
4:   if neighbor is newly created or c < cost(neighbor) then
5:     cost(neighbor) ← c
6:     predecessor(neighbor) ← node
7:     prepredecessor(neighbor) ← predecessor(node)
8:     A.insertOrUpdate(neighbor, cost(neighbor))
9:   else
10:    {May still be cheapest to go to the neighbor's neighbor}
11:    for direction ∈ Directions except for the direction it's comming from do
12:      next_neighbor ← expand from neighbor in direction
13:      next_c ← total cost to next_neighbor via neighbor coming from node
14:      if next_neighbor is newly created or next_c < cost(next_neighbor) then
15:        cost(next_neighbor) ← next_c
16:        predecessor(next_neighbor) ← neighbor
17:        prepredecessor(next_neighbor) ← node
18:        A.insertOrUpdate(next_neighbor, cost(next_neighbor))
19:      end if
20:    end for
21:  end if
22: end for
```

in x-direction of A and C is two steps in y-direction of A. If we arrive at C coming from B, we are facing in y-direction and have to turn to get to F. Hence, we need an additional rotation cost, e.g., turning 90° may have a cost of 6.

Starting at A looking in the direction of B, the cost of the minimal cost path for each node is given in table 4.1. The path to I with minimal cost ist now *A, E, I* with a cost of 9 due to a 45° rotation at the start. The path via C, for example, now has a cost of 10. While this shows that the rotation costs influence the path of minimal cost, it does not explain the concept of the implicit rotation cost. For this, let us have a look at the path from A to H. The path *A, E, H* has a cost of 10, but the path of minimal cost has a cost of 9.

Node	A	B	C	D	E	F	G	H	I
Cost	0	1	2	7	6	9	8	9	9

Table 4.1.: Costs for each node starting from A looking in the direction of B.

Exemplified, the search using Dijkstra would start with expanding A. B can be reached with a cost of 1, E requires a 45° turn and consequently a cost of 6. Next, B is expanded from where C can be reached with a total cost of 2. The cost to reach E from B is 8 due to the required 90° turn, which is

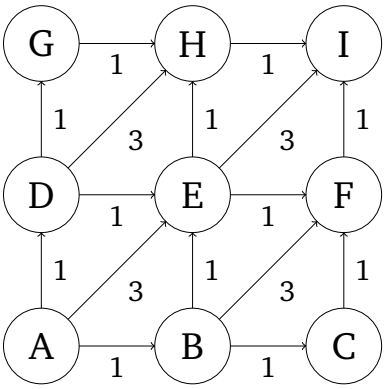


Figure 4.12.: A basic graph problem to illustrate implicit orientations.

higher than the path of minimal cost *A, E*. However, we still have to check E's neighbors, and there we find that we can reach H without another turn for a total cost of 9 if we are coming from B.

Cost Terms

While the pose prediction is the crucial component for stable paths, the cost terms are vital in order to obtain *good* paths. The graph search returns a path that is optimal for the given problem and therefore, the given cost functions.

To select appropriate cost terms, the evaluation criteria that make up a good path have to be determined, and cost terms that ensure high scores for these criteria are chosen. In this work, the focus lies on a trade-off between time-efficiency and safety. Other works cover criteria such as energy-efficiency or sensor-coverage. Safety itself, however, is a broad term and includes a large variety of possible cost terms such as the apparent tip-over risk but also the impact when purposefully tipping over an edge or environmental hazards such as puddles. Many of these cost terms require semantic knowledge about the world.

The safety aspect, is in this work, reduced to tip-over risk, which is minimized by using a cost term that increases with decreasing stability obtained by the *Force-Angle stability measure*. To allow for better comparison and portability between different robot platforms, the stability measure is normalized using the stability on flat ground.

Whereas slight perturbations at high stability should not make a significant difference, at lower stabilities, slight improvements should be strongly encouraged. A safety margin of $\xi_0 = 0.3$ is used to prevent the planning of unstable paths due to inaccuracies. Any stability smaller or equal to this value is treated as non-traversable. As a cost function an inverse quadratic function of the form

$$c_{\text{stability}}(\xi) = \frac{a}{\xi^2} + b$$

was chosen. The parameters a and b are fixed by the constraints $c_{\text{stability}}(\xi_{\text{min}}) = 1$ and $c_{\text{stability}}(1) = 0$. The minimal stability was chosen as $\xi_{\text{min}} = 0.5$ resulting in the cost function

$$c_{\text{stability}}(\xi) = \begin{cases} 0 & if\ \xi \geq 1 \\ \frac{1}{3 \cdot \xi^2} - \frac{1}{3} & if\ \xi > \xi_0 \\ \infty & otherwise \end{cases}$$

The stability chosen is the stability of the pose at the target location for edge costs and the minimal stability of all discrete intermediate orientations when rotating.

The time-efficiency is determined by the estimated time an action (move or rotate) takes. If the robot moves forward, the time taken is given by the distance d divided by its forward velocity v. For rotations, it is given by the rotation angle θ divided by the robot's angular velocity ω.

$$t_{\text{move}}(d) = \frac{d}{v} \qquad\qquad t_{\text{rotate}}(\theta) = \frac{\theta}{\omega}$$

The costs are weighted by the time the action takes to make actions unbiased. Otherwise, safety cost for rotations and move actions as well as edges of differing length would not be comparable. The overall cost is a weighted combination of time and safety cost with the time-safety trade-off parameter δ. A δ of 0 will largely ignore the time or length of the path, whereas a δ of 1 will focus on time only, ignoring any finite safety cost.

$$c_{\text{move}}(d, \xi) = t_{\text{move}}(d) \cdot \left(\delta + (1 - \delta) \cdot c_{\text{stability}}(\xi)\right)$$
$$c_{\text{rotate}}(\theta, \xi) = t_{\text{rotate}}(\theta) \cdot \left(\delta + (1 - \delta) \cdot c_{\text{stability}}(\xi)\right)$$

5. Implementation

The implementation was done using C++ and the *Eigen* framework for vector and matrix operations. Floating-point operations were done with 32-bit IEEE 754 single-precision numbers which can be changed with a single type definition.

The codebase was split into three packages: The base library containing all the relevant code for pose prediction, map integration, planning, and commonly used convenience functions, the planning server providing a *ROS* interface for integration with our robot platform and a demo package containing several example executables, maps, and a robot URDF.

5.1. World Heightmap

The implementation of the bag of heightmaps world representation was called a *World Heightmap*. It was implemented as a collection of heightmaps where each entry contains, additionally to the heightmap, the position on the x-y-plane (the value of the heightmap is always seen as relative to z=0) and two parameters describing the slope of a fitted plane in x- and y-direction. These slope parameters are used for the map integration to determine whether a new segment should be added to this entry or branch off into a new one.

Each entry has two additional floating-point arrays of the same size as the heightmap to allow the tracking of updates. One represents a back-buffer that is used to determine the differences when the map update is finished. If the difference exceeds a threshold, a map update event is sent to subscribers describing which part of the map was updated, i.e., the start row and column along with the row and column count containing all indices for which the difference exceeded the threshold. The second map is responsible for tracking the differences that do not exceed the threshold. This is done to make sure multiple updates below the threshold accumulating a difference greater than the threshold are propagated as well.

For offline analysis, debugging and driving in a known static environment, a file format was developed allowing the map to be saved to and loaded from a file (see A.1).

5.2. Robot Model

The robot model abstracts required information about the robot such as the bounding box, the center of mass, and the heightmap generation. It also provides caching functionality to speed up planning times significantly. A concrete implementation based on URDFs is provided and is described in this section.

5.2.1. Robot Heightmap Generation

The heightmap is stored as a 2D array with the indices ranging from $(0,0)$ to $(rows, columns)$. Hence, the transformation has to be slightly modified to move the bounding box to the image plane spanning from $(0,0)$ to $(rows, columns)$. This transformation is given by the translation

$$ {}^{H}\boldsymbol{t}_{U} = -\begin{bmatrix} BB_{center,x} \\ BB_{center,y} \\ BB_{min,z} \end{bmatrix} + \frac{\rho}{2}\begin{bmatrix} rows-1 \\ columns-1 \\ 0 \end{bmatrix} $$

The second term moves the robot by half of the image in the URDF coordinate system.

As described in 4.3.1, first, the size of the resulting image is obtained by calculating the bounding box from the URDF for the given joint configuration.

Next, for each geometry of each link, the bounding box in the image reference system is calculated to estimate the affected pixel region and the projection ray $\boldsymbol{r}$ is transformed from straight up the z-axis in the heightmap reference system H into the geometry reference system G. The latter is due to the collision and distance formulas simplifying to a great extent when the shapes are at the origin.

For each pixel in the predetermined region, the distance from the geometry to the image plane is calculated. The following section details the formulas used for the different shapes except for meshes, which are not currently handled.

Apart from the projection ray, all shapes have in common that the location of the heightmap pixel in the geometry reference system is required which is obtained using

$$ {}^{G}\boldsymbol{p} = {}^{G}T_{H} \cdot \begin{bmatrix} x_{2D} \cdot \rho \\ y_{2D} \cdot \rho \\ 0 \end{bmatrix} $$

with x_{2D} and y_{2D} being the pixel coordinates and ${}^{G}T_{H}$ obtained as the inverse of

$$ {}^{H}T_{G} = {}^{H}T_{U} \cdot {}^{U}T_{G} $$

where ${}^{U}T_{G}$ is the absolute transform from the URDF's origin to the geometry.

Distance Calculation

Box To calculate the distance to a cuboid, the minimal distance for each side has to be checked.

$$\min_{i\in\{1,2,3\}} d_i = \begin{cases} \min\left\{\frac{{}^{G}\boldsymbol{b}_i - {}^{G}\boldsymbol{p}_i}{{}^{G}\boldsymbol{r}_i}, \frac{-{}^{G}\boldsymbol{b}_i - {}^{G}\boldsymbol{p}_i}{{}^{G}\boldsymbol{r}_i}\right\} & , \text{if } {}^{G}\boldsymbol{r}_i \neq 0 \\ \infty & , \text{otherwise} \end{cases} \quad \text{s.t.} \quad {}^{G}\boldsymbol{p} + d_i \cdot {}^{G}\boldsymbol{r} \leq {}^{G}\boldsymbol{b}$$

where ${}^{G}\boldsymbol{b}$ is a vector with half the side-length for each dimension. A cuboid from $(-1,-2,-3)$ to $(1,2,3)$ would have the associated ${}^{G}\boldsymbol{b} = \begin{bmatrix}1 & 2 & 3\end{bmatrix}^T$.

Cylinder For cylinders, there is a simple case and a more expensive case. If the transformed ray is parallel to the z-axis, we have the simple case and only have to check if the ray passes through the circular projection on the x-y-plane and get the distance to the top and bottom of the cylinder.

$$d = \begin{cases} \min\left\{\frac{l - {}^{G}\boldsymbol{p}_z}{{}^{G}\boldsymbol{r}_z}, \frac{-l - {}^{G}\boldsymbol{p}_z}{{}^{G}\boldsymbol{r}_z}\right\} & , {}^{G}\boldsymbol{p}_x^2 + {}^{G}\boldsymbol{p}_y^2 \leq r^2 \\ \infty & , \text{otherwise} \end{cases}$$

If the transformed ray is not parallel to the z-axis, first, intersections with a cylinder of infinite length are calculated

$$\left\| {}^{G}\boldsymbol{p}_{1:2} + d \cdot {}^{G}\boldsymbol{r}_{1:2} \right\|_2 = r$$

This has to be solved for d

$$\begin{aligned} \left\| {}^{G}\boldsymbol{p}_{1:2} \right\|_2^2 + 2 \cdot d \cdot {}^{G}\boldsymbol{p}_{1:2}^T \cdot {}^{G}\boldsymbol{r}_{1:2} + d^2 \cdot \left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^2 &= r^2 \\ d^2 \cdot \left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^2 + 2 \cdot d \cdot {}^{G}\boldsymbol{p}_{1:2}^T \cdot {}^{G}\boldsymbol{r}_{1:2} &= r^2 - \left\| {}^{G}\boldsymbol{p}_{1:2} \right\|_2^2 \\ d^2 + \frac{2 \cdot d \cdot {}^{G}\boldsymbol{p}_{1:2}^T \cdot {}^{G}\boldsymbol{r}_{1:2}}{\left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^2} &= \frac{r^2 - \left\| {}^{G}\boldsymbol{p}_{1:2} \right\|_2^2}{\left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^2} \\ \left(d + \frac{{}^{G}\boldsymbol{p}_{1:2}^T \cdot {}^{G}\boldsymbol{r}_{1:2}}{\left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^2} \right)^2 &= \frac{r^2 - \left\| {}^{G}\boldsymbol{p}_{1:2} \right\|_2^2}{\left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^2} + \frac{\left({}^{G}\boldsymbol{p}_{1:2}^T \cdot {}^{G}\boldsymbol{r}_{1:2} \right)^2}{\left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^4} \\ d_{1,2} &= \pm \sqrt{\frac{r^2 - \left\| {}^{G}\boldsymbol{p}_{1:2} \right\|_2^2}{\left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^2} + \frac{\left({}^{G}\boldsymbol{p}_{1:2}^T \cdot {}^{G}\boldsymbol{r}_{1:2} \right)^2}{\left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^4}} - \frac{{}^{G}\boldsymbol{p}_{1:2}^T \cdot {}^{G}\boldsymbol{r}_{1:2}}{\left\| {}^{G}\boldsymbol{r}_{1:2} \right\|_2^2} \end{aligned}$$

with d_1 being the solution for the positive square root and d_2 for the negative square root.

There are three cases relevant for the distance calculation:

1. The ray hits the cylinder's side

   $$\left| {}^{G}\boldsymbol{p}_z + d_2 \cdot {}^{G}\boldsymbol{r}_z \right| \leq l$$

   The distance is obtained as d_2

2. The ray passes through the top or bottom. If

$$\begin{aligned} {}^{G}\boldsymbol{p}_z + d_2 \cdot {}^{G}\boldsymbol{r}_z \geq 0 \quad \text{and} \quad {}^{G}\boldsymbol{p}_z + d_1 \cdot {}^{G}\boldsymbol{r}_z < l, \text{ or} \\ {}^{G}\boldsymbol{p}_z + d_2 \cdot {}^{G}\boldsymbol{r}_z \leq 0 \quad \text{and} \quad {}^{G}\boldsymbol{p}_z + d_1 \cdot {}^{G}\boldsymbol{r}_z > -l \end{aligned}$$

   In that case, we have to solve for the distance to the top/bottom of the cylinder.

$$\left| {}^{G}\boldsymbol{p}_z + d \cdot {}^{G}\boldsymbol{r}_z \right| = l \qquad \min d = \frac{\pm l - {}^{G}\boldsymbol{p}_z}{{}^{G}\boldsymbol{r}_z}$$

3. The ray misses the cylinder. Hence, the distance is ∞.

Sphere The distance to a sphere is given by the relation

$$|| {}^{G}\boldsymbol{p} + d \cdot {}^{G}\boldsymbol{r} ||_2 = r$$

which solved for d results in

$$d = \pm \sqrt{\frac{r^2 - \left\| {}^{G}\boldsymbol{p} \right\|_2^2}{\left\| {}^{G}\boldsymbol{r} \right\|_2^2} + \frac{\left({}^{G}\boldsymbol{p}^T \cdot {}^{G}\boldsymbol{r} \right)^2}{\left\| {}^{G}\boldsymbol{r} \right\|_2^4}} - \frac{{}^{G}\boldsymbol{p}^T \cdot {}^{G}\boldsymbol{r}}{\left\| {}^{G}\boldsymbol{r} \right\|_2^2}$$

with the projection ray ${}^{G}\boldsymbol{r}$ being normalized, i.e., $\left\| {}^{G}\boldsymbol{r} \right\|_2 = 1$, the equation simplifies to

$$d = \pm \sqrt{r^2 - \left\| {}^{G}\boldsymbol{p} \right\|_2^2 + \left({}^{G}\boldsymbol{p}^T \cdot {}^{G}\boldsymbol{r} \right)^2} - {}^{G}\boldsymbol{p}^T \cdot {}^{G}\boldsymbol{r}$$

since the minimal distance is needed, only the case of the negative square root has to be evaluated. If the ray does not intersect with the sphere, the expression inside the square root is negative.

The heightmap contains the minimum calculated distance for each cell. If the ray for a cell does not hit any of the considered geometries, the cell's value is set to *NaN*[1].

5.2.2. Support Polygon

A single threshold proved to be unfavorable for accurate results. Where a conservative threshold would discriminate many points that could be considered within the margin of error and result in many iterations or even lead to no convergence at all since the approximation of the rotation angle with heightmaps has limited accuracy dependant on the resolution, a lax threshold would include too many scattered points leading to incorrect rotation axes.

Hence, the support polygon is calculated for two thresholds. First, it is calculated for a lax threshold to estimate the stability. If the pose is not stable, the process is repeated for a strict threshold for an accurate rotation estimate.

[1] **N**ot **a** **N**umber is a special value declaring the value stored invalid, undefined or not representable. Most operations involving a *NaN* value return *NaN* themselves, and comparisons with *NaN* are always false.

5.2.3. Planning

The path planning was split into three independent modules connected via traits[2]: *Graph*, *Planner*, and *Algorithm*. Following a summary of each, the three modules are detailed. The *Graph* encodes a graph in terms of a two-manifold. It has to provide methods to access a node's immediate neighbors on the x-y-plane. The *Algorithm* is the search algorithm used by the planner. It is responsible for storing the nodes that have not yet been expanded alongside with their respective costs and proposes which node to expand next. Finally, the *Planner* connects to the interfaces of the *Graph* and *Algorithm* and implements the actual planning.

The *Graph* module is the most fundamental building block. It allows the planner to not rely on any implementation details other than the general structure imposed in this work. Provided are methods to obtain the closest node for a given 3D-coordinate, a method to query all neighbors on the x-y-plane and to obtain a node's location in the world. By abstracting these methods, the graph can be used by the planner without restricting the graph to a specific implementation. For example, it is up to the implementation whether the graph nodes contain their location or if it is implicitly encoded in the storage structure. The number of directions is also not fixed but provided as a compile-time constant in the graph implementation. In this implementation, the number of directions was set to eight.

The *Algorithm* is the brain of the planning process. It determines in which order the graph nodes are expanded. To encapsulate the algorithm from the *Planner*, interfaces are provided to add a new node, update the cost for a node, check if there are any nodes left in the queue and most importantly to request the next node to be expanded. Additionally, the algorithm has a boolean property indicating whether or not the algorithm supports goal planning and another one indicating the support for exploration planning. Some algorithms, such as Dijkstra support both, even though it does not benefit from knowing the goal. Other algorithms such as A* that try to expand more targeted towards the goal to reduce unnecessary computation time need a goal-based heuristic and can not be used for exploration planning without modification.

The *Planner* combines the other two modules. Following the structure described in section 4.4, it uses the *Algorithm* to determine the order of expansion and the interfaces provided by the *Graph* to expand the search graph until a goal is found. Whereas neighbor costs that do not depend on orientation are handled and cached in the graph, the costs that require information about the orientation that is implicitly encoded in the planning process (see 4.4) are handled by the planner.

5.2.4. Optimizations

URDF heightmap computation The heightmap generation accounts for roughly 95% of the computation time if the heightmap is not cached. The following optimizations were made to reduce the planning time.

[2] A template-based interface requiring classes implementing the trait to provide specific interfaces without the small overhead and type restrictions of virtual methods.

The URDF contains a complete description of the robot, including all of its joints and links with the relative transformation to their parent. Not all of them play a role in finding the ground contacts. A black-/whitelist approach was implemented to make use of this. If the whitelist is non-empty, only links in the whitelist are used to generate the heightmap. Otherwise, all links except for any listed in the blacklist are used. To prevent repeated computation of the relative transformations when obtaining the absolute transformation for each link, the absolute transformation for each link is cached.

Finding the rotation angle Instead of evaluating the arctangent for each cell in the contact map, which is a very costly operation, the fact that the arctangent is a monotonic function was used, and the ratio $h_{\text{ground}}/d_{\text{heightmap}}$ was evaluated for each cell. Then, the rotation angle was obtained by taking the arctangent of the minimal value.

$$\hat{\alpha} = \arctan\left(\min \frac{h_{\text{ground}}}{d_{\text{heightmap}}}\right)$$

Pose Prediction Rather than reevaluating the pose in each planning run, the predicted pose including the stability and the support polygon for all orientations are cached in each graph node. If the map changes, nodes within the area of change are reset.

5.2.5. ROS Integration

An interface to the existing software on our robot platform was implemented to make the planner available. Our robot platforms are using the **R**obot **O**perating **S**ystem (*ROS*) as a middleware connecting the different software packages.

The interface was implemented as a set of *Action Server*'s which are a *ROS* provided implementation of cancelable *Remote Procedure Calls* (RPC). In essence, this provides an interface to which another part of the robot software may connect and send a request, for example, the coordinate of a goal position. As long as that other part does not send a cancellation request, the server plans a path to the goal and sends it forward to a lower level *Action Server* that follows the path. Once the path follower has reached the path's final goal position, it sends a success message back to the server which, in turn, sends a success message to its caller.

The planning server has four primary connections to the *ROS* environment. First, it is notified about map updates of our local heightmap mapping server and passes these updates to the integrator that integrates the map into our bag of heightmaps world map. Additionally, it provides two *Action Servers*; one for goal planning and a second one to request exploration paths. Finally, it connects to a path follower *Action Server* as an *Action client*. The path follower *Action Server* is responsible for following the paths that the planner sends it.

Apart from the required connections, it also provides two *Services*, which are non-cancelable *RPC*s. Both *Services* can be called with a file path. The first saves the current world map to the given path and the other loads a world map from the given path.

6. Evaluation

The proposed planner was evaluated on the real robot and in simulation. Whereas experiments on the real robot can demonstrate the ability of this work to run on a robot system in real-time with real sensor data, and the advantages over the 2D planning approaches, it can not evaluate the accuracy of the pose prediction because there currently is no method of obtaining the robot location with the necessary precision. Thus, the quantitative evaluation of the pose prediction was done in simulation.

6.1. Time-Safety Trade-off

The experiments on the robot intend to demonstrate the influence of the time-safety trade-off parameter on the paths returned by the planner. While different parameter settings may change the optimal path, the experiments should also show that they do not limit the robot's planning capabilities if the optimal path is blocked.

Figure 6.1.: (Top Left) The robot has the options to drive across or around the obstacle. (Top Right) The path around the obstacle is blocked, and the robot has to cross the obstacle. (Bottom) The robot should avoid the first obstacle but cross the second.

Three test scenarios were created to test (and demonstrate) the influence of the δ parameter and highlight the difference to the current planning approach. The first scenario is a simple test to demonstrate the two expected solutions depending on the value of the time/safety trade-off parameter.

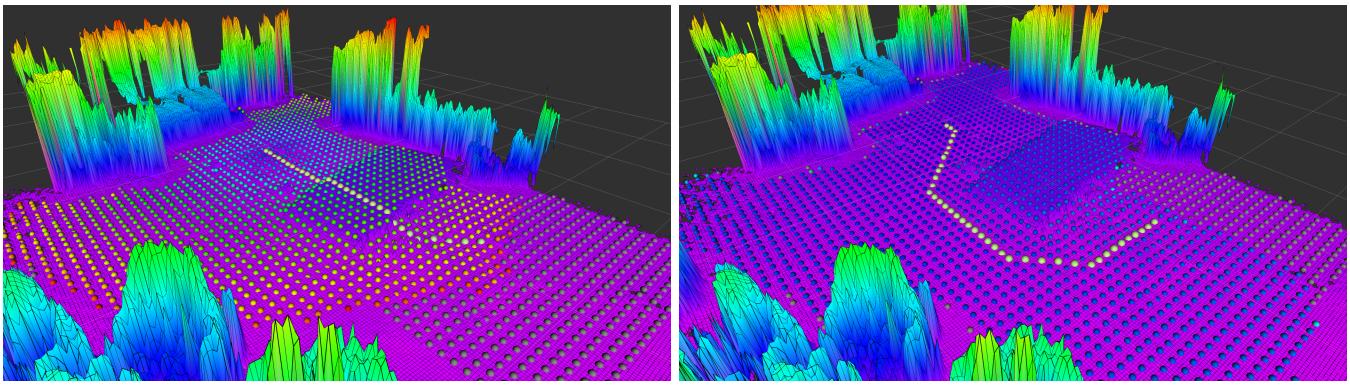


Figure 6.2.: On the left, the planned path for $\delta = 1$ is shown. The grid is colored according to its cost from min to max using a gradient from blue to red through yellow. The path is highlighted in light green with larger spheres. The path with the same start and goal for $\delta = 0.1$ is shown on the right.

The solutions for $\delta = 1$ and $\delta = 0.1$ are shown in figure 6.2. For $\delta = 1$ the time becomes the sole cost factor, and the safety cost only restricts the robot from taking paths that are not traversable without risking tipping over. As expected, the robot takes the path straight across the industrial flooring. For $\delta = 0.1$ the safety becomes the driving cost. It was chosen instead of a δ of 0 because while (for $\delta = 0.1$) the robot will prefer a safe path with high stability, it will not make substantial detours to avoid insignificant safety cost differences due to small inaccuracies in the map and pose prediction.

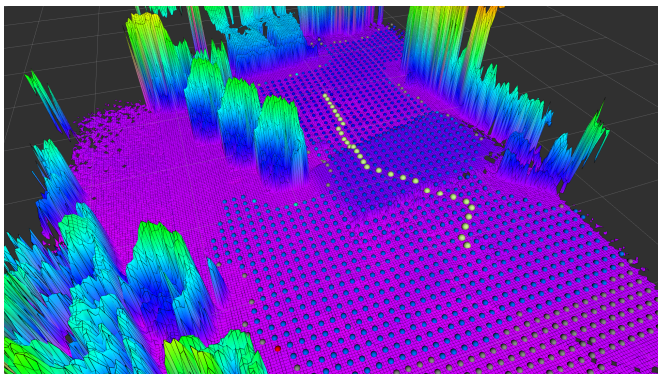


Figure 6.3.: The path planned with $\delta = 0.1$ but this time the path around the obstacle is blocked.

In the second scenario, the alternative path is blocked, and the robot is forced to traverse the industrial flooring. Whereas the two solutions in the first scenario could also be achieved by the current 2D planning algorithm using different thresholds for obstacles, only one of the configurations would be capable of also planning a path in the second scenario. The path for the second scenario is depicted in figure 6.3. It demonstrates that the parameter $\delta = 0.1$ that avoided traversing the obstacle in the first scenario is still able to plan a path if avoiding the obstacle is not possible. This behavior can not be achieved by 2D planning approaches such as our current system[6] (see 3.1).

In the final scenario, the robot is expected to take the longer path around the obstacle and traverse the other obstacle to reach the goal position. This is the first scenario where the current 2D path planning approach would be unable to generate the desired path regardless of the parameterization. Whereas a conservative threshold would force the robot to avoid the first obstacle, it would mean that the planner could not traverse the second obstacle. A more lenient threshold would traverse the second obstacle but would not avoid the first obstacle. The path depicted in figure 6.4 is planned for $\delta = 0.1$ and avoids the first obstacle while traversing the second due to a lack of other options.

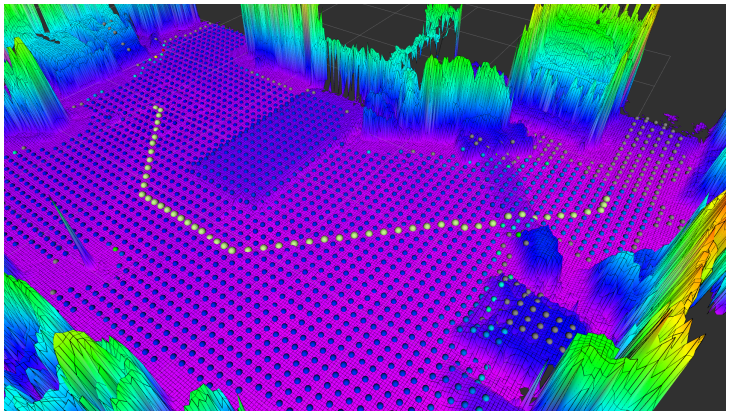


Figure 6.4.: The path for $\delta = 0.1$ in the third scenario.

6.2. Pose Prediction

The accuracy of the pose prediction was evaluated using the robot simulator Gazebo version 9.0 using the default physics simulator based on the Open Dynamics Engine (ODE). It was chosen as a reference because the test setup with the real robot currently lacks a method of obtaining the ground truth of the robot's pose with sufficient accuracy.

The accuracy of the pose prediction was evaluated in two scenarios. Whereas the first one is a very challenging scenario where the pose on several objects in different orientations is predicted, the second scenario is a more realistic scenario where the path of the robot through a simulated NIST (**N**ational **I**nstitute of **S**tandards and **T**echnology) arena was simulated, and the pose was predicted for each grid position the robot passed.

The simulation scenario depicted in figure 6.5 was created to evaluate the accuracy of the predicted pose in a highly challenging scenario. A grid with a resolution of 5 cm was created around each obstacle, and for each location, eight poses rotated around the z-axis in 45° steps were created. For these input poses, a reference pose was obtained by dropping the robot in simulation simulating the fall until the changes of the pose remained smaller than a threshold of 1 mm for the position and for the orientation changes of the quaternion had to remain within 0.001 for half a simulated second. If the robot did not stop moving within a simulated minute, the pose was skipped.

The results of the comparison of the estimated poses with the reference poses are visualized in figure 6.6.

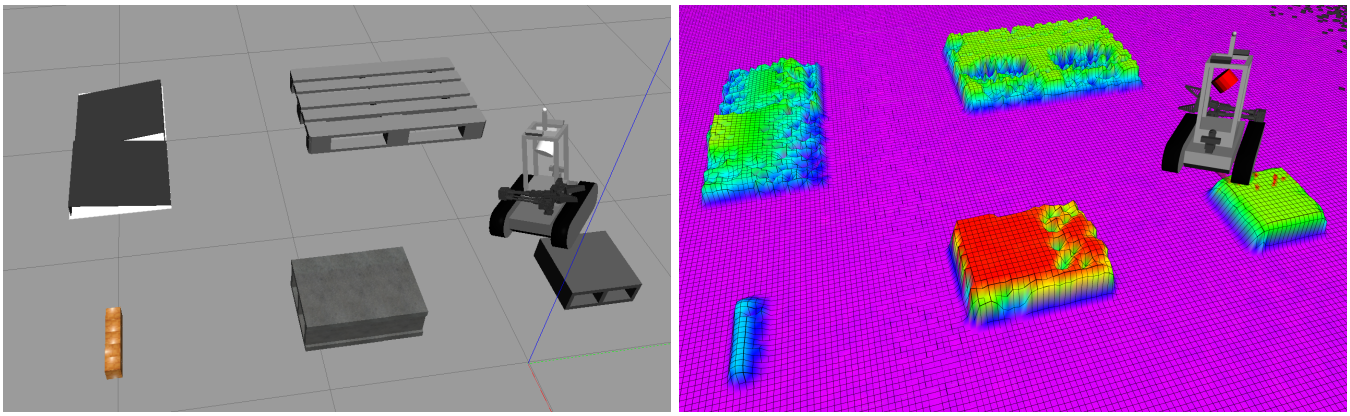


Figure 6.5.: (Left) The simulation scenario consists of five obstacles of different size and slope. (Right) The corresponding world heightmap with a resolution of 2,5 cm.

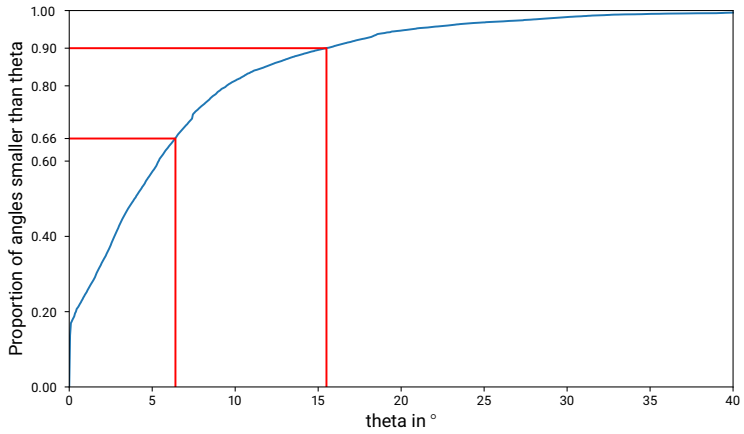


Figure 6.6.: Proportion of poses daviating less than theta in the first pose prediction evaluation scenario.

The graph shows the proportion of poses where the angle between the quaternion[1] estimated by the proposed method and the simulated angle differ less than theta. The angle was calculated in radians based on [21] and converted to degrees – around 66% of the poses where within 6,4° and 90% within 15,5°. For 3% of the poses, the predicted pose had a stability value higher than 0.3 while the simulated pose was not stable. For the evaluation of the orientation, poses that caused the robot to tip-over in the simulation were removed.

The angular errors can be attributed to three error sources. First, the robot can slide in the simulation, and if the drop height is too far from the ground, it may move due to the impact resulting in the robot tipping over the edge differently. Second, the quality of the map significantly influences the results, and as can be seen in figure 6.5, the quality of the map is far from optimal. Lastly, the inherent errors of the pose prediction due to discretization of the contact points and the systematic error of the rotation angle estimation. Despite the magnitude of the angular errors, the proportion of unstable poses that the pose prediction would erroneously predict as stable is in the low single-digit percent.

In the second scenario, the robot drove autonomously through the arena depicted in figure 6.7. The poses were recorded, snapped to a grid of 5 cm, and the closest to each grid location was kept. For each recorded pose, the rotation around the z-axis was extracted, and the pose was estimated based on the rotation around the z-axis and the predicted pose at the previous location.

[1] An extension of the complex numbers that can be used to represent rotations and orientations in three dimensions.

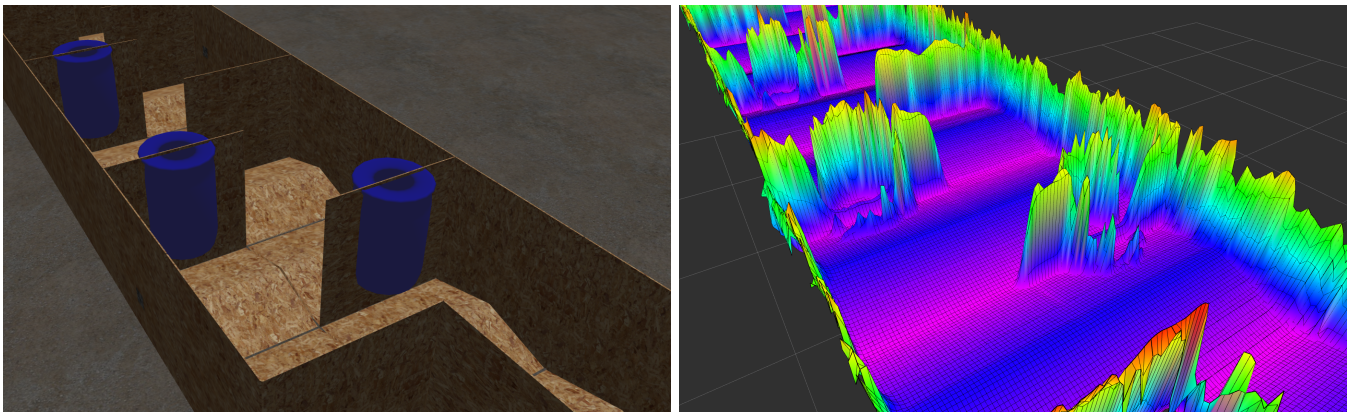


Figure 6.7.: (Left) The simulation scenario consists of a long parcours filled with ramps. (Right) The corresponding world heightmap with a resolution of 2,5 cm.

The results are shown in figure 6.8.

In this scenario, the predicted poses deviated less than 5,5° for 66% of the poses and less than 9,5° for 90% of the poses. Sources for errors in this scenario are the tip-over at the top of the ramp, which is very sensitive to the small change in the position that is necessary to snap the poses to a grid. Another source is that the robot if rotated around the z-axis between two ramps has contact at the opposing diagonal ends, and both the pose rotated clockwise around the diagonal and the pose rotated counter-clockwise are stable.

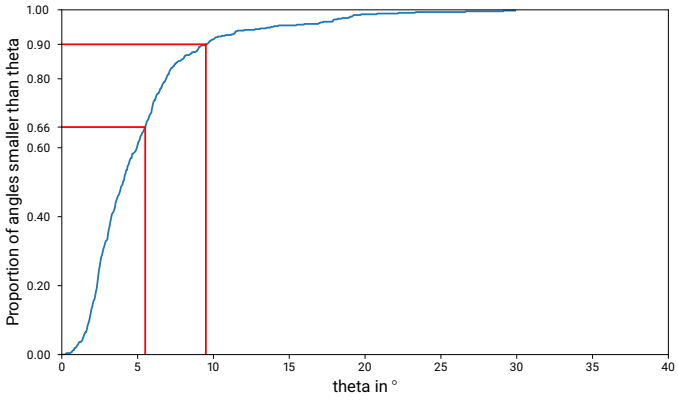


Figure 6.8.: Proportion of poses deviating less than theta in the second pose prediction evaluation scenario.

6.3. Runtime

The runtime evaluation was performed on an i7-8550U 15 W quad-core with a maximum frequency of 4 GHz, which is a slightly lower clocked variant of the processor used on our robot. First, runtime statistics for the previous pose prediction evaluation that predicted the pose for 21824 input poses are presented. Then the runtime is evaluated for different graph resolutions.

6.3.1. Pose Prediction

The pose prediction took on average around 243 µs with a 1σ value of rounded 179 µs. The significant variance is due to the caching of the heightmaps. In some cases, the heightmaps for multiple orientations have to be calculated. On flat ground, the heightmap is already cached, and the pose prediction finishes in a single step. Overall, around 56% of the heightmaps requested were previously cached.

Average	Longest	Shortest
242,912 µs ±178,891 µs	2027,734 µs	4101 ns

Table 6.1.: Runtime statistics for the first pose prediction evaluation scenario.

6.3.2. Path Planning

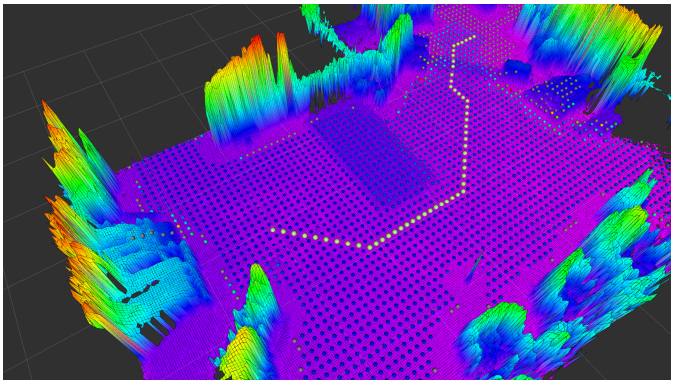
The path planning runtime was evaluated on the map created for the third test scenario using the robot's sensor data, which is approximately 7 m by 5 m. To evaluate the runtime, the path was planned for the resolutions: 10 cm, 20 cm, 30 cm, 40 cm, 50 cm. After the initial planning, the path was replanned multiple times to make leverage of the cached information. The graphs and planned paths are shown in figure 6.9, and the runtimes are summarized in table 6.2.

Resolution	Initial planning	Subsequent runs
10 cm	1,61 s	25,81 ms
20 cm	409,52 ms	4,10 ms
30 cm	184,78 ms	1,85 ms
40 cm	106,64 ms	0,75 ms
50 cm	63,83 ms	0,22 ms

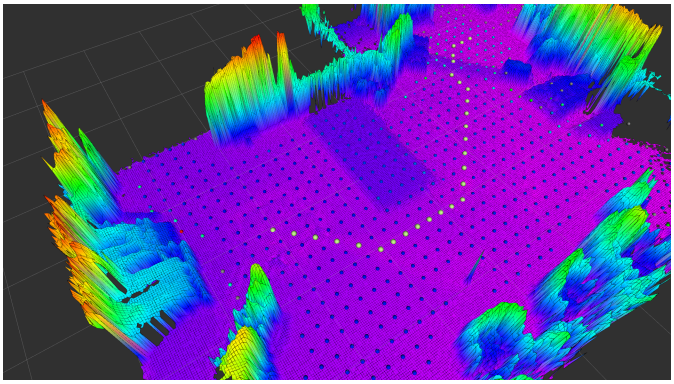
Table 6.2.: Initial and repeated planning times for different graph resolutions.

The planning time was split into initial planning, which is the first planning run where the graph has to be built, and all caches are empty and the repeated planning. The subsequent runs are of particular interest concerning exploration where the map continuously grows. The results of the initial planning time were averaged over 5 runs and 125 runs for the repeated runs. As expected, the runtimes decrease with the square of the resolution since the number of graph nodes for a fixed area is cut by four if the resolution is doubled. In other work, typically half the robot's length is chosen for the graph resolution, which would amount to a resolution of 30 cm for our robot.

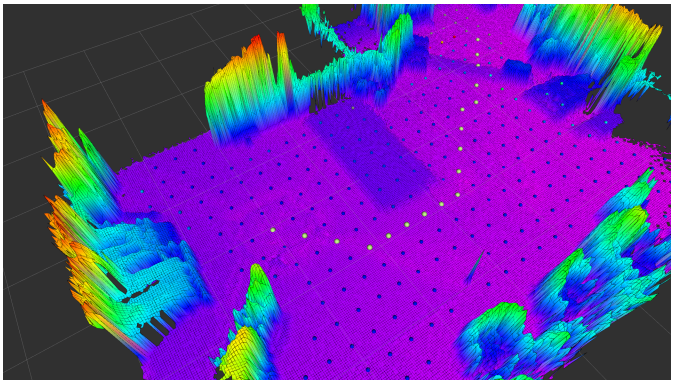
Overall, the results demonstrate that the proposed approach is capable of planning paths in real-time even at a higher resolution than 30 cm.



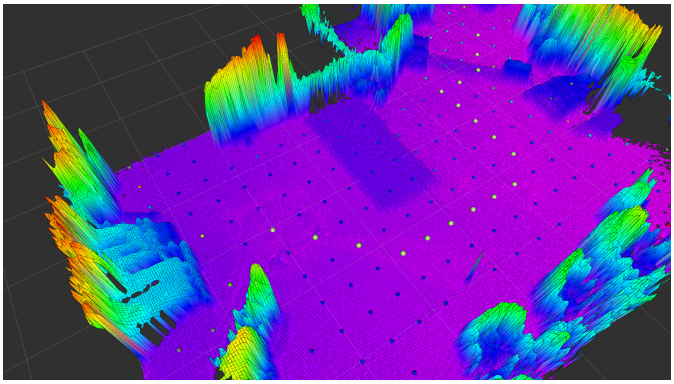
(a) 10 cm



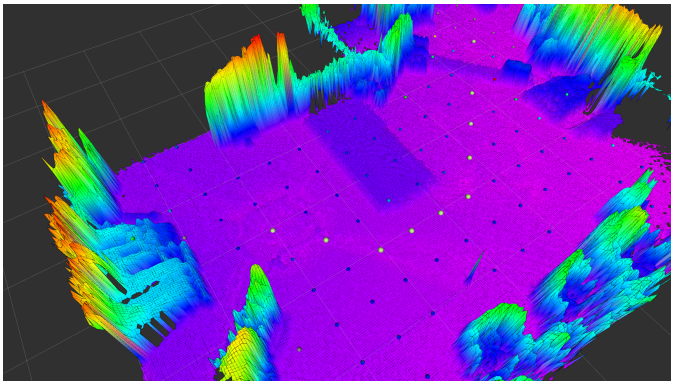
(b) 20 cm



(c) 30 cm



(d) 40 cm



(e) 50 cm

Figure 6.9.: The graphs for the evaluated resolutions. The planned paths are indicated by slightly larger green spheres.

7. Conclusion and Future Work

In this work, an approach for 3D path planning incorporating the robot's stability and capable of running within real-time constraints on the limited resources of a mobile platform has been presented. The accuracy in predicting unstable poses and the real-time feasibility was shown in experiments on the robot and in simulation. The pose prediction is based on the difference between a ground and a robot heightmap, and iterative geometric transformations. Compared to existing approaches, the pose prediction is significantly faster, and as a result, the planning time is vastly reduced. In contrast to most existing approaches, this planning approach does not depend on a specific robot but can comfortably be used with other robots.

In future work, the planning algorithm may be improved by incorporating the dynamic stability and optimizing the robot's velocity along its path for criteria such as time-efficiency versus path deviations and stability. Additionally, the current approach evaluates the stability at fixed grid locations, but the position with the lowest stability might be located in between. The current limitation to a fixed joint configuration may be viewed as undesirable for a reconfigurable robot, and the configuration of the robot could be optimized for prospective paths. This would require an efficient heuristic as current methods are too computationally expensive to be evaluated in the capacity required by online path planning.

Apart from stability aspects, other factors may need to be considered, such as the robot's ability to traverse an edge. Whereas the initial and the goal pose may be stable, the robot hardware may not be able to reach the goal pose coming from its initial pose. The terrain properties are also of significance when planning in rural outdoor environments, and a semantic mapping approach could be employed to add terrain-based costs. Generally, the amount of contact the robot has with the ground could also be considered as a measure favoring locations with strong ground contact. When tipping over an edge, the impact could be considered to avoid risks to the robot's hardware.

The quality of the mapping and thereby, the accuracy of the pose prediction can be improved with a global mapping concept that works well with sparse data and uncertainty. As a different map format, meshes or surfel maps could be evaluated regarding the suitability to estimate the pose in real-time and the capability to create the map in real-time.

Bibliography

[1] inmediahk. *Fukushima, 2011*. [Online; accessed August 2019]. 2013. url: https://www.flickr.com/photos/inmediahk/26630446395.

[2] *inertial.png*. [Online; accessed August 2019]. 2009. url: http://wiki.ros.org/urdf/XML/link.

[3] *joint.png*. [Online; accessed August 2019]. 2012. url: http://wiki.ros.org/urdf/XML/joint.

[4] EG Papadopoulos and Daniel A Rey. "A new measure of tipover stability margin for mobile manipulators". In: *Proceedings of IEEE International Conference on Robotics and Automation*. Vol. 4. IEEE. 1996, pp. 3111–3116.

[5] Chee K Yap. "Algorithmic motion planning". In: *Advances in robotics* 1 (1987), pp. 95–143.

[6] Stefan Kohlbrecher et al. "Hector Open Source Modules for Autonomous Mapping and Navigation with Rescue Robots". In: *RoboCup 2013: Robot World Cup XVII*. Ed. by Sven Behnke et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 624–631. isbn: 978-3-662-44468-9.

[7] S. Wirth and J. Pellenz. "Exploration Transform: A stable exploring algorithm for robots in rescue environments". In: *2007 IEEE International Workshop on Safety, Security and Rescue Robotics*. Sept. 2007, pp. 1–5. doi: 10.1109/SSRR.2007.4381274.

[8] Martin Oehler. "Whole-Body Planning for Obstacle Traversal with Autonomous Mobile Ground Robots". MA thesis. 2018.

[9] Hans Moravec and Alberto Elfes. "High resolution maps from wide angle sonar". In: *Proceedings. 1985 IEEE International Conference on Robotics and Automation*. Vol. 2. IEEE. 1985, pp. 116–121.

[10] Michael Brunner, Bernd Brüggemann, and Dirk Schulz. "Autonomously Traversing Obstacles-Metrics for Path Planning of Reconfigurable Robots on Rough Terrain." In: *ICINCO (2)*. 2012, pp. 58–69.

[11] F. Colas et al. "3D path planning and execution for search and rescue ground robots". In: *2013 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Nov. 2013, pp. 722–727. doi: 10.1109/IROS.2013.6696431.

[12] Federico Ferri et al. "Point cloud segmentation and 3D path planning for tracked vehicles in cluttered and dynamic environments". In: *Proc. of the 3rd IROS Workshop on Robots in Clutter: Perception and Interaction in Clutter*. 2014.

[13] Matteo Menna et al. “Real-time autonomous 3D navigation for tracked vehicles in rescue environments”. In: *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE. 2014, pp. 696–702.

[14] Mohammad Norouzi, Jaime Valls Miro, and Gamini Dissanayake. “Planning Stable and Efficient Paths for Reconfigurable Robots On Uneven Terrain”. In: *Journal of Intelligent & Robotic Systems* 87.2 (Aug. 2017), pp. 291–312. issn: 1573-0409. doi: 10.1007/s10846-017-0495-8. url: https://doi.org/10.1007/s10846-017-0495-8.

[15] Bijo Sebastian and Pinhas Ben-Tzvi. “Physics based path planning for autonomous tracked vehicle in challenging terrain”. In: *Journal of Intelligent & Robotic Systems* (2018), pp. 1–16.

[16] Brian Curless and Marc Levoy. “A volumetric method for building complex models from range images”. In: (1996).

[17] Richard A Newcombe et al. “Kinectfusion: Real-time dense surface mapping and tracking.” In: *ISMAR*. Vol. 11. 2011. 2011, pp. 127–136.

[18] Alan Ettlin, Patrick Büchler, and Hannes Bleuler. “Rough-terrain robot motion planning based on topology and terrain constitution”. In: (Aug. 2019).

[19] Chanoh Park et al. “Elastic LiDAR Fusion: Dense Map-Centric Continuous-Time SLAM”. In: (Nov. 2017).

[20] M. Habbecke and L. Kobbelt. “A Surface-Growing Approach to Multi-View Stereo Reconstruction”. In: *2007 IEEE Conference on Computer Vision and Pattern Recognition*. June 2007, pp. 1–8. doi: 10.1109/CVPR.2007.383195.

[21] Du Q. Huynh. “Metrics for 3D Rotations: Comparison and Analysis”. In: *Journal of Mathematical Imaging and Vision* 35.2 (Oct. 2009), pp. 155–164. issn: 1573-7683. doi: 10.1007/s10851-009-0161-2. url: https://doi.org/10.1007/s10851-009-0161-2.

A. Appendix

List of Figures

List of Tables

Procedure A.1 Planning

Input: A start S, a goal or objective function G and an algorithm A
Output: Path with minimal cost to goal node or failure if none exist

```
1: {Without evaluating cost check if a goal may be reachable}
2: if not checkReachability(S, G) then
3:     return failed
4: end if
5: A.initialize(G)
6: A.insertOrUpdate(S, 0)
7: while not A.empty do
8:     node = A.takeNext()
9:     if node is goal then
10:        path ← [node]
11:        while node has predecessor do
12:            node ← predecessor(node)
13:            path ← [node] ∪ path
14:        end while
15:        return path
16:    end if
17:    for direction ∈ Directions do
18:        neighbor ← expand from node in direction
19:        c ← total cost to neighbor via node
20:        if neighbor is newly created or c < cost(neighbor) then
21:            cost(neighbor) ← c
22:            predecessor(neighbor) ← node
23:            A.insertOrUpdate(neighbor, cost(neighbor))
24:        end if
25:    end for
26: end while
27: return failed {All nodes expanded but goal was not found}
```

# of Bytes	Content	Type	Description
3	"*WHM*"	char[]	Binary encoded string.
1	*RESERVED*		
2	Version	uint16	Version of the stored *World Heightmap*.
2	Minimum Version		The minimum version to which the format is downwards compatible.
24	*RESERVED*		For future use.
4	Resolution	IEEE754 SP	The grid resolution in m
4	Merge distance	IEEE754 SP	
4	Max. submap size	int32	Maximum number of rows/columns of a single heightmap entry.
4	Min. map growth	int32	The minimum amount of grid cells a map grows when resized.
4	Min. map overlap	int32	The minimum overlap in grid cells adjacent heightmaps must have.
4	Branch z-difference	IEEE754 SP	The minimum z-difference from the approximated plane required to branch into a new heightmap.
38	*RESERVED*		For future use.
2	l	uint16	The length of the world frame string.
l	World frame	char[]	ASCII encoded world frame string.
X	Map frames	Map Frame[]	

Map Frame

# of Bytes	Content	Type	Description
4	"*HMAP*"	char[]	Binary encoded string.
4	X	IEEE754 SP	x-coordinate of the heightmap.
4	Y	IEEE754 SP	y-coordinate of the heightmap.
4	Z	IEEE754 SP	z-value of the fitted plane used to determine when to branch of at the map origin.
4	1	IEEE754 SP	The number 1. *Legacy*
12	3×0	IEEE754 SP	Three times the number 0.
4	Plane α	IEEE754 SP	Slope in x-direction of the fitted plane.
4	Plane β	IEEE754 SP	Slope in y-direction of the fitted plane.
24	*RESERVED*		For future use.
4	r	uint32	Number of rows of the stored heightmap.
4	c	uint32	Number of columns of the stored heightmap.
$r \times c \times 4$	Heightmap	IEEE754 SP[]	Heightmap content stored in column major.

Table A.1.: World Heightmap File Format (version 3).